Communication Synthesis for MIMO Decoder Algorithms

Joshua D. Quesenberry

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Cameron D. Patterson, Chair

Michael S. Hsiao

Thomas L. Martin

August 9, 2011

Bradley Department of Electrical and Computer Engineering

Blacksburg, Virginia

Keywords: FPGA, Xilinx, Communication Synthesis, MIMO

Communication Synthesis for MIMO Decoder Algorithms

Joshua D. Quesenberry

(ABSTRACT)

The design in this work provides an easy and cost-efficient way of performing an FPGA implementation of a specific algorithm through use of a custom hardware design language and communication synthesis. The framework is designed to optimize performance with matrix-type mathematical operations. The largest matrices used in this process are 4×4 matrices. The primary example modeled in this work is MIMO decoding. Making this possible are 16 functional unit containers within the framework, with generalized interfaces, which can hold custom user hardware and IP cores.

This framework, which is controlled by a microsequencer, is centered on a matrix-based memory structure comprised of 64 individual dual-ported memory blocks. The microsequencer uses an instruction word that can control every element of the architecture during a single clock cycle. Routing to and from the memory structure uses an optimized form of a crossbar switch with predefined routing paths supporting any combination of input/output pairs needed by the algorithm.

A goal at the start of the design was to achieve a clock speed of over 100 MHz; a clock speed of 183 MHz has been achieved. This design is capable of performing a 4×4 matrix inversion within 335 clock cycles, or 1,829 ns. The power efficiency of the design is measured at 17.15 MFLOPS/W.

Acknowledgments

I would like to thank Dr. Cameron Patterson for his guidance, for being my academic advisor while pursuing this degree at Virginia Tech, and for giving me the opportunity to participate in this project.

I would also like to thank Dr. Tom Martin and Dr. Michael Hsiao for their support, for agreeing to serve on my committee, and for the courses that they taught over the period of my studies at this university, which have taught me so much.

Finally, I would like to thank God along with my family and friends for their continued prayer and support over the many years I have spent in school preparing to reach this point in my education.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Occasionally a hardware implementation of a specific algorithm is required. In an effort to minimize the time and money required to carry out such an implementation, it becomes necessary to move away from custom integrated circuits to configurable circuits. The particular type of configurable circuit used here is known as a Field Programmable Gate Array (FPGA). During the implementation of these specific algorithms, it is not unusual to encounter intellectual property (IP) cores where the most efficient means of interconnecting these cores is through communication synthesis rather than hand-coded connections.

1.1 Motivation

The major design objective of this research was to create a hardware architecture that allows a generic algorithm, operating on a small matrix data structure, to be easily implemented

on an FPGA. Due to a provided overall framework, this device is the "glue" that allows for multiple "black box" IP cores to be integrated together easily. The root of this problem was formulated based on a programmable accelerator for multiple-in-multiple-out (MIMO) wireless communications, created by Karim Mohammed and Babak Daneshrad at the University of California [16], and the need to integrate unknown circuit designs. Mohammed and Daneshrad's design is based on a fixed point number format, has a predetermined set of function units, and has complex routing to and from the memory structure.

1.2 Contributions

This design uses a 32-bit floating point format, an appropriate format for the type of calculations to be done on this device. Using the floating point format allows for greater precision over a wider dynamic range, and the values input to the device are likely to already be in a floating point format. However, one disadvantage of using floating point function units is increased resource utilization on the FPGA.

The design permits user-defined function units; by allowing for this level of user customization, one gains the ability to implement any necessary algorithm. The user even has the ability to add generic function units, as well as the ability to use their own IP cores for algorithm optimization. The function units will be attached to the overall structure through a normalized interface; by doing this, the task of integrating unrelated components together is simpler and will lead to less time spent getting function units to work together.

In order to move data to and from the memory structure, a crossbar switch routing structure has been implemented, allowing for all possible paths of data to be realized. In order to minimize resource overhead, the algorithm is profiled, and the paths required are then stored for use as different modes to configure the crossbar switch. This method reduces the extra bits in the instruction word that would otherwise be needed to steer data.

The memory structure used is optimized for matrices of size 4×4 or smaller by using a memory module for each element of the matrix. The overall memory structure contains 64 individual memory modules, which supports four 4×4 matrices. Normally a system will contain a single-ported monolithic memory, which creates a serious bottleneck. This bottleneck is reduced by having 64 individual memory modules that can be accessed in parallel. Each memory module is dual-port and supports 32-bit floating point data. Each memory module is also designed to hold a designated number of subcarriers, a requirement in network communications.

A microsequencer is used to control the entire architecture. The microsequencer controls the crossbar switches and the memory structure, selects which matrix is used in the crossbar switch for each operand, selects the function unit output to retrieve from and route to the memory structure, and controls the individual function units. Two advantages of using a microsequencer instead of a microprocessor are area and speed efficiencies, and the ability to operate all design components in parallel. A disadvantage is that the instruction words are longer and require more chip resources for implementation.

In Figure 1.1 the overall design layout is shown. The entire design is controlled by microsequencer instructions, which the user provides during FPGA configuration. The central block memory 8×8 matrix has a number of subcarrier planes, as is required by the application. Two 4×1 multiplexers are located at the output of the block memories. Each of these multiplexers selects a block memory 4×4 matrix, which is the source of each of the two operands required by all of the function units. Each operand matrix is sent through a sorting circuit to retrieve the necessary input data. The function units then perform the necessary calculations, and the outputs are sent to a multiplexer to select a single function unit output and route it to the appropriate block memories within the matrix.

Before attempting to model the design in Verilog, the hardware design was simulated in software using C. By doing this, the overall design model was verified at a high level for correct operation. Two primary test areas were the microsequencer and the crossbar switch routing structures. Hardware implementation began upon completion of the software simulation. The software model was redesigned in Verilog for use in the programming of the FPGA chip. Xilinx tools were used to implement and test the Verilog code on the FPGA.

Due to the nature of this design, a custom register transfer language was designed to help the user implement the desired algorithms efficiently. The language is fairly straightforward, and the instructions mostly parallel their function unit. An assembler must also be created to convert the register transfer language into machine code. In this case, the register transfer language is converted into the memory used by the microsequencer. The assembler profiles the code written by the user, and determines the set of modes each crossbar switch uses

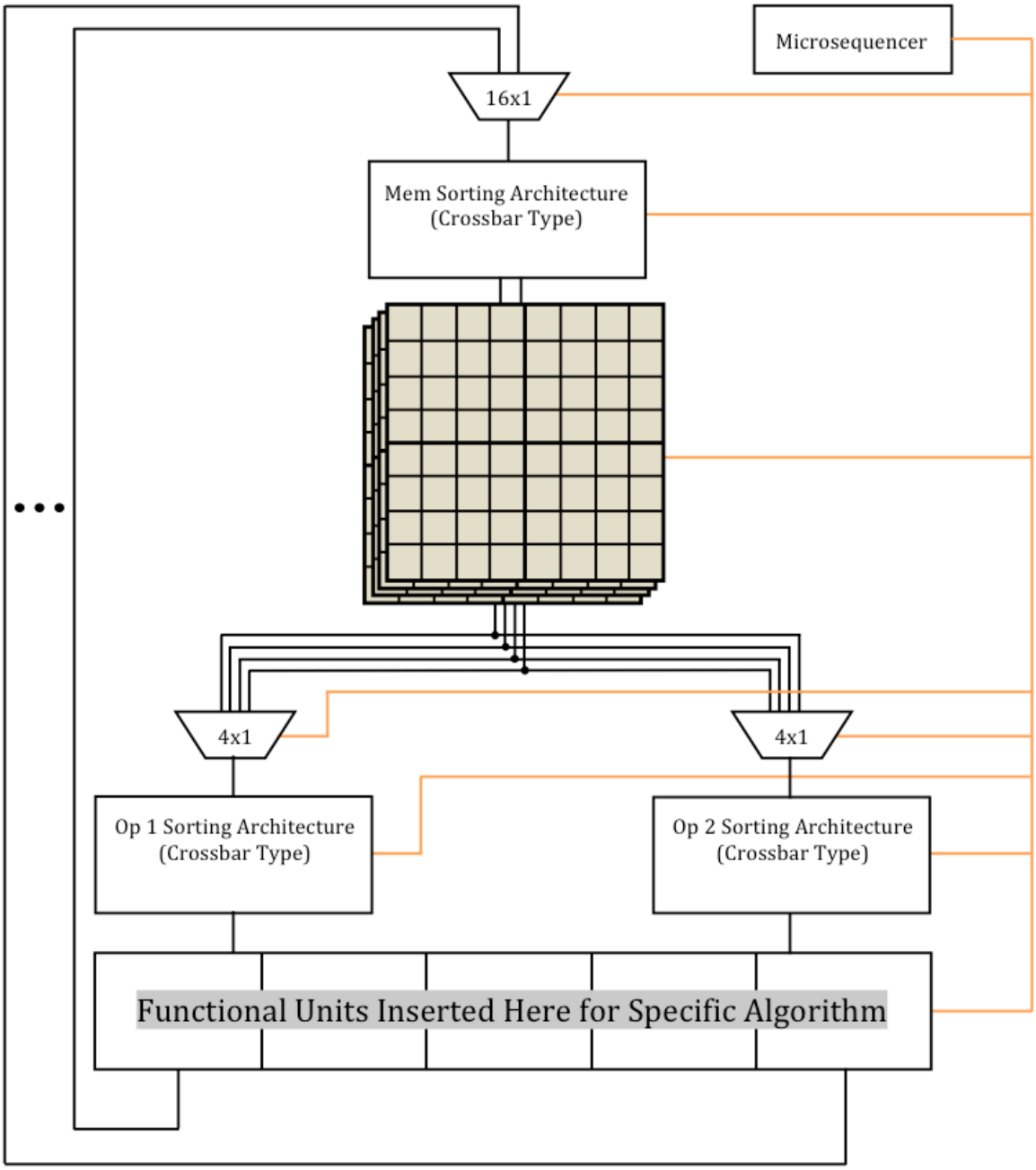

Figure 1.1: System Overview

during execution. Once a list of modes is constructed, the assembler can begin converting instructions into binary strings loaded into the microsequencer. The assembler then takes the original Verilog code and modifies it to reflect the user-supplied algorithm.

The system runs at a clock speed of 183 MHz; the microsequencer is the major bottleneck. This design is capable of performing a matrix inversion within 335 clock cycles, or 1,829 ns. The power efficiency of the design is measured at 17.15 MFLOPS/W. Two algorithms running on a MicroBlaze processor are used as baselines for performance comparison. As an average, the hardware design performs 4.2 times faster and is 1.9 times more efficient in regards to floating point operations per second per Watt.

1.3 Organization

In an effort to completely describe the research done for this thesis, six chapters are provided. The first chapter is an introduction to the overall problem and the solution presented. The second chapter provides background information that the reader will need in order to better understand the information in this thesis. The third chapter is an overview of the overall system that has been implemented on the FPGA and how the custom hardware design language is synthesized into an FPGA hardware component. The fourth chapter describes the entire design and another design used for comparison. The fifth chapter provides resource usage data for the implemented system, the design used for comparison, and compares the two. The sixth chapter summarizes the work done and discusses future work.

Chapter 2

Background

For the work done in this thesis there are a few underlying topics that are very important to understand. These topics include the MIMO algorithm, communication synthesis, FPGAs, and the FPGA synthesis tools. In this thesis, the MIMO algorithm is selected for testing purposes. Communication synthesis is the process integrating all of the hardware components together. Lastly, FPGAs are the targeted hardware for testing, and the synthesis tools implement the design on the FPGA.

2.1 MIMO Algorithm

MIMO supports multiple antennas on both the transmitting and receiving sides of the communication link. Because it has the ability to split the main signal into many signals, Orthogonal Frequency Division Multiplexing (OFDM) is used in conjunction with the MIMO

algorithm. The many signals created are referred to as subchannels, and the memory modules used to store the matrices are as deep as the number of subchannels used in the algorithm [25]. An example of a wireless communication technology based on MIMO is 802.11n [27].

Figures 2.1 and 2.2 show models of the OFDM transmitter and receiver. For the transmitter side, the main signal is divided into subchannels that are input to the inverse Fast Fourier Transform (FFT) module. The output of the FFT has two parts, real and imaginary, which go through digital-to-analog converters (DACs). The output analog signals are then modulated with waves at the carrier frequency: cosine for the real component and sine with a 90° phase shift for the imaginary component. The two signals are then summed together and transmitted. The receiver is similar to the transmitter, but in reverse order and with a few adjustments. As in the transmitting stage, the signal is first received and then modulated again. Signals are then sent through low-pass filters and the analog-to-digital converters (ADCs). The signals are then sent through an FFT module and finally converted back to an individual signal.

Spatial multiplexing, a method of transmission commonly used in MIMO technologies, allows for a single signal to be broken down into many subchannels and then transmitted across multiple antennas [28, 30]. When using spatial multiplexing, Vertical-Bell Laboratories Layered Space-Time (V-BLAST) is a common detection algorithm required by the MIMO receiver [26]. V-BLAST works by performing multiple calculations on a signal, where with each calculation the most powerful signal is removed until all of the original signals from the MIMO transmitter have been separated.

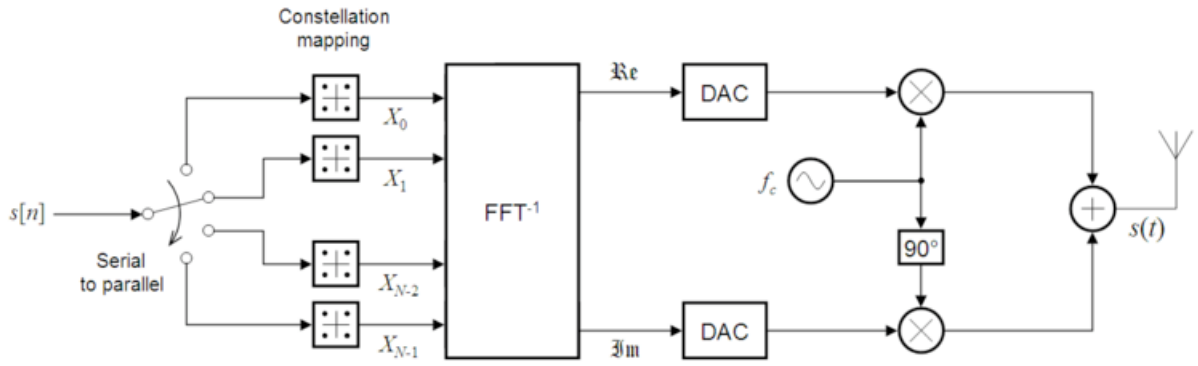

Figure 2.1: OFDM Transmitter Model [29]. Used under fair use, 2011.

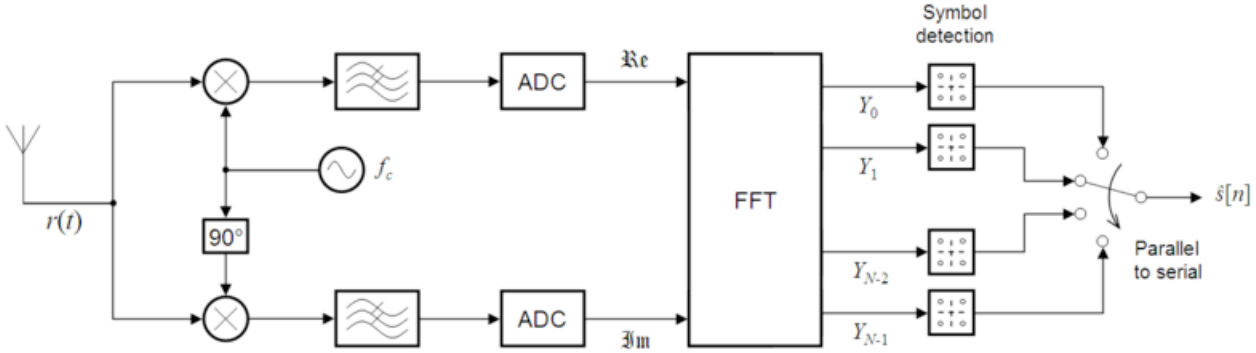

Figure 2.2: OFDM Receiver Model [29]. Used under fair use, 2011.

2.2 Communication Synthesis

Alberto L. Sangiovanni-Vincentelli defines communication synthesis as "a reuse methodology [which] starts with a set of functional requirements and constraints on the interaction among components and then proceeds to build protocols, topology, and physical implementations that satisfy requirements and constraints while optimizing appropriate measures of efficiency of the implementation" [19]. This concept is the focus of this thesis. A major objective of this research is to establish guidelines for how device interfaces are designed. Through the use of these guidelines one can establish a common protocol for communicating to all the

attached devices. This method is particularly useful for intellectual property where the end user does not have access to the design within the "black box", and permits designs to be automatically constructed with less difficulty.

One key component of communication synthesis is automatically generating the software, hardware, and routing between the hardware components in the system. In this method, cost metrics are used to evaluate the best possible placement and routing paths. Each placement requires a different set of optimized routing paths to minimize the amount of area used and to minimize wire length between two points which will lead to increased performance. Generally, part of the job of communication synthesis tools is to strategically place the components, decide on the best type of interconnect hardware for the components, and then to route between the components in the most efficient manner. In [8] and [18], communication synthesis is discussed, more specifically the generation of the interconnect between components.

The configuration of FPGA hardware requires the use of the Xilinx Integrated Synthesis Environment (ISE), which includes the Embedded Development Kit (EDK). The EDK program contains two tools, the Xilinx Platform Studio (XPS) and the Software Development Kit (SDK). Within the EDK are also IP cores that are required for using the embedded PowerPC hard processor and the MicroBlaze soft processor. XPS configures the hardware setup on the device; while the SDK configures the software run on the PowerPC or MicroBlaze processors. The two tools, XPS and SDK, are essential to communication synthesis. XPS, specifically, is similar to the solution needed for the problem presented in this thesis. One

major drawback is that the XPS software requires an embedded processor, a complex system which requires additional resource usage and adds overhead that may result in a slower system. Additionally, XPS interconnect hardware is limited and likely to be inefficient for the problem presented here.

Sonics [11] is a leading company in Network on Chip (NoC) technology. The term NoC denotes the intelligent network communications used within a System on Chip (SoC), which in this case consists of multiple IP blocks. SonicsLX, a software produced by Sonics, is used to connect IP blocks. The program uses multi-threaded, non-blocking, crossbar switches so high bandwidth and power efficient routing between IP blocks is possible. An overarching design environment, SonicsStudio, integrates the entire process of instantiating IP blocks, interconnecting the blocks, system analysis, and system synthesis. The system does not, however, supply users with a control unit for the overall design; thus, users must design their own or find suitable IP blocks. Furthermore, SonicsStudio uses standard memory structures, that are not optimized for the problem addrssed in this thesis.

Coral, software developed by IBM in 2000, is considered to be one of the first tools to integrate IP cores [3]. Coral allows the user to design the system at a high level of abstraction and to place IP cores where they are needed without making any connections. Eventually, the user will invoke algorithms that automate the process of connecting the signals of the IP cores throughout the system. An advantage of Coral is that it connects the IP cores into the overall system, however, Coral does not offer any predefined system controllers or memory storage options. The lack of these elements places the burden on the user to get the system

up and running or to purchase third-party IP cores.

Qsys is the system integration tool, developed by Altera, which is the successor to the recent SOPC Builder [1]. This tool is more focused towards an FPGA NoC, when compared to its predecessor, and automatically generates optimized interconnect networks for the IP cores used in the design. Qsys, unlike some other communication synthesis tools, has packaged with it processor options, mainly the Nios II processor, and basic memory options. While the advanced routing techniques are very useful, this software does not incorporate the advanced crossbar switch implemented in this work, which allows for many different datapaths to be configured during a program execution, or the matrix memory structure that exploits parallelism in algorithms.

In the past few years a standard, IEEE 1685-2009, has been released [10]. This standard is molded around IP-XACT, which was developed by the SPIRIT consortium in an attempt to set standards to ease the difficulty of implementing IP cores in designs. IP-XACT covers the standard questions of IP integration which are mainly what are the buses used, which ports need to be available, and how the two are interconnected. IP-XACT also goes a step further by using eXtensible Markup Language (XML) documents, which would define all the necessary information for a piece of IP to be delivered. With the XML document, the user would know how to connect the IP core and could also provide this information to an automatic routing tool.

Another attempt to standardize IP cores has been taken on by Open Core Protocol International Partnership (OCP-IP) [2]. The vision of OCP-IP is to create universal standards for

IP interfaces, so that IP can be dropped into a design and require minimal effort from the user to connect the interface to the buses already in place. The efforts of OCP-IP have been strengthened with Sonics components requiring IP cores to meet the open core protocol. One example of this is the Sonics OCP Library for Verification (SOLV) [12].

2.3 FPGAs and Synthesis Tools

FPGAs are programmable devices which can provide functionality similar to custom integrated circuits. FPGAs provide fast design to market times at the expense of slower speeds, higher chip area, higher unit costs, and higher power consumption. Figure 2.3 shows a generalized diagram of a typical FPGA. The three main components are the logic blocks, interconnection resources, and the I/O cells. The logic blocks modify the data, the interconnection resources move data around the FPGA, and the I/O cells get the data in and out of the FPGA.

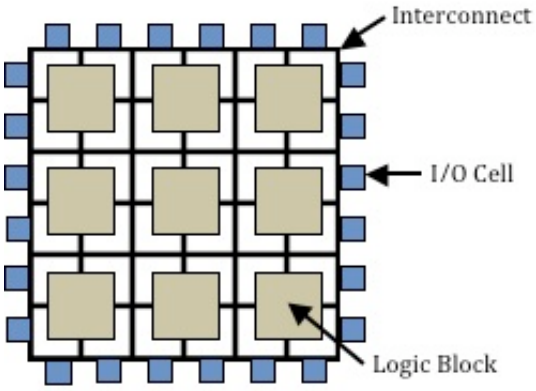

Figure 2.3: Generic FPGA Architecture

The Configurable Logic Block (CLB) is the major component within a Xilinx FPGA that allows the user to implement basic digital logic circuitry needed during the typical design process; Figure 2.4 shows the basic input/output signals of this component. Within each CLB are slices and the circuit structures required to connect those slices together, and to other CLBs within the FPGA. Within each slice are Look-Up Tables (LUTs), four storage elements, multiplexers, and carry logic. The Xilinx XC6VLX240T FPGA has an array of CLBs containing with a total of 150,720 6-input LUTss and 37,680 slices. The FPGA resource information included in this thesis is obtained from a combination of Xilinx user guides [35, 39, 40, 36, 37].

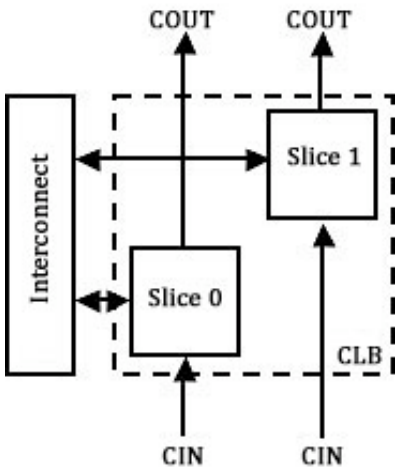

Figure 2.4: Configurable Logic Block

Current Xilinx FPGAs also contain BlockRAM and DSP slices; Figures 2.5 and 2.6 show the basic input/output signals of these components. The Xilinx XC6VLX240T, has 416 BlockRAMs and 768 DSP slices. Each BlockRAM contains two independent 18 Kb RAMs, combining for a total of 36 Kb per BlockRAM and 14,976 Kb on the FPGA.

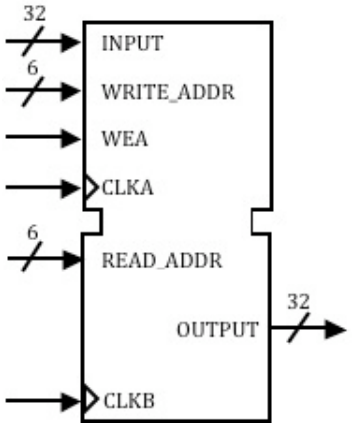

Figure 2.5: Virtex-6 BlockRAM

Xilinx provides a few basic tools that have been used to develop the design in this thesis. These tools are mainly the ISE Project Navigator, the CORE Generator, and iSim. These tools, unlike the EDK, are less focused on embedded systems and enable implementing circuits on an FPGA captured with a Hardware Design Language (HDL), such as Verilog. The ISE Project Navigator provides the user with a centralized interface to access the other tools needed in a standard design flow, a place to edit all the HDL files in the project, and a way to configure the hardware with the HDL. The CORE Generator is a tool used to create hardware using IP provided by Xilinx; an example of its usage in this work is the creation of

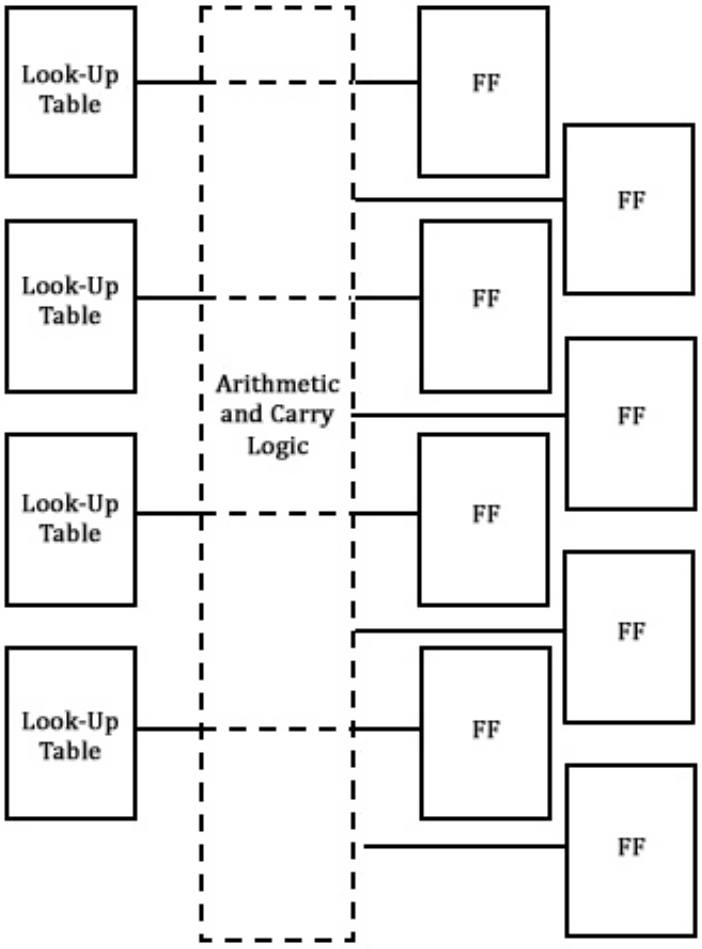

Figure 2.6: Virtex-6 SLICEM

the floating point function unit, which can perform operations on floating point data such as addition, subtraction, multiplication, division, and many others. iSim, a simulator designed by Xilinx, takes the hardware settings from ISE Project Navigator, the HDL files being used in the project, and the simulation models for the IP instantiated using CORE Generator and uses them to create a simulation model for the entire design, which allows the user to verify hardware behavior before synthesis and FPGA implementation.

Chapter 3

System Overview

This research was inspired by a MIMO decoder accelerator architecture developed by Karim Mohammed and Babak Daneshrad at the University of California [16]. Their design is specifically intended for use with the algorithms encountered with MIMO wireless communications. The overarching focus of their design is matrix inversion, performed using a provided set of function units.

In order to do matrix inversion, a Singular Valued Decomposition (SVD) is first performed on the matrix. After decomposition, the matrix is easier to invert. The SVD uses a systolic array of CORDIC processors. It is important to note that the matrix values are fixed point, meaning that their design would in a real world application need to convert from floating point to fixed point and vice versa.

In the final design of the MIMO decoder accelerator there exist four main function units:

a CORDIC rotation unit, complex adder, complex multiplier, and complex divider. These function units were chosen to allow implementation of almost any algorithm desired. A similar, primitive set of function units will be provided by the end user in the design constructed in this work.

One of the major innovations in [16], is memory access. A set of memory blocks are combined to resemble a matrix. By doing this they are able to create a memory block for each element of four 4x4 matrices, each having a depth equivalent to the number of subcarriers required by the algorithm. This method allows matrix elements to be accessed in parallel rather than needing to wait multiple cycles to get all of the data from a single memory block.

A major obstacle encountered was routing data to and from this large memory structure. Two operands at a time must be fetched from the memory structure to the function units, while also sending the output from the function units back into the the memory structure. In order to do this, sorting structures were built from a large number of multiplexers. In terms of resources used and number of bits required in the instructions, this type of design creates a very large overhead in the final implementation.

The CORDIC rotation unit used in the MIMO decoder accelerator is capable of performing various unitary transformations. There are four operations that combine within this unit to calculate hyperbolic and trigonometric functions: addition, bitshift, subtraction, and lookup tables. When in rotation mode, the CORDIC unit can be used to calculate sine and cosine for a fixed point format angle given in radians. The CORDIC rotation unit is useful because it avoids the need for complex multipliers. This method is discussed in great detail by

Hemkumar in his thesis [9].

3.1 Reconfigurable Interconnect

In [6], two types of routing are discussed: deterministic and adaptive. Deterministic routing uses a fixed path, while adaptive routing permits different paths. In this thesis, the memory routing structures are adaptive; rather than the data always going to the same locations, data is sent wherever each cycle necessitates.

The most degenerate form of a routing network is a bus, which is considered unicast. Unicast is a term which refers to a data path which takes in, from a source, a single piece of data at a time and then can route that piece of data to a single output of the users choice. The alternative to this, which is the type needed in this design, is multicast, where multiple inputs are being retrieved at once and then being sent to multiple outputs.

The design used is a single stage crossbar switch, which is not to be confused with a staged network. An example of a staged network is a Clos network, which was created by Charles Clos in 1953 for telephone networks [32]. The premise of the 'Clos network is that a crossbar switch is needed, but no crossbar switch exists that is large enough for the job. To compensate, multiple stages of crossbar switches are used, which will in the end provide the paths necessary to get an input to whatever output is chosen. The single stage crossbar switch is ideal for this research due to the small number of paths required, with the added advantage of less overhead and fewer control bits required.

An example of a state-of-the-art crossbar switch is the 288×288 crossbar switch developed by Mindspeed [15]. The crossbar switch is designed for high-speed data switching. The device is asynchronous and non-blocking, meaning it is not dependent on the system clock and no buffers are present so that inputs propagate to the outputs over a single clock cycle. The device is capable of transmission speeds as high as 3.2 Gbps.

Two different types of switching are packet and circuit switching. The crossbar switch design in this thesis utilizes circuit switching because it suits the partial re-configurability required. In a crossbar switch, each input to the circuit can be effectively routed to each output of the circuit via a switching matrix.

One method of designing a crossbar switch is with multiplexers; however, this has high resource usage on the FPGA. An alternate but complex method for creating a crossbar switch uses Programmable Interconnect Points (PIPs) as cross-points on a Xilinx FPGA [41]. The final design was able to achieve clock speeds of 155.5 MHz and could reconfigure the crossbar switch in 220μs; however, the design is not implementable on current FPGA devices.

Ultimately, packet switching, crossbar switching using multiplexers, and crossbar switching using PIPs are insufficient methods of routing. Fischer [6] looked into one final implementation of a crossbar switch that would meet all of his necessary requirements. Unfortunately, the modular design flow, which is crucial in the implementation of the hard macro that he has designed, is not available in the current version of the Xilinx ISE design tools, making the design unusable.

Given the limits of the predefined architectures with an FPGA and limited access to the low level architecture when using Xilinx ISE design tools, there is no efficient method for implementation on an FPGA, comparable to a VLSI implementation of a crossbar switch. Research conducted up to this point to try and overcome these hurdles is rendered ineffective by more recent versions of Xilinx hardware and software.

The design in this thesis implements a crossbar similar to the crossbar constructed in [4]. In this method, 2-input AND gates, which are wired with the input line and control signal, are used as the switches, and all of the output signals of the AND gates are sent to an OR gate that delivers the final routed signal. Compared to other designs used for crossbar switches, the design in [4] requires less area and had improved signal delay. It is important to note that this design is synchronous, meaning that the inputs are only captured when a clock event is encountered, and non-blocking.

3.2 Control Structure

One requirement is a control unit, of which there are three possibilites: a finite state machine, a microsequencer, or a microprocessor. Of the three options, a microsequencer is the most appropriate choice; it provides more power and flexibility than a standard finite state machine since it is programmable, while also requiring less resources than a microprocessor. There is an extensive amount of information regarding microsequencers and the microprograms used to run them in Chapter 5 of [23]. Four reasons make a microsequencer more desirable

than a finite state machine: scalability, the ability to easily implement control hierarchies, exception handling is made easier, and flexible programmability. All these attributes are taken advantage of in this thesis.

The microprograms, which run on a microsequencer, are composed of a set of microinstructions. These microinstructions are a collection of bits used to control all of the various parts of the system during a single clock cycle. Depending on the number of elements in the system that need to be controlled, a large number of bits may be required in the microinstruction. There are two basic types of microinstructions: horizontal and vertical. A horizontal microinstruction is straightforward, and each bit is mapped directly to what it controls. A vertical microinstruction requires an intermediate circuit to decode the bits and route the decoded bits to hardware they control. Both horizontal and vertical microinstructions are used in this work.

There are two main parts to a microinstruction: the command field and jump field. The command field is used to control logic along the datapath, and the jump field indicates which microinstruction to execute next. The jump field itself can be broken into two parts: the next address logic bits and the address bits. The next address logic bits can be set in a manner such that the address bit field is jumped to, or incremented upon, or any other transformation needed to navigate through the microinstructions. It is also worthwhile mentioning that one can wire interrupts into the next address logic of the microsequencer, which would allow for specialized actions to take place as appropriate for the interrupt.

Single Instruction Multiple Data (SIMD) is a term used to describe a system that is capable

of performing the same operation on multiple pieces of data simultaneously [34]. The Intel Multimedia Instructions (MMX), developed in 1996, are an example of an instruction set based on the SIMD technique [33, 24, 13]. MMX adds support for parallelizing calculations in multimedia applications by introducing new instructions, data types, and 64-bit registers. The operation of MMX instructions is slightly different from the concepts utilized in this thesis in that only certain processes can be parallelized with the MMX instructions. The rest of the time only one calculation at a time is performed, but in the design here data operations are always parallelized.

3.3 Memory Structure

The Xilinx FPGA BlockRAM memory modules are used as the memory structure storage elements and they have the ability to read and/or write in one cycle. The BlockRAMs have two modes of operation: single-port and dual-port. The two modes differ in that a dual-port BlockRAM has the ability to read two items, write two items, or to simultaneously read and write an item from the memory array during a single clock cycle. A single-port BlockRAM is limited to a single memory array access over a single clock cycle.

There exist three different write modes if there is a simultaneous read and write to the same memory cell; WRITE_FIRST, READ_FIRST, and NO_CHANGE. WRITE_FIRST mode sets the output to the newest written data, READ_FIRST mode sets the output to the previous written data, and NO_CHANGE keeps the output from the last read operation.

The BlockRAM's default write mode is WRITE_FIRST. In order to reliably perform a read operation while a write operation is in progress, the default write mode is incorrect and the READ_FIRST option should be selected instead.

Beyond choosing the number of ports and the write mode of the BlockRAM, there is one final design decision: how to clock the dual-port BlockRAM. The two clocking modes are asynchronous and synchronous clocking. In asynchronous timing, two read operations can occur whenever needed; however, during a write operation the second port cannot read/write to the memory being written to. In synchronous timing, read and write operations act as in asynchronous timing, but the write mode, as mentioned above, must be configured per the design requirements.

In modern processors, memory interleaving is often used [14]. This method is used because processors and buses today are capable of performing at speeds which are much faster than those of the memory. The technique involves instantiating multiple blocks of memory, and storing consecutive memory addresses across them, thereby allowing multiple memory addresses to be accessed simultaneously and returned one at a time with each successive cycle. The actions described here, though they may sound familiar, are different from the design in this thesis where up to 16 memory elements can be returned at the same time rather than needing to wait 16 cycles.

Register files are another part of modern microprocessor architecture, which resemble the memory structure being implemented here [31]. Register files differ in that they are primarily used as an intermediate step in the process of getting data to and from the main memory

and are located within the CPU itself. Register files are implemented in SRAM or flip-flops, which is a type of memory that is very fast and expensive, leading to a desire to have a small number of them available. The memory structure implemented in this thesis is directly connected to the system, similar to register files. The structure does not cache data for another set of memory, and is fast enough that the microprocessor can cheaply use it without bottlenecks while still providing the microprocessor with all of the workspace it would possibly need.

3.4 Assembler

An assembler is software that reads a file written in a pre-defined syntax and outputs the appropriate machine code [22]. For an assembler there must be two inputs: a command line, which executes the assembler and specifies the source file, and the source file itself. The output from the assembler is the object file containing the machine code. In this work, the object file is a Verilog file with the microsequencer design, and a read-only memory called CSTORE, which stores the microprogram.

One-pass and a two-pass assemblers are used, the latter being more prevalent. These two assembler types differ in how unresolved references within the source file are handled. In a two-pass assembler, the first pass looks for all of the labels in the source file and creates a symbol table containing those labels and their values, which are assigned by the location counter. The second pass translates the instructions into machine code using the values now

in the symbol table. A one-pass assembler operates in a different and more complex manner. Because of the added complexity of the one-pass assembler, a two-pass assembler, is used in this thesis.

Five methods are suggested for the symbol table data structure: a linear array, a sorted array with binary search, buckets with linked lists, a binary search tree, and a hash table. The method used here is a linear array; it provides fast insertion and simple operations, but searching is less efficient. For the purposes of this thesis, the microprograms will not likely be very large, and as such search time will not play a significant factor.

In general, an assembler needs to take into account whether or not the machine code will be placed starting at memory location zero. Depending on the choice made by the user in regards to the starting position of the program, there are two types of object files that can be used: absolute and relocatable. An absolute object file, which is the kind generally created by a one-pass assembler, will be incorrect if the the machine code is not located at position zero. A relocatable object file is designed so that if the machine code is not at position zero, there are bits in the file which indicate the machine code that must be relocated. Due to the nature of the design in this work, the code will always start at position zero in the control store.

The assembler program, illustrated in Figure 3.1, has two major phases during execution. In the first phase, the program reads instructions from the input file and stores them in the appropriate data structures. These data structures are read-only memories containing either the microprogram or configuration bits for the crossbar switches. In the second phase, the

program takes the data structures from the first phase and a pre-designed microsequencer, and creates the *Microsequencer.v* Verilog file, which is the source code for the main controller of this design.

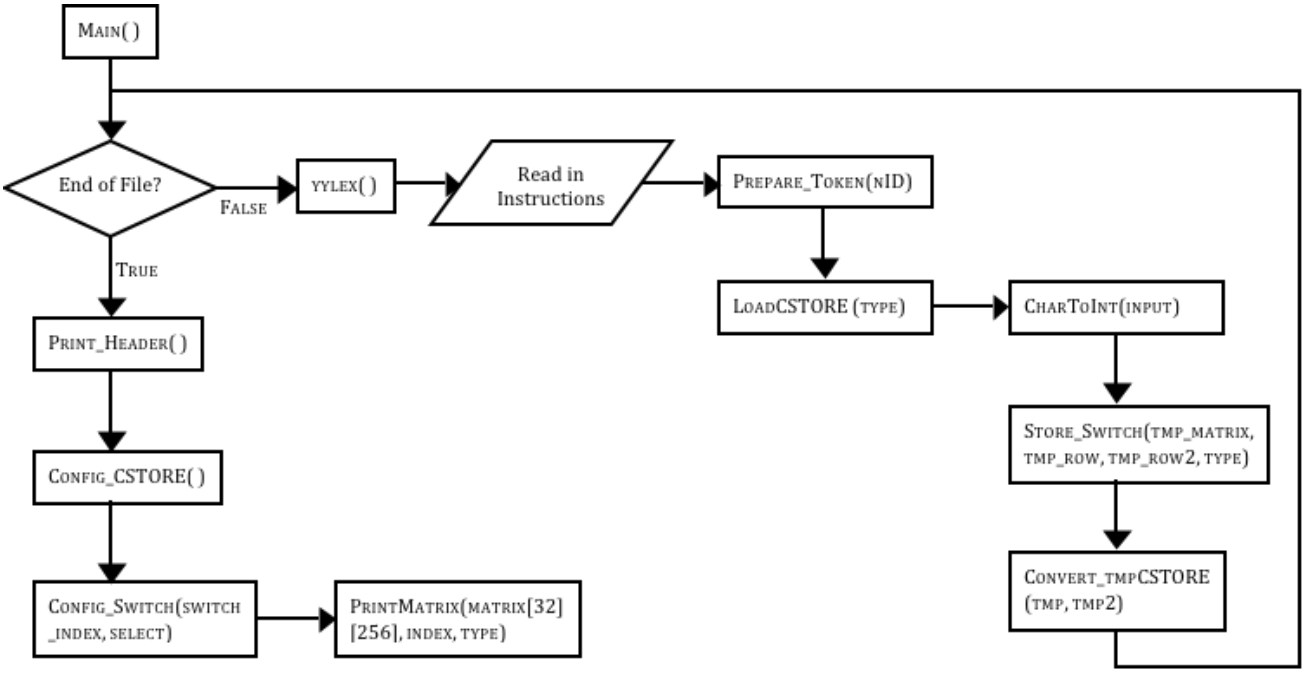

Figure 3.1: Assembler Flow Chart

The predominant function used in the first phase is `yylex()` [7, 21]. The function reads and analyzes each line of the input file and compares it against a set of rules that have been defined to reflect the format of the instructions to be used. Once a match is found between a line of the input file and a rule in the analyzer, tokens are created and the function `LoadCSTORE()` is invoked. `LoadCSTORE()` parses the line from the file and converts appropriate characters to numerical values, converts the instruction to binary microcode, computes the crossbar

switch settings, and stores these settings in a data structure later used to create the Verilog code needed to create an FPGA ROM specifying the appropriate routing.

In the second phase of the assembler, the first task is printing out header information to the Verilog file being created. The header information is static and independent of the program being passed as input. Next, the function `Config_CSTORE()` is executed, which takes the stored binary microcode from the first phase and translates it to the Verilog code necessary to represent a ROM. This ROM is where the microsequencer fetches code to execute. Afterwards, the function `Config_Switch()` is executed, which takes the routing information stored in data structures in the first phase, and translates it into a ROM. During this step the assembler displays a visual representation of the routing in the crossbar switch on the terminal for the users' convenience.

The overall flow of the software is illustrated in Figure 3.2. Under normal circumstances, all the user has to do to generate `Microsequencer.v` is to navigate to a linux command prompt and enter the command: `./Assembler <` ***`Input File`***, where input file is the microprogram written by the user. The format of the input file is detailed in Table 3.1. If, however, changes are made by the user to the assembler for more customized usage, there are two additional commands that must be issued prior to the previous command. The first command is: `flex` ***`scanner.lex`***, where `scanner.lex` is the framework of assembler that has been designed to create `Microsequencer.v`; this command converts the flex file to a C program. The second command is: `gcc` ***`lex.yy.c`*** `-lfl -o Assembler`, where `lex.yy.c` is the C program generated by the first command and `-lfl` tells `gcc` to link the FLEX library;

this command creates the executable.

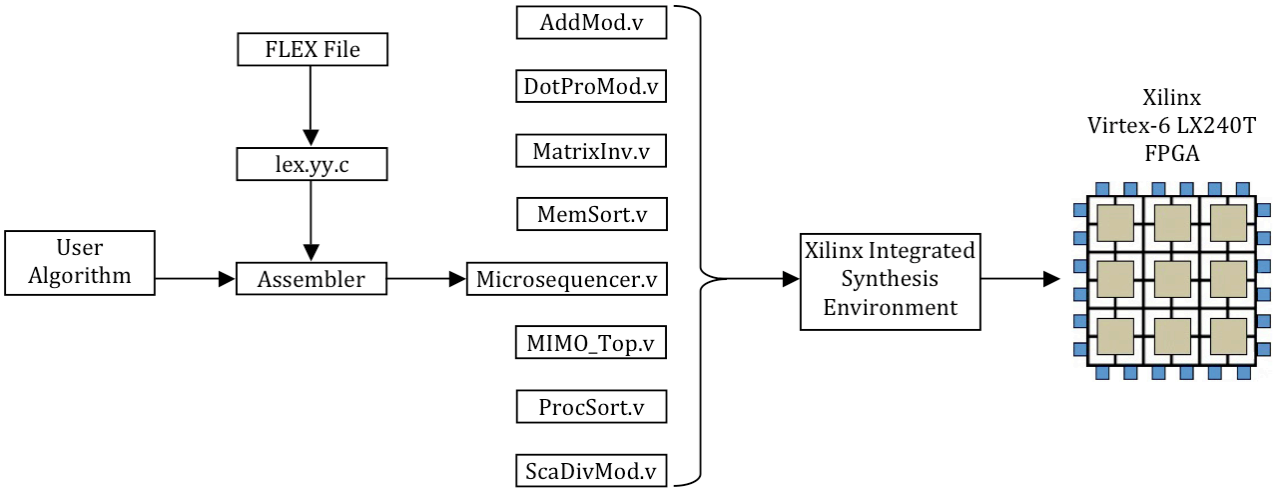

Figure 3.2: Flow of Software Development

Table 3.1: Register Transfer Language Instructions

Instruction	Definitions
load	
nop, Repeat #	a nop with a repeat value
Matrix(R{,R}) = Matrix() | Func #	direct map of a matrix to a function unit
Matrix(R{,R}) = Matrix(R/C/T) | Func #	maps an element of a matrix to a function unit
Matrix(R{,R}) = Matrix(R/C/T), Matrix(R/C/T) | Func #	maps elements of two matrices to a function unit

Matrix: A, B, C, D
Repeat #: 0 - 1023
Func #: Represents the position of function unit in the design left to right (0-15)
R: Row # 0, 1, 2, 3
C: Column # a, b, c, d
T: Transpose
{ }: denotes optional input

Chapter 4

Implementation

An overall system has been implemented that is capable of handling the majority of the operations necessary to perform MIMO decoding. In the MIMO decoding process, matrix inversion is the fundamental operation. As a consequence, matrix inversion is the primary focus of the verification and testing in this work and is the primary basis of comparison. Both systems are modeled on the Xilinx Virtex-6 LX240T FPGA, ensuring that they both have the same set of constraints, which leads to a more accurate comparison. The design being compared against is an FPGA configured with a Xilinx MicroBlaze soft-processor running a basic version of an LU decomposition algorithm written in C.

4.1 Small Matrix Memory Structure

The design in this thesis requires the use of a matrix-based memory structure, containing four 4×4 matrices, as shown in Figure 4.1. Each of the sixty-four elements used is a dual-port BlockRAM, capable of performing a read and a write operation within a single clock cycle. In order to support a 32-bit floating point data format, the data width of each BlockRAM is 32 bits. The number of addresses within each BlockRAM is equivalent to the number of subcarriers needed for the algorithm being implemented; the default number of subcarriers used is 64.

For each instruction executed in a given clock cycle, a maximum of 16 inputs may be required for each of the two operands. There is a possibility of needing up to 32 inputs to a function unit. Also, there is a maximum of 8 possible outputs to the memory structure from the function units. Due to these design constraints, it is essential to implement a memory structure more efficient than a standard design using a single BlockRAM.

If a single BlockRAM was used, a maximum of 32 clock cycles would be needed to retrieve all operands data, an additional 8 clock cycles would be needed to write back the data to memory; this would be a total of 40 clock cycles. By implementing a dual-port BlockRAM, it is possible to use at most 32 clock cycles because data can be written to the BlockRAM while data is also being read. Implementing a dual-port BlockRAM for each of the 64 matrix elements makes it possible to do all necessary reads and writes to memory in a single clock cycle. The exception to this is a data hazard where a read is required and the data to be

retrieved has not finished writing to memory.

In order to load data into the memory structure, the command `load` has been implemented in which data from outside the device can be retrieved and stored within matrix A (memory blocks 1-16). Implementing a command to do this was essential for populating the memory structure with usable data. This would occur with MIMO when the wireless receiver is acquiring new data from a wireless device, for example. It is worth noting that write enables across the row of a matrix are tied together; hence the microsequencer only needs to have 16 write enable bits in the microinstruction instead of 256. Algorithm 1 shows pseudocode for the memory module that reflects the design mentioned above.

```
1.1  reg [# of bits] ram [# of subcarriers];
1.2  always @(posedge clock) begin
1.3  |  if write_enable = 1 then
1.4  |  |  ram[write address] = input;
1.5  |  end
1.6  end
1.7  always @(posedge clock) begin
1.8  |  tmp_out = ram[read address];
1.9  end
1.10 assign out = tmp_out;
```

Algorithm 1: Memory Module Pseudocode

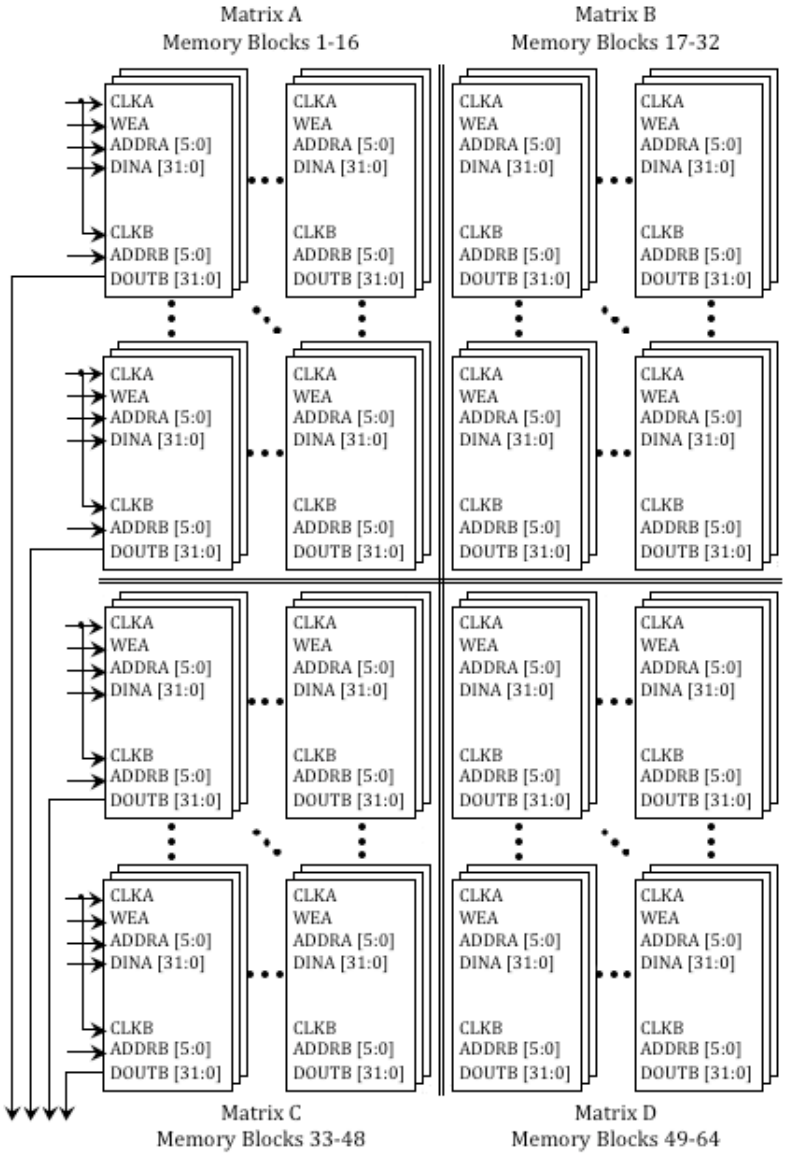

Figure 4.1: Memory Structure

4.2 Crossbar Switch and System Interconnect

The design of the system interconnect is very important, since this could pose a serious bottleneck for the overall system. The two main areas of focus concerning the system interconnect are: the path from the output of the memory blocks to the function unit inputs, and from the function unit outputs to the input of the memory blocks. Within these two areas there exist multiple crossbar switch modules, which allow routing any inputs signal to any output signal.

For the first area, where the memory block outputs are routed to function unit inputs, there are actually two identical interconnects: one for the first operand and one for the second. Each of the interconnects here consists of a 4×1 multiplexer and a 16×16 crossbar switch. The 4×1 multiplexer is designed such that it allows the interconnect to select an entire 4×4 matrix (16 inputs) at once. This implies there are 512 4×1 multiplexers controlled by the same select bits. The 16×16 crossbar switch has 16 inputs and 16 outputs which creates a total of 256 crosspoints. Each crosspoint consists of a 2×1 multiplexer. When the enable bit is 1 the 32-bit floating point value is allowed to pass through, and if the enable bit is 0 then a 0 is output instead. The 2×1 multiplexer actually consists of 32 2×1 multiplexers to account for each individual bit of the data. At each of the 16 outputs of the crossbar switch there are 32 16-input OR gates. Figure 4.2 gives an illustration of the layout of the crossbar switch.

A single interconnect is used, where the function unit outputs are routed to the memory

block inputs. This interconnect consists of a 16×1 multiplexer and an 8×32 crossbar switch. The 16×1 multiplexer is designed such that it can pick one of the function units to route to memory, and there are 16 inputs because a maximum of 16 function units can be inserted into the system. Because any given function unit has a maximum of 8 outputs, there are only 8 inputs to this crossbar switch. Also, there are 32 outputs from the crossbar switch because the memory blocks have a mirror bit that allows the user to either write to *A/B* or *C/D*. The inner-workings of this crossbar switch are the same as with the first except it has been modified to work with 8 inputs and 32 outputs.

Initially, only AND and OR gates were expected to be used. During implementation it was realized that writing Verilog code to reflect this structure was very inefficient. As a result, conditional operators, which create multiplexers rather than AND gates, were used in an effort to simplify the design process. Algorithms 2 and 4 demonstrate this methodology, while Algorithms 3 and 5 show the top level of the routing circuit and how inputs/outputs are connected to the crossbars. The end result is a design that is easy to capture in Verilog.

```
2.1  assign o1 = (configure bit 0 = 1) ? func unit input 1 : 0;
2.2  assign o2 = (configure bit 1 = 1) ? func unit input 2 : 0;
2.3  assign o3 = (configure bit 2 = 1) ? func unit input 3 : 0;
2.4  assign o4 = (configure bit 3 = 1) ? func unit input 4 : 0;
2.5  assign o5 = (configure bit 4 = 1) ? func unit input 5 : 0;
2.6  assign o6 = (configure bit 5 = 1) ? func unit input 6 : 0;
2.7  assign o7 = (configure bit 6 = 1) ? func unit input 7 : 0;
2.8  assign o8 = (configure bit 7 = 1) ? func unit input 8 : 0;
2.9  repeat o9 → o256 reusing func unit input 1 → 8
2.10 assign output 1 = o1 through o8 as inputs to OR gate
2.11 repeat outputs 2 → 32
```

Algorithm 2: Memory Crossbar Pseudocode

```
3.1  always @(posedge clock) begin
3.2      case (muxSel)
3.3      0: begin
3.4          bus1 = func unit 0 bit 0;
3.5          bus2 = func unit 0 bit 1;
3.6          bus3 = func unit 0 bit 2;
3.7          bus4 = func unit 0 bit 3;
3.8          bus5 = func unit 0 bit 4;
3.9          bus6 = func unit 0 bit 5;
3.10         bus7 = func unit 0 bit 6;
3.11         bus8 = func unit 0 bit 7;
3.12     end
3.13     ...
3.14     15: begin
3.15         bus1 = func unit 15 bit 0;
3.16         bus2 = func unit 15 bit 1;
3.17         bus3 = func unit 15 bit 2;
3.18         bus4 = func unit 15 bit 3;
3.19         bus5 = func unit 15 bit 4;
3.20         bus6 = func unit 15 bit 5;
3.21         bus7 = func unit 15 bit 6;
3.22         bus8 = func unit 15 bit 7;
3.23     end
3.24 end
3.25 mem_crossbar cb (256 configure bits, selected func unit inputs: bus1-8, 32 outputs
     to memory blocks);
```

Algorithm 3: Memory Routing Circuit Pseudocode

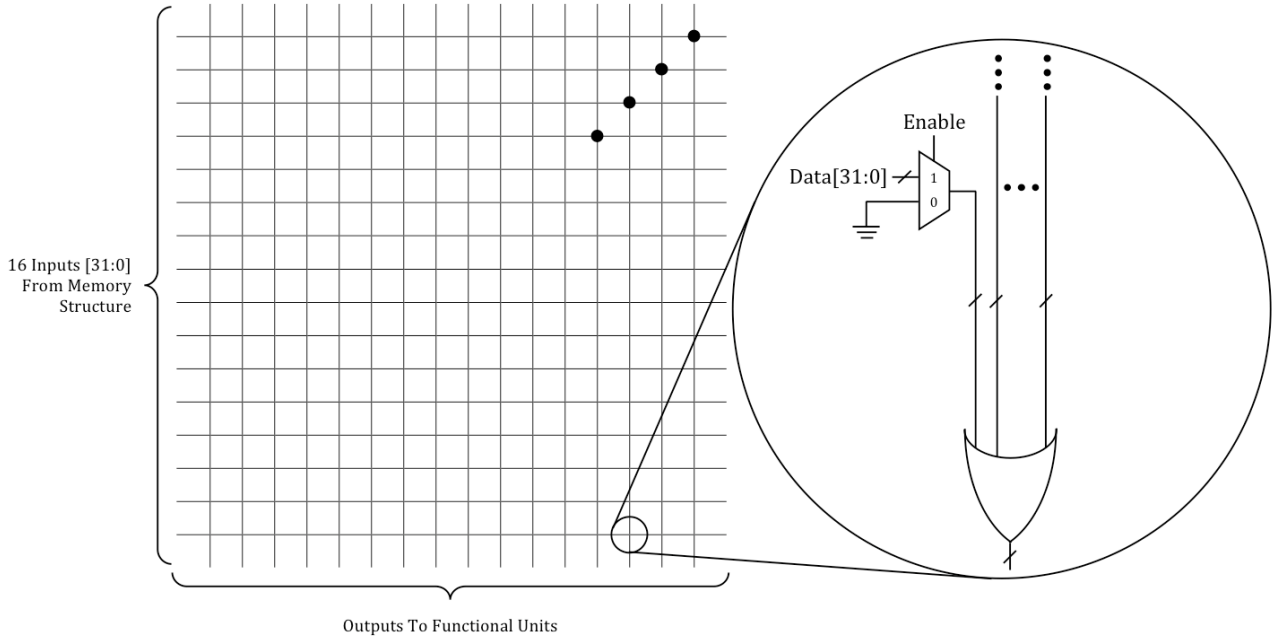

Figure 4.2: Crossbar Switch

4.1 **assign** o1 = (config bit 0 = 1) ? mem input 1 : 0;
4.2 **assign** o2 = (config bit 1 = 1) ? mem input 2 : 0;
4.3 **assign** o3 = (config bit 2 = 1) ? mem input 3 : 0;
4.4 **assign** o4 = (config bit 3 = 1) ? mem input 4 : 0;
4.5 **assign** o5 = (config bit 4 = 1) ? mem input 5 : 0;
4.6 **assign** o6 = (config bit 5 = 1) ? mem input 6 : 0;
4.7 **assign** o7 = (config bit 6 = 1) ? mem input 7 : 0;
4.8 **assign** o8 = (config bit 7 = 1) ? mem input 8 : 0;
4.9 **assign** o9 = (config bit 8 = 1) ? mem input 9 : 0;
4.10 **assign** o10 = (config bit 9 = 1) ? mem input 10 : 0;
4.11 **assign** o11 = (config bit 10 = 1) ? mem input 11 : 0;
4.12 **assign** o12 = (config bit 11 = 1) ? mem input 12 : 0;
4.13 **assign** o13 = (config bit 12 = 1) ? mem input 13 : 0;
4.14 **assign** o14 = (config bit 13 = 1) ? mem input 14 : 0;
4.15 **assign** o15 = (config bit 14 = 1) ? mem input 15 : 0;
4.16 **assign** o16 = (config bit 15 = 1) ? mem input 16 : 0;
4.17 **repeat** for o17 → o256 reusing mem input 1 → 16
4.18 **assign** output 1 = o1 → o16 as inputs to OR gate
4.19 **repeat** for outputs 2 → 16

Algorithm 4: Processor Crossbar Pseudocode

```
5.1  always @(posedge clock) begin
5.2  |  case (muxSel1)
5.3  |  0: begin
5.4  |  |  bus1_1 = matA element 1;
5.5  |  |  ...
5.6  |  |  bus1_16 = matA element 16;
5.7  |  end
5.8  |  ...
5.9  |  3: begin
5.10 |  |  bus1_1 = matD element 1;
5.11 |  |  ...
5.12 |  |  bus1_16 = matD element 16;
5.13 |  end
5.14 end
5.15 always @(posedge clock) begin
5.16 |  case (muxSel2)
5.17 |  0: begin
5.18 |  |  bus2_1 = matA element 1;
5.19 |  |  ...
5.20 |  |  bus2_16 = matA element 16;
5.21 |  end
5.22 |  ...
5.23 |  3: begin
5.24 |  |  bus2_1 = matD element 1;
5.25 |  |  ...
5.26 |  |  bus2_16 = matD element 16;
5.27 |  end
5.28 end
5.29 proc_crossbar cb1 (256 configure bits, 16 inputs from the memory structure, 16
     outputs to the function units);
5.30 proc_crossbar cb2 (256 configure bits, 16 inputs from the memory structure, 16
     outputs to the function units);
```

Algorithm 5: Processor Routing Circuit Pseudocode

4.3 Function Units

One of the key constraints for this design was being able to use function units without having access to the internals of a function unit. In order to do this, a generic interface was constructed with a clock input, function unit select signals, 32 data inputs, and 8 data outputs. For more specific routing beyond just row and column accesses, one can connect the inputs/outputs to achieve the required connections. An example of this would be to acquire a specific element of a row/column, or to output a single value to a static position in the matrix-based memory structure.

In order to fully test design functionality, ten fundamental function units were designed. Some of these function units were chosen because they are basic to matrix mathematics and the rest support the Laplace Expansion Theorem, which is the method used in the design to compute a matrix inversion [5]. The function units created perform functions such as: matrix addition, dot product, scalar divide, copying the top two rows of a matrix, copying the bottom two rows of a matrix, calculating sub-matrix C determinants, calculating sub-matrix S determinants, calculating the determinant for the entire matrix, calculating the adjugate of the upper matrix, and calculating the adjugate of the lower matrix.

4.4 Microsequencer

The control unit used in this design is a microsequencer. The microsequencer, shown in Figure 4.3, is capable of stepping through the instructions one at a time, jump to an absolute address, jump to a relative address, or perform a `nop` a specified number of times. Table 4.1 shows how the next address logic and repeat value affect the overall operation of the microsequencer.

Table 4.1: Microsequencer Next Address Logic

Type	Next[1]	Next[0]	Repeat_Value
CSAR + 1	0	0	Zero
Absolute Address	X	1	Zero
Relative Address	1	0	Zero
Repeat Current Instruction	X	X	10-bit Value

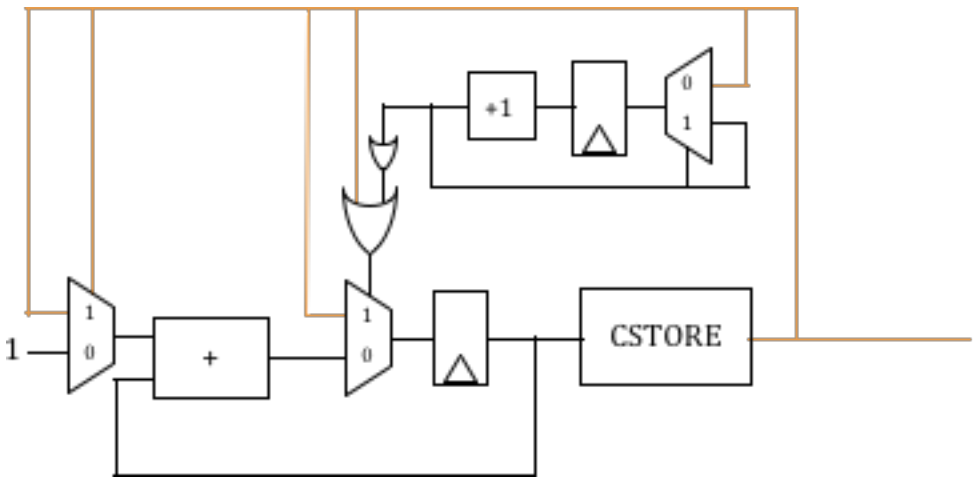

Figure 4.3: Microsequencer

This particular design uses both horizontal and vertical microinstructions. An instance where vertical microinstructions are used includes the crossbar routing bits. The microinstruction

contains addresses for four separate ROMs: `CSTORE`, `mem_store`, `op1_store`, and `op2_store`. `CSTORE` contains the microinstructions needed to operate the overall system. `Mem_store`, `op1_store`, and `op2_store` each contain 256 bits to control the crossbar routing structure of the respective crossbar switches. The microinstructions used are 52 bits in length, which are divided into 12 fields, each of which are described in detail in Table 4.2.

Table 4.2: Microsequencer Instruction Bits

Name	Position	# of Bits	Purpose
Address	[8:0]	9	Value To Be Used for Relative/Absolute Jump
Next	[10:9]	2	Configures The Next Address Logic
NB	[12:11]	2	Op2 MUX Select Line
NA	[14:13]	2	Op1 MUX Select Line
MC	[16:15]	2	Mem MUX Select Line
Op2_Switch	[21:17]	5	Op2 Crossbar Switch Configure Bits
Op1_Switch	[26:22]	5	Op1 Crossbar Switch Configure Bits
Mem_Switch	[31:27]	5	Mem Crossbar Switch Configure Bits
WE	[39:32]	8	Write Enable Bits For Memory Blocks
Mirror	[40]	1	Mirror Between Matrices A/B And C/D
Load	[41]	1	Load Values Into Matrix A
Repeat Value	[51:42]	10	Tells Microsequencer to Repeat X Times

Algorithms 6 and 7 comprise pseudocode for the microsequencer design in Verilog. The module has inputs `clock` and `reset`, and outputs `mem_mux`, `WE`, `op1_mux`, `op2_mux`, `mem_switch`, `op1_switch`, `op2_switch`, and `load`. When a reset signal is asserted, the `CSTORE` address, known as `CSAR`, is set to 0, as well as the `decrement_out`, `address`, and `next` value. When a reset signal is not asserted, `CSAR` is tied into the next address logic discussed earlier in this section, and `decrement_out` is set to the value on `mux3`, which is a counter that starts at the repeat value defined by the microprogram. The remainder of the code assigns the

microinstruction bits to either output signals or ROM addresses.

```
6.1  CSTORE cs (CLOCK, ENABLE, CSAR, DATA);
6.2  always @(posedge clock) begin
6.3  |  if reset = 1 then
6.4  |  |  CSAR = 0;
6.5  |  |  reset = 0;
6.6  |  else
6.7  |  |  CSAR = mux2;
6.8  |  end
6.9  end
6.10 always @(negedge clock) begin
6.11 |  if reset = 1 then
6.12 |  |  decrement_out = 0;
6.13 |  else
6.14 |  |  decrement_out = (mux3 = 0) ? decrement_out : mux3 - 1;
6.15 |  end
6.16 end
6.17 assign address = (reset = 1) ? 0 : data[8:0];
6.18 assign next = (reset = 1) ? 0 : data[10:9];
6.19 always @(posedge clock) begin
6.20 |  op2_mux = data[12:11];
6.21 |  op1_mux = data[14:13];
6.22 |  mem_mux = data[18:15];
6.23 |  op2_switch = tmp_op2_switch;
6.24 |  op1_switch = tmp_op1_switch;
6.25 |  mem_switch = tmp_mem_switch;
6.26 end
6.27 op2_store op2S1(clock, enable, data[23:19], tmp_op2_switch);
6.28 op1_store op1S1(clock, enable, data[28:24], tmp_op1_switch);
6.29 mem_store memS1(clock, enable, data[33:29], tmp_mem_switch);
```

Algorithm 6: Microsequencer Pseudocode 1 of 2

```
7.1  assign mux3 = (decrement_out=0) ? repeat_value : decrement_out;
7.2  or or1(bus2wire,decrement_out[0],...,decrement_out[9]);
7.3  or or2(mux2_select, next[0], bus2wire);

7.4  assign mux1 = (next[1]=1) ? address : 1;
7.5  assign sum = mux1 + CSAR;
7.6  assign mux2 = (mux2_select) ? address : sum;

7.7  always @(data[41:34]) begin
7.8  |   if data[42] = 1'b0 then
7.9  |   |   WE[3:0] = 4data[34];
7.10 |   |   ...
7.11 |   |   WE[31:28] = 4data[41];
7.12 |   |   WE[35:32] = 4'b0000;
7.13 |   |   ...
7.14 |   |   WE[63:60] = 4'b000;
7.15 |   end
7.16 |   else
7.17 |   |   WE[3:0] = 4'b0000;
7.18 |   |   ...
7.19 |   |   WE[31:28] = 4'b0000;
7.20 |   |   WE[35:32] = 4data[34];
7.21 |   |   ...
7.22 |   |   WE[63:60] = 4data[41];
7.23 |   end
7.24 end

7.25 always @(posedge clock) begin
7.26 |   load = data[43];
7.27 end

7.28 assign repeat_value = data[53:44];
```

Algorithm 7: Microsequencer Pseudocode 2 of 2

4.5 Combined System Implementation

The microprogram in Algorithm 8 is used to configure the hardware design code of the ROMs that are associated with the microsequencer. These ROMs control the routing structure within the system and the operation of the microsequencer. The syntax of this microprogram is discussed in Table 3.1. The function units implemented in the system and their position are as follows:

Table 4.3: Position of Function Units

Position	Function Unit
0:	Addition Unit
1:	Dot Product Unit
2:	Scalar Division Unit
3:	Copy Upper Two Rows of Matrix
4:	Copy Lower Two Rows of Matrix
5:	Calculate SubMatrix Determinate S Values
6:	Calculate SubMatrix Determinate C Values
7:	Calculate Determinate for Entire Matrix
8:	Calculate Adjoint for the Upper Two Rows of Matrix
9:	Calculate Adjoint for the Lower Two Rows of Matrix

The Laplace Expansion matrix inversion algorithm used is discussed in [5] and is illustrated in Figures 4.4 and 4.5. Algorithm 8 is the microprogram that reflects this procedure, and Algorithm 9 is a model of this procedure in C. The program starts by loading the external data into default matrix A. The data in matrix A is then used to calculate submatrix determinate S and C values and the results are stored in matrix B. Next, the S and C values stored in matrix B are used in the calculation of the overall determinant for the original matrix and stored in matrix C. After all the determinant values are calculated, the adjugate

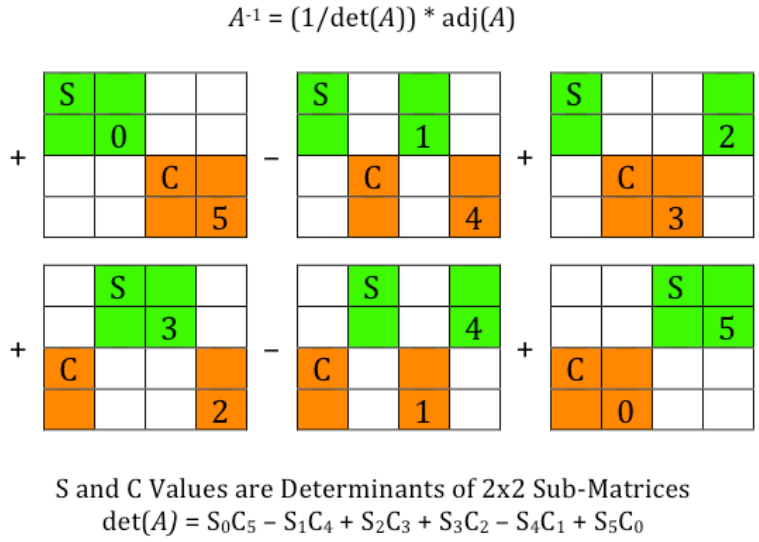

Figure 4.4: Calculating Matrix Determinant

$$\text{adj}(A) = \begin{bmatrix} +a_{11}c_5 - a_{12}c_4 + a_{13}c_3 & -a_{01}c_5 + a_{02}c_4 - a_{03}c_3 & +a_{31}s_5 - a_{32}s_4 + a_{33}s_3 & -a_{21}s_5 + a_{22}s_4 - a_{23}s_3 \\ -a_{10}c_5 + a_{12}c_2 - a_{13}c_1 & +a_{00}c_5 - a_{02}c_2 + a_{03}c_1 & -a_{30}s_5 + a_{32}s_2 - a_{33}s_1 & +a_{20}s_5 - a_{22}s_2 + a_{23}s_1 \\ +a_{10}c_4 - a_{11}c_2 + a_{13}c_0 & -a_{00}c_4 + a_{01}c_2 - a_{03}c_0 & +a_{30}s_4 - a_{31}s_2 + a_{33}s_0 & -a_{20}s_4 + a_{21}s_2 - a_{23}s_0 \\ -a_{10}c_3 + a_{11}c_1 - a_{12}c_0 & +a_{00}c_3 - a_{01}c_1 + a_{02}c_0 & -a_{30}s_3 + a_{31}s_1 - a_{32}s_0 & +a_{20}s_3 - a_{21}s_1 + a_{22}s_0 \end{bmatrix}$$

Figure 4.5: Calculating Adjoint Matrix

of the original matrix must be calculated, using values in both matrix A and B, and stored in matrix D. Finally, a scalar divide is performed on the adjugate matrix, stored in matrix D, using the overall matrix determinant, stored in matrix C, and stored back in matrix A. Below is the input matrix used for testing and its corresponding output matrix values. The values to the left are the decimal representation of the floating point hex value in parentheses.

Input Matrix:

$$\begin{pmatrix} 1.0000(3f800000) & 2.0000(40000000) & 3.0000(40400000) & 4.0000(40800000) \\ 2.0000(40000000) & 2.0000(40000000) & 6.0000(40c00000) & 4.0000(40800000) \\ 2.0000(40000000) & 2.0000(40000000) & 3.0000(40400000) & 8.0000(41000000) \\ 4.0000(40800000) & 2.0000(40000000) & 3.0000(40400000) & 4.0000(40800000) \end{pmatrix}$$

Output Matrix (inverse matrix solution found using Octave [17]):

$$\begin{pmatrix} -0.3333(beaaaaab) & -0.0000(80000000) & -0.0000(80000000) & 0.3333(3eaaaaab) \\ 1.3333(3faaaaab) & -0.5000(bf000000) & -0.5000(bf000000) & 0.1667(3e2aaaab) \\ -0.2222(be638e39) & 0.3333(3eaaaaab) & -0.0000(80000000) & -0.1111(bde38e39) \\ -0.1667(be2aaaab) & -0.0000(80000000) & 0.2500(3e800000) & -0.0833(bdaaaaab) \end{pmatrix}$$

```
8.1  load;                                   /* load the external input values */
8.2  nop, 22;
8.3  B(0,1) = A() | 5;                  /* calc S values of A, store in upper B */
8.4  nop, 23;
8.5  B(2,3) = A() | 6;                  /* calc C values of A, store in lower B */
8.6  nop, 23;
8.7  C(0) = B() | 7;          /* calc det using S/C values in B and store in C */
8.8  nop, 57;
8.9  D(0,1) = A(), B() | 8;          /* calc adj of A using B, store in upper D */
8.10 nop, 35;
8.11 D(2,3) = A(), B() | 9;          /* calc adj of A using B, store in lower D */
8.12 nop, 35;
8.13 A(0) = D(0), C(0) | 2;           /* sca div D row 0 by C, store in A row 0 */
8.14 nop, 35;
8.15 A(1) = D(1), C(0) | 2;           /* sca div D row 1 by C, store in A row 1 */
8.16 nop, 35;
8.17 A(2) = D(2), C(0) | 2;           /* sca div D row 2 by C, store in A row 2 */
8.18 nop, 35;
8.19 A(3) = D(3), C(0) | 2;           /* sca div D row 3 by C, store in A row 3 */
8.20 nop, 35;
```

Algorithm 8: Matrix Inversion Microprogram

```
9.1  float det (float a, float b, float c, float d) begin
9.2  |  return (a)*(b)-(c)*(d);
9.3  end
9.4  float pmp (float a, float b, float c, float d, float e, float f) begin
9.5  |  return (a)*(b)-(c)*(d)+(e)*(f);
9.6  end
9.7  float mpm (float a, float b, float c, float d, float e, float f) begin
9.8  |  return 0-(a)*(b)+(c)*(d)-(e)*(f);
9.9  end
9.10 int main () begin
9.11 |  a = input matrix;
9.12 |  calculate sub-matrix determinates s0 through s5 using det ();
9.13 |  calculate sub-matrix determinates c5 through c0 using det ();
9.14 |  detA = (s0)*(c5) - (s1)*(c4) + (s2)*(c3) + (s3)*(c2) - (s4)*(c1) + (s5)*(c0);
9.15 |  adjA[0][0] = pmp (a[1][1], c5, a[1][2], c4, a[1][3], c3);
9.16 |  adjA[0][1] = mpm (a[0][1], c5, a[0][2], c4, a[0][3], c3);
9.17 |  adjA[0][2] = pmp (a[3][1], s5, a[3][2], s4, a[3][3], s3);
9.18 |  adjA[0][3] = mpm (a[2][1], s5, a[2][2], s4, a[2][3], s3);
9.19 |  adjA[1][0] = mpm (a[1][0], c5, a[1][2], c2, a[1][3], c1);
9.20 |  adjA[1][1] = pmp (a[0][0], c5, a[0][2], c2, a[0][3], c1);
9.21 |  adjA[1][2] = mpm (a[3][0], s5, a[3][2], s2, a[3][3], s1);
9.22 |  adjA[1][3] = pmp (a[2][0], s5, a[2][2], s2, a[2][3], s1);
9.23 |  adjA[2][0] = pmp (a[1][0], c4, a[1][1], c2, a[1][3], c0);
9.24 |  adjA[2][1] = mpm (a[0][0], c4, a[0][1], c2, a[0][3], c0);
9.25 |  adjA[2][2] = pmp (a[3][0], s4, a[3][1], s2, a[3][3], s0);
9.26 |  adjA[2][3] = mpm (a[2][0], s4, a[2][1], s2, a[2][3], s0);
9.27 |  adjA[3][0] = mpm (a[1][0], c3, a[1][1], c1, a[1][2], c0);
9.28 |  adjA[3][1] = pmp (a[0][0], c3, a[0][1], c1, a[0][2], c0);
9.29 |  adjA[3][2] = mpm (a[3][0], s3, a[3][1], s1, a[3][2], s0);
9.30 |  adjA[3][3] = pmp (a[2][0], s3, a[2][1], s1, a[2][2], s0);
9.31 |  for i ← 0 to 3 do
9.32 |  |  for j ← 0 to 3 do
9.33 |  |  |  invA[i][j] = (1/(detA))*(adjA[i][j]);
9.34 |  |  end
9.35 |  end
9.36 |  return 0;
9.37 end
```

Algorithm 9: Matrix Inversion C Pseudo Code

In order to make the combined system implementation faster and more efficient, it was necessary to implement some sort of pipelining. In Figure 4.6, it can be observed that all of the outputs of the microsequencer are registered as well as the outputs of the memory multiplexer, memory crossbar switch, op1/op2 multiplexer, and the op1/op2 crossbar switches. As a consequence, the design has a five-stage pipeline. This affects the overall design in that when a program is written, not only must the number of clock cycles for the current function unit be taken into account, but also an additional four clock cycles so that the data can be written to memory before the next instruction executes.

4.6 Comparison with Microprocessor Implementation

In order to test and critique the specialized hardware design, it is desirable to have a second design to test against. The baseline design uses a Xilinx MicroBlaze soft-processor, which is similar in capability to standard processors. An important aspect of this testing method is that the MicroBlaze implementation uses the same FPGA as the specialized hardware design used in this thesis. Two algorithms are used on the MicroBlaze processor. The first is the Laplace Expansion algorithm; this is used because it is the algorithm used in the hardware implementation. The second is the LU decomposition algorithm; this algorithm is being used because it is an algorithm typically encountered in real world applications. The LU decomposition algorithm used is presented in [20].

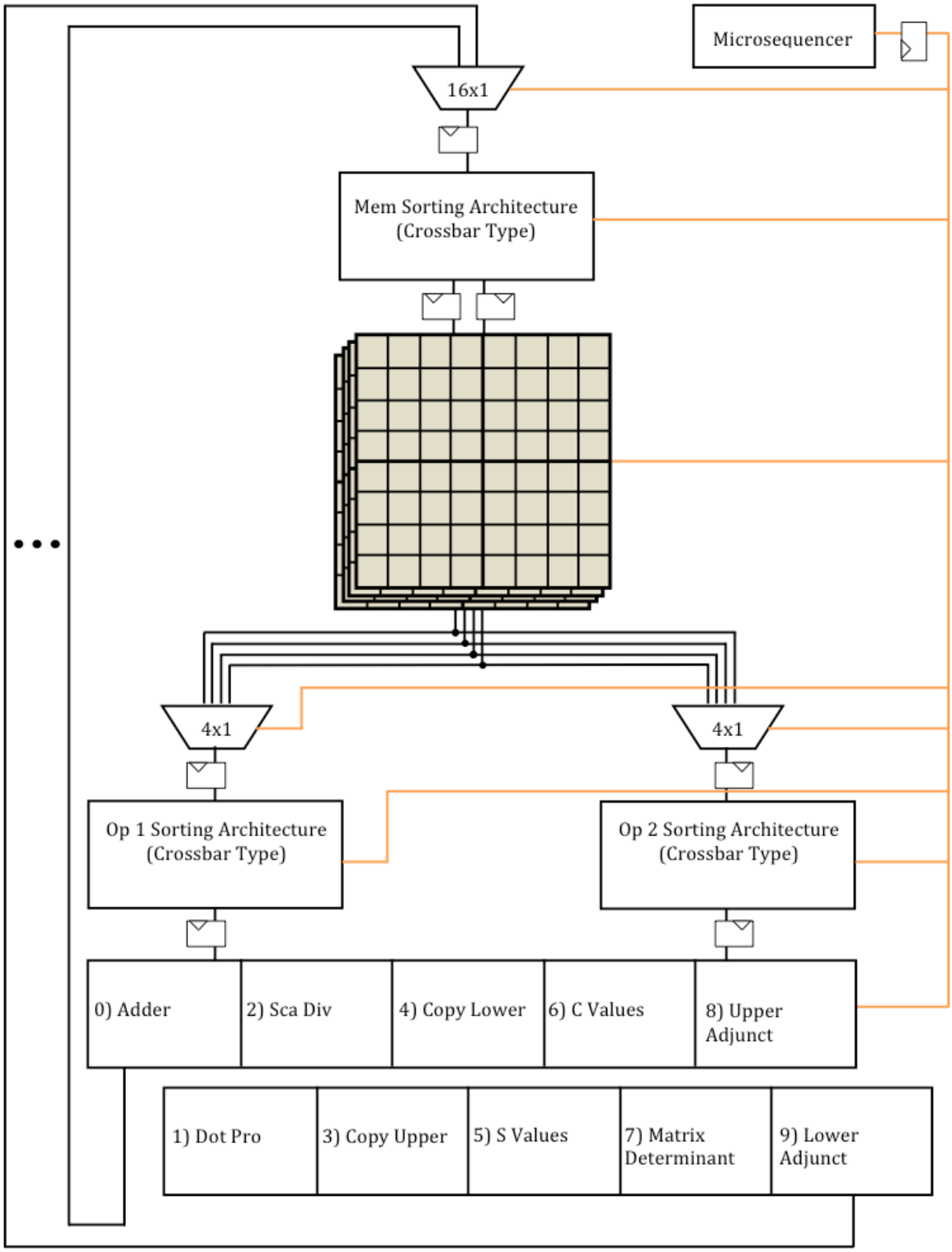

Figure 4.6: Combined System Implementaton

Chapter 5

Evaluation

The information and hardware presented in this chapter reflects a system implemented on a Xilinx Virtex-6 LX240T FPGA. Simulation and synthesis of this system has been performed on a Dell Studio XPS desktop computer with an 2.67 GHz Intel Core i7 processor and 12 GB of memory. The host operating system running is 64-bit Ubuntu version 10.10, and the version of Xilinx tools used is the ISE Design Suite 12.4: System Edition.

5.1 Small Matrix Memory Structure

The FPGA BlockRAMs have a maximum clock frequency of 599.88 MHz, which is more than adequate for use with the other hardware elements in the system. As one can see from Table 5.1 it would be very feasible to expand the amount of storage available, because the memory required is only 7.69%. However, hardware code changes would also be required to

expand the system interconnect.

Table 5.1: Memory Structure Resource Usage

Type	Used	Available	Utilization
BlockRAM	32	416	7.69%
Total Mem (KB)	1152	14976	7.69%

5.2 Crossbar Switch and System Interconnect

According to the Xilinx timing analyzer, the pipelined memory routing structure is capable of operating at a maximum speed of 376 MHz, and the processor routing structure can operate at a maximum speed of 317 MHz. These speeds are sufficient to handle the high-speed data transfers needed by the microsequencer during the execution of a microprogram. If the size of the memory structure is increased, the number of function units is increased beyond 16 units, or the data format is increased from the 32 bits currently used, then the crossbar switches and system interconnect would need to be modified. Due to the modest amount of resources required by this component, expansion would not be an issue. On the other hand, vastly expanding the crossbar switches, because of their complex structure, would cause the maximum speeds to diminish quickly. The resources used by these two routing structures are summarized in Tables 5.2 and 5.3.

Table 5.2: MemSort Resource Usage

Type	Used	Available	Utilization
Slices	1243	37680	3.29%
Slice Reg	1280	301440	0.42%
LUTs	2240	150720	1.48%

Table 5.3: ProcSort Resource Usage

Type	Used	Available	Utilization
Slices	1361	37680	3.61%
Slice Reg	2048	301440	0.67%
LUTs	2752	150720	1.82%

5.3 Function Units

For all of the function units implemented in this design, the maximum clock speed achievable is 256 MHz. The slowest function units are the ones which calculate the sub-matrix determinant values and the overall matrix determinant value. The unit calculating the overall matrix determinant is part of the critical timing path of the overall system implementation. Tables 5.4 and 5.5 show the resource usage of the function units implemented in this design. Out of all of these function units, in every area the adjoint calculation units are the most resource intensive.

Table 5.4: Function Unit Resource Usage

Type	Used	Available	Utilization
Addition			
Slices	620	37680	1.64%
Slice Reg	2592	301440	0.85%
LUTs	1658	150720	1.10%
DSP48E1	16	768	2.08%
Dot Product			
Slices	1517	37680	4.03%
Slice Reg	5584	301440	1.85%
LUTs	4022	150720	2.67%
DSP48E1	72	768	9.38%
Scalar Division			
Slices	1212	37680	3.22%
Slice Reg	5372	301440	1.78%
LUTs	3106	150720	2.06%
DSP48E1	0	768	0.00%
Copy Upper Two Rows			
Slices	58	37680	0.15%
Slice Reg	256	301440	0.08%
LUTs	2	150720	0.00%
DSP48E1	0	768	0.00%
Copy Lower Two Rows			
Slices	59	37680	0.16%
Slice Reg	256	301440	0.08%
LUTs	2	150720	0.00%
DSP48E1	0	768	0.00%

Table 5.5: Function Unit Resource Usage (continued)

Type	Used	Available	Utilization
Sub-Matrix Determinate S			
Slices	908	37680	2.41%
Slice Reg	3216	301440	1.07%
LUTs	2402	150720	1.59%
DSP48E1	48	768	6.25%
Sub-Matrix Determinate C			
Slices	907	37680	2.41%
Slice Reg	3216	301440	1.07%
LUTs	2402	150720	1.59%
DSP48E1	48	768	6.25%
Matrix Determinate			
Slices	610	37680	1.62%
Slice Reg	2256	301440	0.75%
LUTs	1615	150720	1.07%
DSP48E1	28	768	3.65%
Adjoint of Upper Two Rows			
Slices	2103	37680	5.58%
Slice Reg	7728	301440	2.56%
LUTs	5630	150720	3.74%
DSP48E1	104	768	13.54%
Adjoint of Lower Two Rows			
Slices	2113	37680	5.61%
Slice Reg	7728	301440	2.56%
LUTs	5630	150720	3.74%
DSP48E1	104	768	13.54%

5.4 Microsequencer

The final microsequencer implementation is capable of operating at a maximum speed of 198 MHz. The speed of this microsequencer, as one can see from the discussions of data in the sections above, is the major limiting factor in this design. One advantage of this design is that the controller is space efficient, as seen in Table 5.6. Also, it was fairly straightforward to implement an assembler to convert the user algorithm to synthesizeable hardware code.

Table 5.6: Microsequencer Resource Usage

Type	Used	Available	Utilization
Slices	52	37680	0.13%
Slice Reg	101	301440	0.03%
LUTs	72	150720	0.04%

5.5 Combined System Implementation

Details of the overall implementation are listed in Tables 5.7 and 5.8. The design is able to achieve a maximum clock rate of 183 MHz. The design was simulated using Xilinx's iSim, where output from the system is shown in Figure 5.1. It was found by inspection of the output that the design operates correctly and provides the expected output values, as listed in Section 4.5. A total of 335 clock cycles is required for matrix inversion, which corresponds to a total execution time of 1,829 ns.

Table 5.7: Specialized Hardware Design Power and FPU Usage

Metric	HW
Clock Speed (MHz)	183
Clock Cycles	335
Execution Time (ns)	1829
Floating Point Operations (FLOP)	143
Floating Point Operations / Sec (MFLOPS)	78.185
Power (W)	4.559
Floating Point Operations / Sec / Watt (MFLOPS/W)	17.150
Energy (μJ)	8.34
Floating Point Operations / μJ	17.146

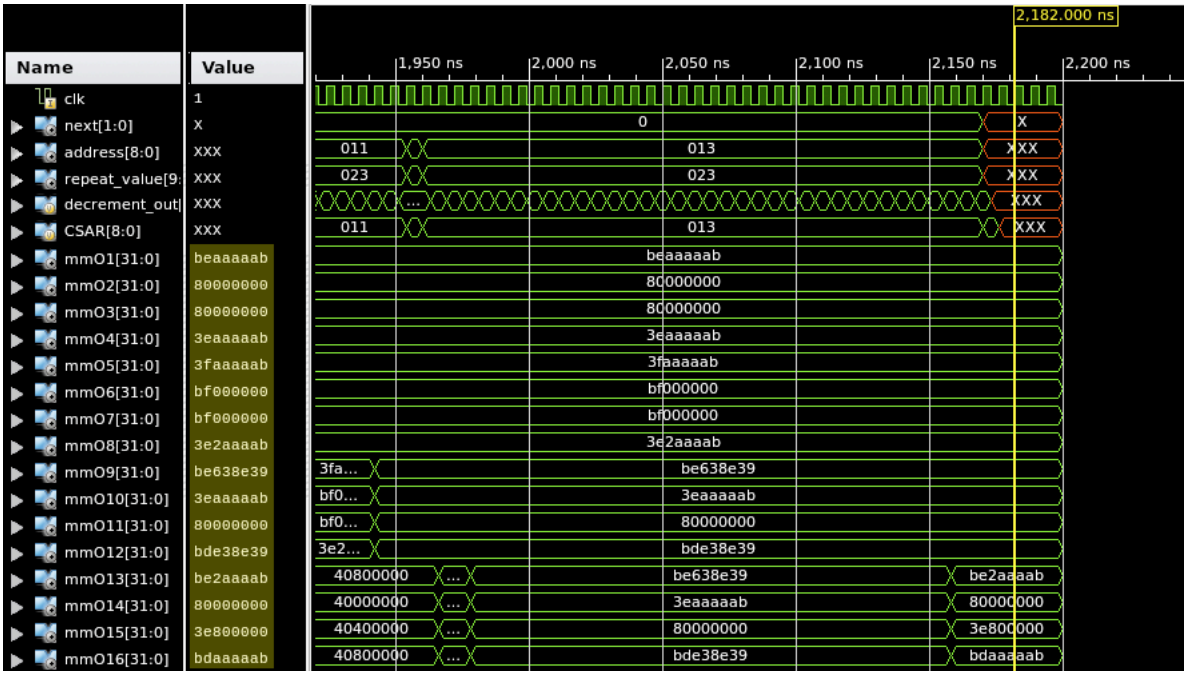

Figure 5.1: System Simulation Output

Table 5.8: Specialized Hardware Design Resource Usage

Type	Used	Available	Utilization
RAMB36E1/FIFO36E1	0	416	0.00%
RAMB18E1/FIFO18E1	64	832	7.00%
BUFG/BUFGCTRL	11	32	34.00%
DSP48E1	420	768	54.00%
Slices	12818	37680	34.00%
Slice Reg	42150	301440	13.00%
LUTs	36338	150720	24.00%

5.6 Comparison with Microprocessor Implementation

For the targeted FPGA, it was found that the MicroBlaze design was able to run at a maximum clock rate of 200 MHz, as recorded in Table 5.9. A MicroBlaze was configured to use barrel shifters and the extended floating point unit. The program written in C code was compiled using `gcc` configured with the -O3 optimization level, was run through terminal, and provided the correct output matrix. Also, no method was found to count the number of cycles required to run the program on the MicroBlaze processor without having access to the hardware being modeled. Consequently, it was required that the code be simplified, IF/FOR loops were eliminated along with all but one function call, and finally the cycle count information was calculated by hand using information provided in the MicroBlaze Reference Guide [38]. Calculations show that the LU Decomposition algorithm requires 1,646 clock cycles (8,230 ns) and the Laplace Expansion algorithm requires 1,426 clock cycles (7,130 ns). Table 5.10 shows the FPGA resources used by the microprocessor implementation.

Table 5.9: MicroBlaze Design Power and FPU Usage

Metric	LU Decomp.	Laplace Exp.
Clock Speed (MHz)	200	200
Clock Cycles	1646	1426
Execution Time (ns)	8230	7130
Floating Point Operations (FLOP)	125	166
Floating Point Operations / Sec (MFLOPS)	15.188	23.282
Power (W)	1.984	1.984
Floating Point Operations / Sec / Watt (MFLOPS/W)	7.655	11.735
Energy (μJ)	16.33	14.15
Floating Point Operations / μJ	7.655	11.731

Table 5.10: MicroBlaze Design Resource Usage

Type	Used	Available	Utilization
RAMB36E1/FIFO36E1	64	416	15.00%
RAMB18E1/FIFO18E1	0	832	0.00%
BUFG/BUFGCTRL	2	32	6.00%
DSP48E1	5	768	1.00%
Slices	1398	37680	1.00%
Slice Reg	2221	301440	1.00%
LUTs	775	150720	2.00%

In comparison with the microprocessor design, the specialized hardware is 4.5 times faster than the LU Decomposition algorithm and is 2.2 times more efficient at performing floating point operations per second per Watt of power. The specialized hardware is 3.9 times faster than the Laplace Expansion algorithm and is 1.5 times more efficient at performing floating point operations per second per Watt. A drawback is that the total number of slices required for the specialized hardware design (12,818) compared to the MicroBlaze implementation (1,398) is larger by about 9x. Both designs use the same floating point IP cores provided by Xilinx. One consideration is that the LU Decomposition algorithm run within the MicroBlaze

processor has only been optimized for the input matrix used for testing both designs, and this code would not work for all 4×4 matrices as the specialized hardware design would. Implementing the full LU decomposition algorithm on the MicroBlaze processor would likely take much longer to execute. Tables 5.11 and 5.12 show a detailed comparison between the all of the implementations.

Table 5.11: Design Performance Comparison

Metric	HW	LU Decomp.	Laplace Exp.
Clock Speed (MHz)	183	200	200
Clock Cycles	335	1646	1426
Execution Time (ns)	1829	8230	7130
Floating Point Operations (FLOP)	143	125	166
Floating Point Operations / Sec (MFLOPS)	78.185	15.188	23.282
Power (W)	4.559	1.984	1.984
Floating Point Operations / Sec / Watt (MFLOPS/W)	17.150	7.655	11.735
Energy (μJ)	8.34	16.33	14.15
Floating Point Operations / μJ	17.146	7.655	11.731

Table 5.12: Design Resource Usage Comparison

Type	Specialized Hardware	MicroBlaze
RAMB36E1/FIFO36E1	0.00%	15.00%
RAMB18E1/FIFO18E1	7.00%	0.00%
BUFG/BUFGCTRL	34.00%	6.00%
DSP48E1	54.00%	1.00%
Slices	34.00%	1.00%
Slice Reg	13.00%	1.00%
LUTs	24.00%	2.00%

Chapter 6

Conclusions

6.1 Summary

In this work, an optimized memory access framework has been implemented on a Xilinx Virtex-6 LX240T FPGA capable of implementing algorithms using a matrix-based data structure and a set of function units. The system allows for common mathematical operations on small matrices as large as 4×4. The objective was to have a system that, in the future, could potentially implement the entire MIMO decoding algorithm or something of similar complexity. Through study of the MIMO decoding algorithm, it was discovered that the most compute-intensive operation is matrix inversion. As a consequence, matrix inversion became the major focus of implementation and evaluation.

This system has its foundations in the research originally done by Karim Mohammed and

Babak Daneshrad [16]. Many changes and additions have been made to the original research in order to meet the end goal of having a system that allows implementing an algorithm requiring small matrices, and to easily integrate a set of IP blocks, under the assumption that the user may not have knowledge of the inner-workings of the IP. In other words communication synthesis has been utilized, including implementing generic function unit containers, complex programmable routing structures, an advanced memory structure, pipelined datapaths, a microsequencer to control the system, and a microprogram assembler.

The main storage of the system supports four 4×4 matrices, which demand a total of 64 memory blocks. Overall, each instantiated memory block is independent of the rest; thus, read and write operations can be done in parallel across entire matrix rows. Each memory block is dual-ported so that reading and writing can occur simultaneously, assuming the user is avoiding any type of hazards, and 32-bit floating-point data is adequate. Because the potential for the memory blocks to be a bottleneck is reduced, this type of storage system has a tremendous advantage over the typical single-ported monolithic memory which would be implemented in an average system. This architecture allows for the parallelization of many inner loops that might be encountered in a C program, resulting in an overall decrease in execution time.

The overall system framework is controlled by a microsequencer. This microsequencer controls all the major parts of the system at the same time, including the memory structure, the crossbar switches, the multiplexers, and the function units. Within this microsequencer are four ROMs: CSTORE containing the microprogram, memory crossbar switch configuration

bits, and the operand crossbar switch configuration bits for each operand. Implemented within the microsequencer is next address logic necessary to perform jumps as may be required by the microprogram.

The assembler is capable of reading in a basic program and configuring the hardware code of the microsequencer to reflect the desired operations. One disadvantage of this assembler is that it currently requires the user to manually enter in the number of `nop` cycles needed. A second disadvantage is that support for the next address logic in the microsequencer is not yet implemented.

A simple hardware code container has been written, which the user can easily integrate, allowing both function units and IP to be used with little knowledge of their inner-workings. Within each function unit container is logic that disables the unit from working unless its output has been selected for input to the memory routing structure. The final selection of function units developed is capable of performing the primary matrix operations needed by an algorithm. In particular, specialized function units were developed to perform a matrix inversion using the Laplace Expansion theorem. The results of these function units were verified with the Octave mathematical software.

Each section of the reconfigurable interconnect in this design is composed primarily of two parts: a multiplexer and a crossbar switch. There are two uses of the multiplexer; the first is to select one of the four matrices as an operand and the second is to select which of the function units is being routed to the memory structure. The purpose of the crossbar switch is to allow routing any given input to any of the available outputs. The reconfigurable

interconnect is configured by the microsequencer; more specifically, the microsequencer has the bits for the multiplexers directly in the microinstruction. It also retrieves bits for the crossbar switches from the ROMs in the microsequencer, through use of a ROM address that is part of the microinstruction.

In the planning phases of this design, the initial hope was to achieve an overall system speed of at least 100 MHz; however, the final implementation surpassed this goal and achieved an overall speed of approximately 183 MHz. The addition of a five-stage pipeline to the design, which breaks up the long critical paths, was the major contributor to performance. Currently, the microsequencer limits the overall design speed since its maximum speed is only 198 MHz; the other major parts of the system which are not user-defined are the memory routing structure (376 MHz), the processor routing structure (317 MHz), and the memory structure (600 MHz).

In order to test design performance, an FPGA configured with a MicroBlaze soft processor running two algorithms, a highly modified form of the LU decomposition algorithm and the Laplace Expansion algorithm used in the hardware implementation, was used as a benchmark. The LU Decomposition algorithm was modified in such a way that the code was no longer generalized, but instead was changed to include as few branches as possible by evaluating beforehand how conditional operators would react and to hard-code that behavior. The MicroBlaze system is capable of speeds at 200 MHz and has an overall execution time of 8,230 ns for the LU Decomposition and 7,130 for the Laplace Expansion algorithm. After testing it was determined that the system designed here is 4.5 times faster than the LU

Decomposition algorithm, and 3.9 times faster than the Laplace Expansion algorithm.

6.2 Future Work

The major objectives set out for this design have been achieved, but there exist additional objectives which could optimize and improve the overall system. Some of these future objectives include function units operating in parallel, integration of advanced routines in the microprogram, and support for non-standard data access patterns to retrieve data from the matrix data structure. All of these objectives have the potential of speeding up the overall system framework as well as making it a more powerful and user-friendly tool.

As mentioned, the function units do not work in parallel as was initially hoped. In order to implement parallelism it would be necessary to redesign both the assembler and the hardware design in such a manner that the five-stage pipeline is exploited for more than just breaking apart the critical path. Required hardware changes would be mostly in the design of the microsequencer and the function units containers. The containers would need to have counters that have the number of cycles required hard-coded into them. Also, the function unit containers would need signals that indicate when the unit is ready to receive data and when it is done processing data. A problem that would arise is function units finishing out of order and sending or receiving incorrect data. In order to counteract these problems, the function units would depend on the microsequencer for management. Also, FIFOs would be required on function unit inputs and outputs so that the routing structure could be changed

in each instruction.

Also, the microsequencer has next address logic, which is necessary to perform both absolute and relative jumps as may be required by the microprogram, but this is not currently available at the software level. These features are not being used because their implementation requires that the entire assembler be restructured. One of the first modifications to the assembler would be handling all of the conditional statements that a normal programming language would implement such as: subroutines, while loops, for loops, if-then-else statements, and branching. Implementing such behavior requires a two-pass methodology, or with much effort the single-pass method discussed, in order to perform jumps.

Because the operand choices are limited to rows, columns, and entire matrices, the assembler does not currently utilize the full potential of the crossbar switches. The ability to choose more specific matrix elements would once again require major changes to the assembler. It is important to note that these challenges can be overcome in the hardware through carefully choosing how the outputs of the operand routing structures are mapped to the inputs of the function units and how the outputs of the function units are mapped to the inputs of the memory routing structure. One instance of this in the implemented system is how the submatrix determinant values are stored and retrieved.

There are a few, but important, improvements that can be made to the overall system. Hardware can be improved by parallelizing function units, adding control signals to the function unit containers, adding more management support to the microsequencer, and by adding FIFOs in the function unit containers. Additionally, an overall scheduling scheme is

required. The software needs support added to the assembler for conditional statements and subroutines, as well as support for more flexible access to matrix memory elements. All of these changes would contribute to a system that maximizes work done each clock cycle, as well as a system capable of handling advanced algorithms in a more efficient manner.

Bibliography

[1] Altera. Qsys system integration tool. http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html.

[2] OCP-IP Association. Ocp-ip : Home page. http://www.ocpip.org/, 2011.

[3] Reinaldo Bergamaschi. IBM research — projects — Coral. http://www.research.ibm.com/da/coral.html.

[4] Kyusun Choi and William S. Adams. VLSI implementation of a 256 x 256 crossbar interconnection network. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 289–293, Washington, DC, USA, 1992. IEEE Computer Society.

[5] David Eberly. The Laplace expansion theorem: Computing the determinants and inverses of matrices. http://www.geometrictools.com/Documentation/LaplaceExpansionTheorem.pdf, August 2008.

[6] Till Fischer. FPGA crossbar switch architecture for partially reconfigurable systems. *Karlsruhe Institute of Technology*, 2009.

[7] Lan Gao. FLEX tutorial. `http://alumni.cs.ucr.edu/~lgao/teaching/flex.html`, 2011.

[8] M. Gasteier, M. Munch, and M. Glesner. Generation of interconnect topologies for communication synthesis. In *Design, Automation and Test in Europe, 1998., Proceedings*, pages 36 –42, February 1998.

[9] N.D. Hemkumar. Efficient VLSI architectures for matrix factorizations. *Ph.D. Dissertation, Rice University, Houston, TX*, 1994.

[10] IEEE. IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing ip within tool flows. `http://standards.ieee.org/getieee/1685/download/1685-2009.pdf`, February 2010.

[11] Sonics Inc. Sonics, Inc. - the official website - moving SoC design beyond the NoC. `http://www.sonicsinc.com/`.

[12] Sonics Inc. Sonics OCP library for verification (SOLV). `http://www.sonicsinc.com/SOLV_Product_Brief.htm`, 2011.

[13] Intel. Details about MMX(tm) technology intrinsics. `http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/`

cpp/win/intref_cls/common/intref_mmx_details.htm#intref_mmx_details, 1996-2010.

[14] Oak Ridge National Laboratory. 3.3.1 interleaved memory. http://www.phy.ornl.gov/csep/ca/node19.html.

[15] MindSpeed. M21170 3.2 Gbps 288x288 crosspoint switch. http://www.mindspeed.com/assets/001/34729.pdf/, 2010.

[16] K. Mohammed and B. Daneshrad. A programmable accelerator for next generation wireless communications. *17th European Signal Processing Conference*, pages 1294–1298, 2009.

[17] Octave. Octave. http://www.gnu.org/software/octave/, June 2011.

[18] R.B. Ortega and G. Borriello. Communication synthesis for distributed embedded systems. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*, pages 437 – 444, November 1998.

[19] Alessandro Pinto, Alvise Bonivento, Allberto L. Sangiovanni-Vincentelli, Roberto Passerone, and Marco Sgroi. System level design paradigms: Platform-based design and communication synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 11:537–563, June 2004.

[20] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C book set: Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge University Press, 1992.

[21] The Flex Project. flex: The fast lexical analyzer. `http://flex.sourceforge.net/`, 2011.

[22] D. Salomon. *Assemblers and Loaders*. Ellis Horwood, New York, 1992.

[23] P. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer Circuits and Systems Series, July 2010.

[24] Stefano Tommesani. MMX primer. `http://www.tommesani.com/MMXPrimer.html`, 2004.

[25] UCLA. The non-engineer's introduction to MIMO and MIMO-OFDM. `http://www.mimo.ucla.edu/`.

[26] Wikipedia. Bell laboratories layered space-time - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Bell_Laboratories_Layered_Space-Time`.

[27] Wikipedia. IEEE 802.11n-2009 - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/802.11n`.

[28] Wikipedia. MIMO - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/MIMO`.

[29] Wikipedia. Orthogonal frequency-division multiplexing - wikipedia, the free encyclopedia.

`http://en.wikipedia.org/wiki/Orthogonal_frequency-division_multiplexing`.

[30] Wikipedia. Spatial multiplexing - wikipedia, the free encyclopedia.

`http://en.wikipedia.org/wiki/Spatial_multiplexing`.

[31] Wikipedia. Register file - wikipedia, the free encyclopedia.

`http://en.wikipedia.org/wiki/Register_file`, November 2010.

[32] Wikipedia. Clos network - wikipedia, the free encyclopedia.

`http://en.wikipedia.org/wiki/Clos_network`, June 2011.

[33] Wikipedia. MMX (instruction set) - wikipedia, the free encyclopedia.

`http://en.wikipedia.org/wiki/MMX_(instruction_set)`, May 2011.

[34] Wikipedia. SIMD - wikipedia, the free encyclopedia.

`http://en.wikipedia.org/wiki/SIMD`, May 2011.

[35] Xilinx. Xilinx DS150 Virtex-6 family overview.

`http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`, 2009.

[36] Xilinx. Xilnx UG364 Virtex-6 FPGA configurable logic block user guide.

`http://www.xilinx.com/support/documentation/user_guides/ug364.pdf`, 2009.

[37] Xilinx. Xilnx UG369 Virtex-6 FPGA DSP48E1 slice user guide.

`http://www.xilinx.com/support/documentation/user_guides/ug369.pdf`, 2009.

[38] Xilinx. MicroBlaze processor reference guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/mb_ref_guide.pdf`, 2010.

[39] Xilinx. Xilinx UG360 Virtex-6 FPGA configuration user guide. `http://www.xilinx.com/support/documentation/user_guides/ug360.pdf`, 2010.

[40] Xilinx. Xilnx UG363 Virtex-6 FPGA memory resources user guide. `http://www.xilinx.com/support/documentation/user_guides/ug363.pdf`, 2010.

[41] Steve Young, Peter Alfke, Colm Fewer, Scott McMillan, Brandon Blodget, and Delon Levi. A high I/O reconfigurable crossbar switch. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–, Washington, DC, USA, 2003. IEEE Computer Society.

Acronyms

ADCs analog-to-digital converters. 9

CLB Configurable Logic Block. 15

DACs digital-to-analog converters. 9

EDK Embedded Development Kit. 11, 16

FFT Fast Fourier Transform. 9

FPGA Field Programmable Gate Array. 1, 2, 4, 6, 7, 8, 11, 13, 14, 15, 16, 17, 21, 22, 24, 29, 32, 51, 53, 60, 63, 66

HDL Hardware Design Language. 16, 17

IP intellectual property. 1, 2, 3, 11, 12, 13, 14, 16, 17, 61, 64, 65

ISE Integrated Synthesis Environment. 11, 16, 17, 21, 22, 53

LUTs Look-Up Tables. 15

MIMO multiple-in-multiple-out. 2, 8, 9, 18, 19, 32, 34, 63

MMX Intel Multimedia Instructions. 24

NoC Network on Chip. 12, 13

OCP-IP Open Core Protocol International Partnership. 13, 14

OFDM Orthogonal Frequency Division Multiplexing. 8, 9

PIPs Programmable Interconnect Points. 21

SIMD Single Instruction Multiple Data. 23, 24

SoC System on Chip. 12

SOLV Sonics OCP Library for Verification. 14

SVD Singular Valued Decomposition. 18

V-BLAST Vertical-Bell Laboratories Layered Space-Time. 9

XML eXtensible Markup Language. 13

XPS Xilinx Platform Studio. 11, 12