Fast Fourier Transform on a 3D FPGA

by

Elizabeth Basha

B.S. Computer Engineering, University of the Pacific 2003

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author .....
Department of Electrical Engineering and Computer Science
August 19, 2005

Certified by ....
Krste Asanovic
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Fast Fourier Transform on a 3D FPGA

by

Elizabeth Basha

Submitted to the Department of Electrical Engineering and Computer Science
on August 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

Abstract

Fast Fourier Transforms perform a vital role in many applications from astronomy to cellphones. The complexity of these algorithms results from the many computational steps, including multiplications, they require and, as such, many researchers focus on implementing better FFT systems. However, all research to date focuses on the algorithm within a 2-Dimensional architecture ignoring the opportunities available in recently proposed 3-Dimensional implementation technologies. This project examines FFTs in a 3D context, developing an architecture on a Field Programmable Gate Array system, to demonstrate the advantage of a 3D system.

Thesis Supervisor: Krste Asanovic
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

First, I would like to thank NSF for the fellowship and IBM for funding this project. Thank you to Krste Asanovic for advising me on this project and the research group for their friendly encouragement. Jared Casper and Vimal Bhalodia helped with various FPGA work and debugging some of my more interesting problems.

I would next like to thank TCC and FloodSafe Honduras for all of their support and understanding these past couple months. ESC is an exceptional group of people; without their encouragement I would have been lost. Christine Ng and Adam Nolte, your support and friendship are greatly appreciated, especially during thesis discussions.

Finally, thank you to my family for cheering me on throughout, commiserating with my frustrations, and sharing in my excitements. Mom, I would not have made it this far without you-thank you always!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Discrete Fourier Transform converts data from a time domain representation to a frequency domain representation allowing simplification of certain operations. This simplification makes it key to a wide range of systems from networking to image processing. However, the number of operations required made the time-to-frequency conversion computationally expensive until the development of the Fast Fourier Transform (FFT), which takes advantage of inherent symmetries. The FFT still requires significant communication and data throughput resulting in several variations on the algorithm and implementations. This project develops a new implementation of the FFT, improving further by developing an implementation for a 3-Dimensional Field Programmable Gate Array (FPGA) system.

The 3D FPGA system consists of several FPGA chips, each connected to a bank of Dynamic Random Access Memory (DRAM) chips within a single package (see Figure 1-1), allowing improved communication between the FPGAs, and between a FPGA and its DRAMs. This decrease in the delay and cost of inter-chip communication increases the speed of the algorithm. In addition, the use of programmable FPGA logic decreases the implementation time as development of the algorithm and verification can be accomplished in-situ on the actual hardware, and increases the flexibility of both the algorithm design and the variety of algorithms the system can run.

This project focuses on large FFTs, designing a scalable 2^{20}-point implementation. The FFT uses a 4-step approach to create an iterative algorithm that allows for a variable number of FPGAs. Due to the lack of manufactured 3D FPGA systems, the design targets a single Xilinx Virtex2 4000 chip for implementation. Based on measurements from this single Virtex2, a model describing the IO, area and power effects extrapolates a multi-layer, multi-chip system using Xilinx Virtex2 Pro 40 FPGAs. This then is compared to other FPGA and ASIC FFT designs to determine the improvements possible with 3D FFT structures.

The thesis initially introduces some background of FFT algorithms and related work, followed by both a Unit-Transaction Level and a Micro-Architectural system description, concluding with the multi-chip model and results.

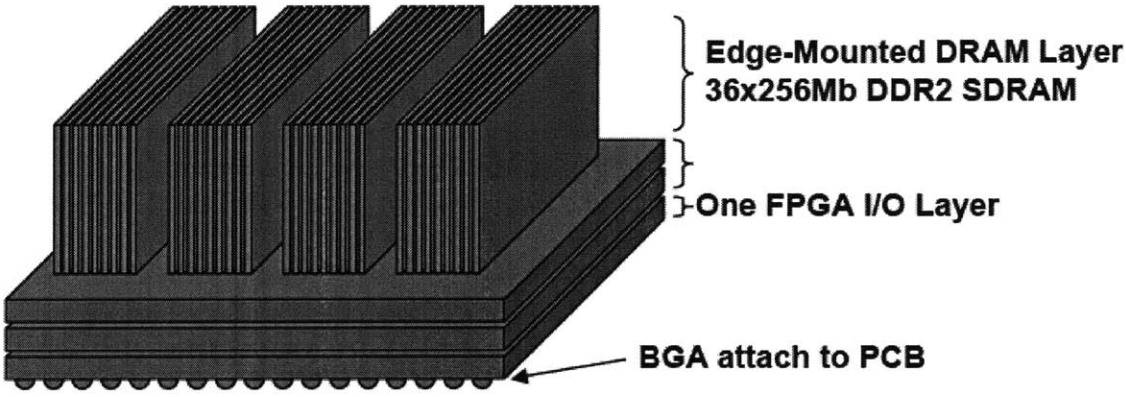

Figure 1-1: Overall 3D FPGA Diagram

Chapter 2

Background and Related Work

This chapter discusses the background and design points of Fast Fourier Transform algorithms, describes relevant hardware implementations, and concludes with an overview of 3-Dimensional implementation technologies.

2.1 FFT History and Algorithms

The Fast Fourier Transform exploits the symmetry of the Discrete Fourier Transform to recursively divide the calculation. Carl Gauss initially developed the algorithm in 1805, but it faded into obscurity until 1965 when Cooley and Tukey re-developed the first well known version in a Mathematical Computing paper [24]. Once re-discovered in an era of computers, people realized the speedup available and experimented with different variations. Gentleman-Sande [12], Winograd [25], Rader-Brenner [21], and Bruun [7] are just a few of the many variants on the FFT.

To distinguish between these different algorithms, a number of variables describe an FFT algorithm, including the method of division, the size of the smallest element, the storage of the inputs and outputs of the calculation, and approach to large data sets. The basic structure of the algorithm consists of scaling steps and combination steps (see Figure 2-1). As will be discussed below, the order of scaling and combination influences the division design decision of the algorithm as well as the storage of the inputs and outputs. The number of steps depends on the number of inputs into the algorithm, also referred to as points and commonly represented by N, as well as the size of the smallest element, also called the radix. The choice of approach to large data sets affects both the radix and the division of the algorithm.

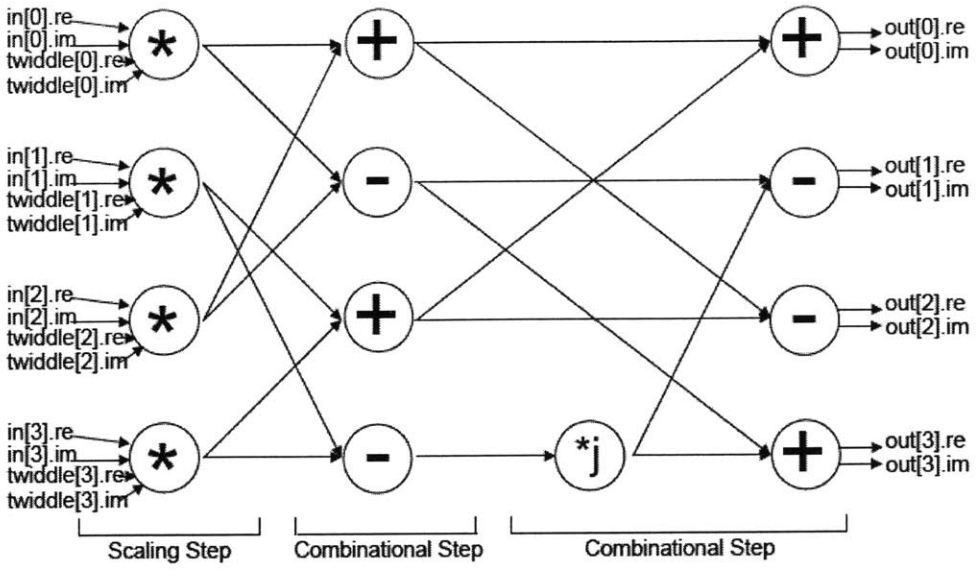

Figure 2-1: Example of Basic FFT Structure Consisting of Scaling and Combinational Steps

Division Method Two methods exist to recursively divide the calculation. Mathematically, either the data can be divided at the time-domain input resulting in a method called Decimation in Time (DIT) or the data can be divided at the frequency-domain output resulting in a Decimation in Frequency (DIF) method (see Figure 2-2). Within an algorithm, this affects where the scaling of the data occurs. A DIT algorithm multiplies the inputs by the scaling factor (also know as a twiddle factor) before combining them; a DIF multiplies after the combination.

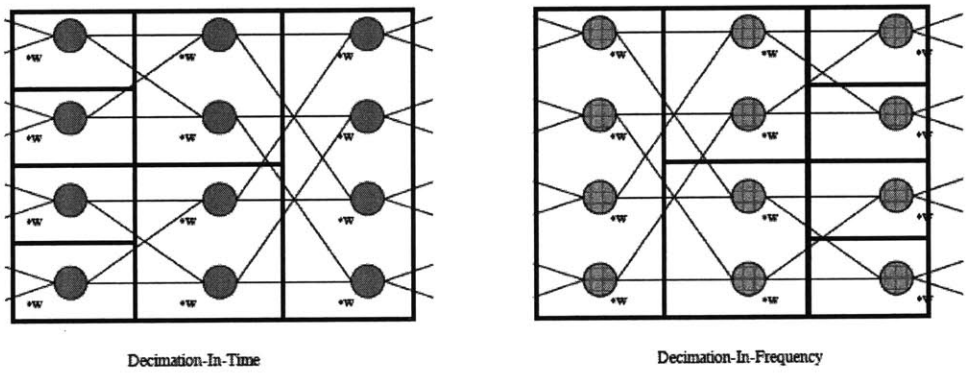

Figure 2-2: DIT Compared to DIF Diagram

Radix Size The divisor used within the recursion defines the radix size, which in turn defines the smallest element of the calculation. Mathematically, if an N-point FFT is performed in radix R, the number of stages required is $\frac{log(N)}{log(R)}$ and the number of radix nodes per stage is $\frac{N}{R}$, resulting in $(\frac{N}{R}) * (\frac{log(N)}{log(R)})$ radix nodes for the algorithm (see Figure 2-3). Complexity of the smallest element increases with radix size, while the number of multiplies decreases. Commonly power-of-two radices are used; if the number of inputs is odd, then mixed-radix algorithms are required increasing the complexity of the system.

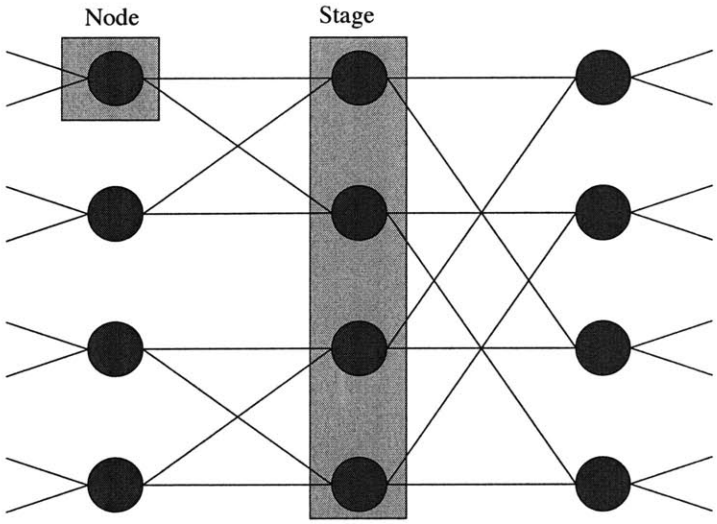

Figure 2-3: Radix Size Depiction

Input and Output Storage Certain algorithms allow the same location to be read for input and written for output. This category of in-place algorithms decreases the storage size required, but increases the complexity especially when dealing with custom hardware. The other option doubles the amount of storage required by reading from one location and writing to a different location (see Figure 2-4). Other considerations include the stride access of the data: is the data loaded into the location so that the algorithm can read it out unit sequentially or does the data need to be read in stride patterns based on the radix or point size of the algorithm?

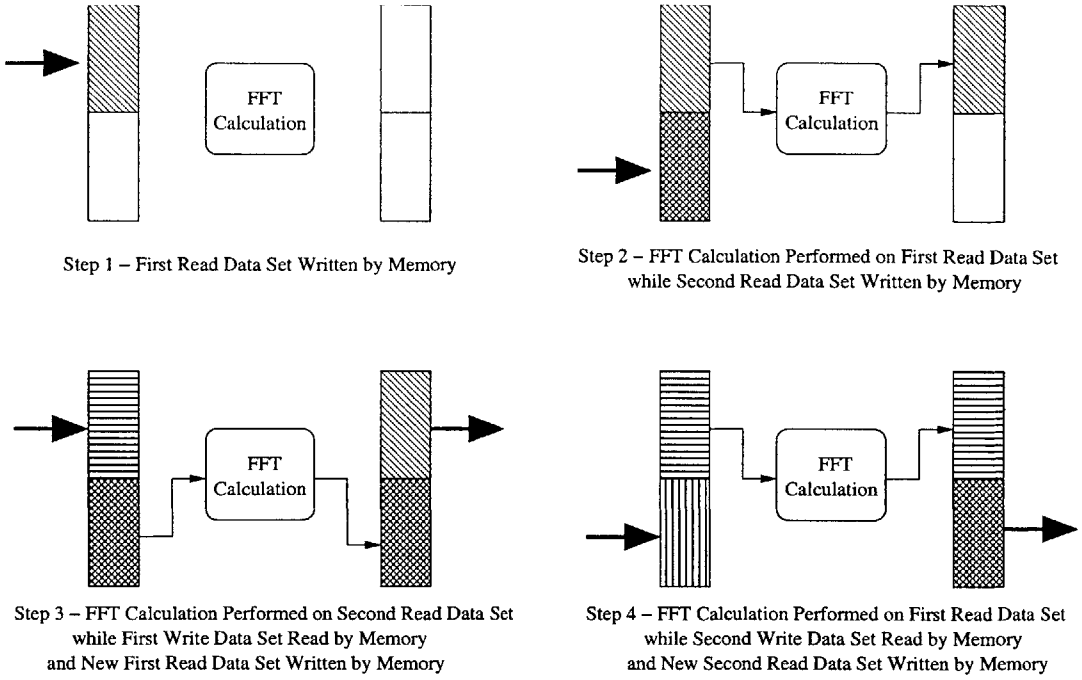

Figure 2-4: FFT Calculation and Memory Flow Diagram

Large Data Set Approach If, as in the case of this project, the data set exceeds the implementable FFT datapath size, two approaches can be used. First, the same algorithm used if the datapath size was possible can be utilized by performing sections of the calculation on the implementable subset of the hardware. This allows the computation to be scheduled in time on a small portion of the hardware. This complicates the scheduling and increases the complexity of generating and scheduling the scaling (or twiddle) factors for the scaling step. These design choice trade-offs lead to another option involving changing the algorithm used to a 4-step algorithm by mapping the 1D data set to a 2D data set (see Figure 2-5). This algorithm developed by Gentleman and Sande [12] consists of viewing an N-point FFT as a (n*m)-point FFT, where N=n*m. First, n m-point FFTs are performed, covering the rows of the array. The results are multiplied by a twiddle factor, based on N, and then transposed. Step four involves m n-point FFTs, covering the new transposed rows of the array. This 4-step algorithm reduces the amount of on-chip memory space by requiring only a subset of the twiddle factors and allows for better hardware reuse.

2.2 FFT Hardware Implementations

Past implementations focused on two different options - either to develop an algorithm for a processor or other pre-built hardware system, or to develop new hardware, usually consisting of either an application specific integrated circuit (ASIC) design

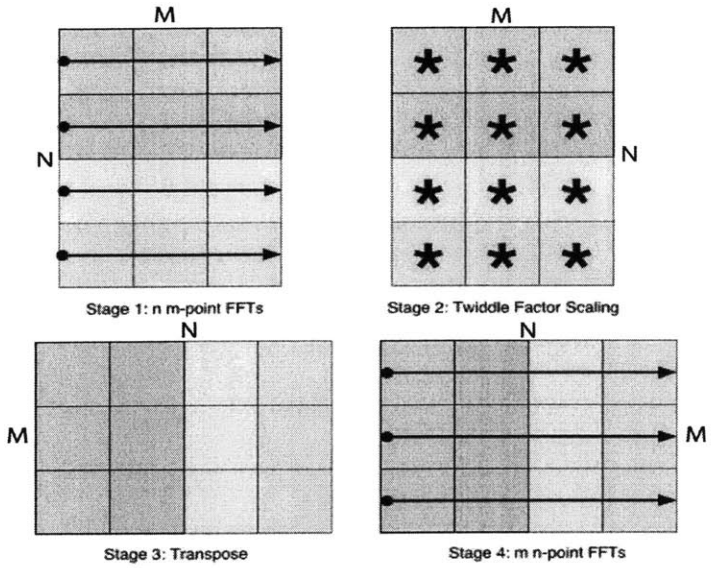

Figure 2-5: 4-Step FFT Method

or, recently, an FPGA design. An FPGA design, which this project explores, allows the hardware to focus on the specific complexities of the FFT, creating a more efficient and faster system at a loss of generality. Although both ASIC and FPGA designs are more customized than software algorithms, these differences in the architectures vary due to the unique features of both implementation options.

2.2.1 ASIC Designs

ASIC designs vary greatly in size and implementation from the 64-point requirements of a 802.11a wireless processor to the 8K needs of a COFDM digital broadcasting system to the 1M needs of a radio astronomy array. The key divisions between these designs are the architecture of the FFT calculation and memory, the architecture of the FFT itself, and the FFT algorithm used. As outlined in Baas [3], five options exist for the FFT-Memory architecture: a single memory module and a single FFT module, dual memory modules and a single FFT module, multiple pipelined memory and FFT modules, an array of memory and FFT modules, or a cache-memory module and a single FFT module (see Figure 2-6).

Within these options, the FFT can consist of one computational element (also called a butterfly) performing the complete FFT serially, a stage of the butterflies performing the computation in parallel, some pipelined combination of the two, or splitting the calculation into two separate calculations where each can either be a parallel or serial implementation. Gentleman and Sande introduced this latter method known as the 4-step method as, in addition to the two calculation steps, it requires

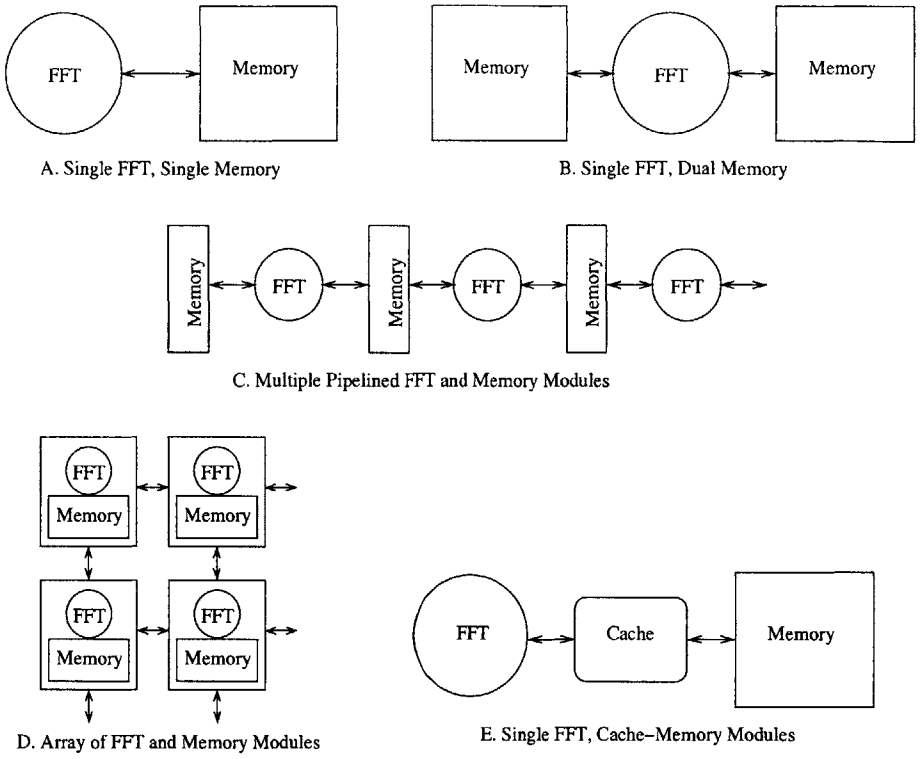

Figure 2-6: FFT-Memory Architectures Options

an additional multiplication step and a transform step (see Figure 2-5, Section 2.1). Finally, the FFT algorithm used can vary widely due to the different parameters available (see Section 2.1).

Combinations of the above options create a wide variety of FFT ASIC implementations, mostly for small FFT sizes (see Table 2.1 for a numeric view). To reduce the number of complex multiplications, Maharatna [18] implemented a 64-point FFT using the 4-step method with no internal memory while a slightly bigger 256-point implementation by Cetin [8] serially reused one butterfly with ROM storage for the scaling factors and RAM storage for data. A similar 1024-point implementation by Baas [3] also used a serially pipelined element with local ROM storage, but inserted a cache between the FFT processor and the data RAM. Instead of only one butterfly, Lenart [16] unrolled the calculation and pipelined each stage with small memory between stages; a larger point implementation by Bidet [6] implemented the same style as did Park [20] although that implementation increased the bit width at each stage to reduce noise effects.

A more appropriate comparison to the proposed design is a 1M FFT designed by Mehrotra [19]. Their system divides into four 64-point units each consisting of radix-8 butterflies. DRAMs bracket these four units to create the complete system. The calculation accesses the four units four times, calculating a stage of the computation

Designer	FFT Points	ASIC Process (μm) or FPGA Type	Area (ASIC=mm^2, FPGA=LUTs)	Freq.	Calculation Type (μs)
Maharatna[18]	64	0.25	13.50	20 MHz	2174.00
Cetin[8]	256	0.70	15.17	40 MHz	102.40
Baas[3] @ 3.3V	1024	0.60	24.00	173 MHz	30.00
Lenart[16]	2048	0.35	~6	76 MHz	27.00
Bidet[6]	8192	0.50	1000.00	20 MHz	400.00
Fuster[10]	64	Altera Apex	24320	33.54 MHz	1.91
Sansaloni[23] (1 CORDIC)	1024	Xilinx Virtex	626	125 MSPS	8.19
Dillon Eng.[13]	2048	Xilinx Virtex2	9510	125 MHz	4.20
RF Engines[17]	16K	Xilinx Virtex2	6292	250 MSPS*	65.53
Dillon Eng.[9]	512K	Xilinx Virtex2 Pro	12500	80 MHz	Unknown

Table 2.1: Comparison of ASIC and FPGA FFT Implementations *MSPS=MegaSamplesPerSecond

each time. 64 DRAM chips store all data with one set designated as the input to the units and other as output during the first stage, and this designation alternating at each latter stage.

2.2.2 FPGA Designs

Due to their more limited space and speed capabilities, FPGA designs vary slightly less than ASIC designs, although they share the same key divisions (see Table 2.1 for a numeric comparison). Multiplications, being the most costly both in terms of area required and calculation time, are the common focus of optimizations. To perform a 1024-point FFT, Sansaloni [23] designed a serial architecture reusing one butterfly element with a ROM for scaling and a RAM for data storage. However, to reduce the multiplier area required, the butterfly performs the scaling at the end with a serial multiply for both complex inputs, requiring only one multiplier instead of the usual four. Another method for reducing the multiplication, used in Fuster [10], split the 64-point calculation into two 8-point calculations, an unrolling step that requires no multiplies within these two calculations. This 4-step algorithm does require a scaling multiply between the two stages; however this consists of 3 multipliers compared to the 24 necessary for the standard 64-point implementation.

Larger point FFT designs within FPGAs focus more on the overall structure of the computation. RF Engines Limited [17] produced a series of pipelined FFT cores using a typical approach of unrolling the complete algorithm with pipelines at each stage of the calculation. This design is parameterizable and can compute up to 64K-point FFTs. Another company, Dillon Engineering [9] [13], approaches large FFTs differently. Their design computes 256M-point FFTs using the 4-step

method. External memory stores the input, which first is transposed before entering the first FFT calculation. The transpose step also accesses an external SRAM for temporary storage of some values, followed by a twiddle multiply stage and the second FFT calculation. A final matrix transpose reorders the data and stores it in another external SRAM.

2.3 3D Architectures

Two different approaches define current 3D architectures. First, standard processes build the chip by creating several wafers. These wafers glue together in either a face-to-face, back-to-back or face-to-back configuration with metal vias carefully placed to connect the layers thus creating a 3D system. Another option builds a 3D system out of existing chips, known as a Multi-Chip Module (MCM), embedding them in a glue layer with routed wires to interconnect the various modules. Production of either of these approaches challenges current process technologies, especially in areas such as heat dissipation and physical design [1]. However, while waiting for process technologies to advance, researchers use modeling and CAD tool design to explore the potential of 3D systems, determining the performance benefits and discussing solutions for issues like routing between layers. Rahman [22] found that interconnect delay can be decreased as much as 60% and power dissipation decreased up to 55% while Leeser [15] discussed a potential design for connecting the routing blocks of one layer to another. Yet, few studies examine the architectural possibilities of 3D systems nor possible implementations on 3D FPGAs. Alam [2] test their layout methodology by simulating an 8-bit encryption processor and Isshiki [14] develops a FIR filter for their MCM technology. To begin to fill this void, this thesis examines the architectural possibilities and implementations of FFTs on 3D systems.

Chapter 3

System Design Description

This chapter describes the complete system, the system divided into four FPGAs, and a single FPGA within the system. Each description breaks down into a Unit-Transaction Level (UTL) perspective first and then the UTL translates into a micro-architectural description of hardware implemented.

3.1 System UTL Description

To understand the overall requirements of the system, a UTL description explains what the overall system needs to do. As Figure 3-1 demonstrates, the system accepts N complex data values, performs an FFT calculation, and outputs N complex data values.

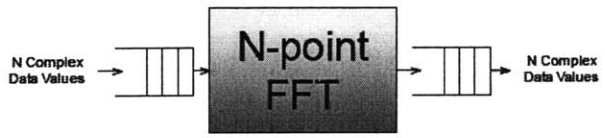

Figure 3-1: System UTL Block Diagram

3.2 System Micro-Architectural Description

The overall system consists of one 3D block using a Ball Grid Array (BGA) attachment to a Printed Circuit Board (PCB) (see Figure 1-1).

3.3 Four FPGA UTL Description

One level below this UTL describes the connections of the different sub-blocks (see Figure 3-2). Each sub-block is identical and each computes an equal portion of the calculation, communicating to the others via queues, with each knowing which data

to communicate to the others in which order such that no conflicts occur. For the purposes of this system, the system contains four sub-blocks although, since all are equivalent and share the calculation equally, the system as designed can work with any number of sub-blocks. Four reasonably describes the feasible number of FPGAs achievable with current processes and sub-divides the required calculation evenly.

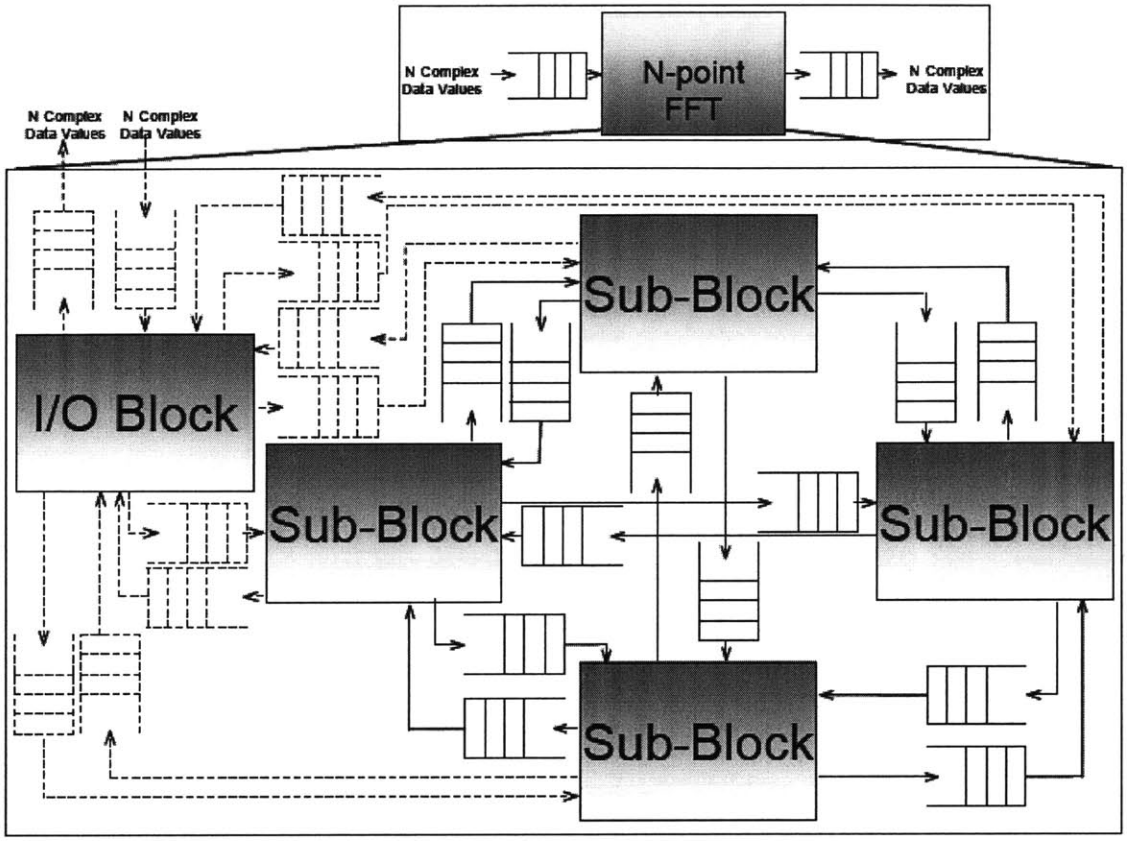

Figure 3-2: Four FPGA UTL Block Diagram

Figure 3-3 demonstrates how the algorithm divides among the sub-blocks. During the first stage, each sub-block contains all of the necessary data, requiring no external communication. The second stage requires communication between pairs of sub-blocks, while all sub-bocks communicate during the final stage. The example demonstrates how a simple radix-4 64-point FFT can divide, with similar patterns allowing sub-blocks to perform larger calculations, 2^{20}-point for example.

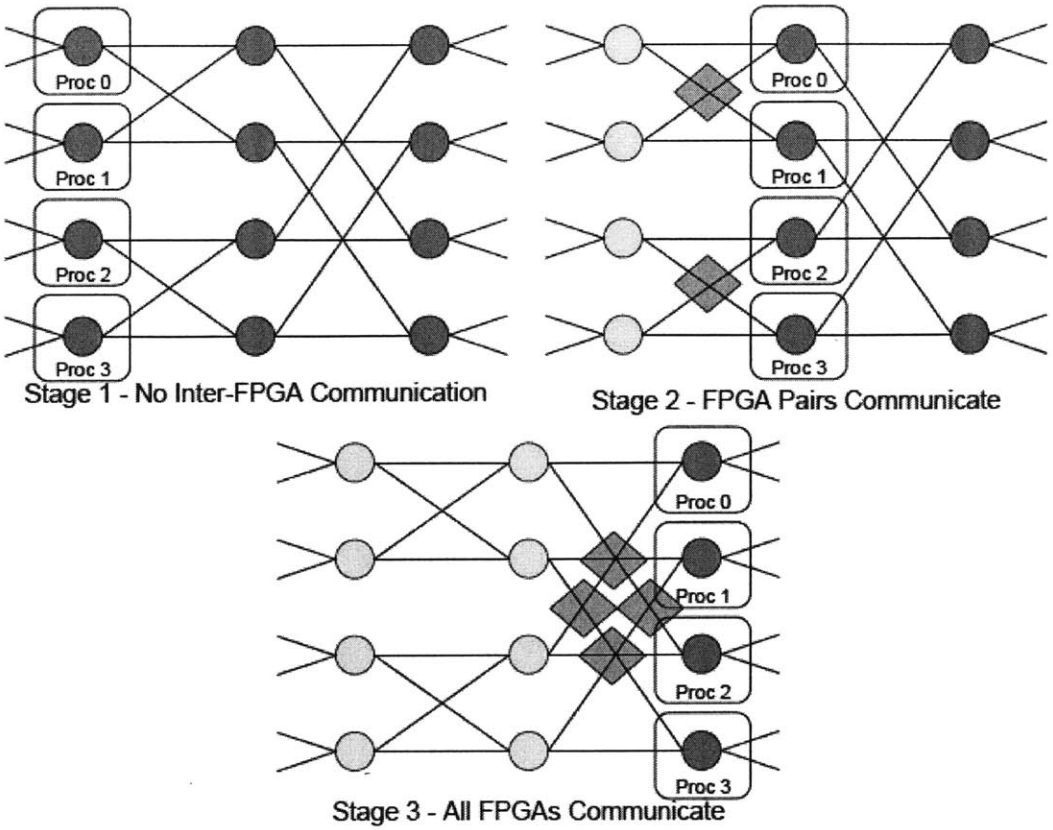

Figure 3-3: Division among 4 Sub-Blocks

3.4 Four FPGA Micro-Architectural Description

Each sub-block consists of one computational FPGAs connected to 9 DRAMs (see Figure 3-4). Two additional FPGAs supply external IO communication paths (see Figure 3-5). All FPGAs can communicate with each other through both standard and special Rocket IO paths.

3.5 Single FPGA UTL Description

Since all four FPGAs compute the algorithm equally, an UTL description of one FPGA describes all FPGAs. Given the size of the FFT, the entire algorithm and data set will not fit within this one FPGA, requiring it to contain a sub-set of the FFT hardware to perform a portion of the calculation. A UTL description defines what this calculation requires (see Figure 3-6) in the following sections.

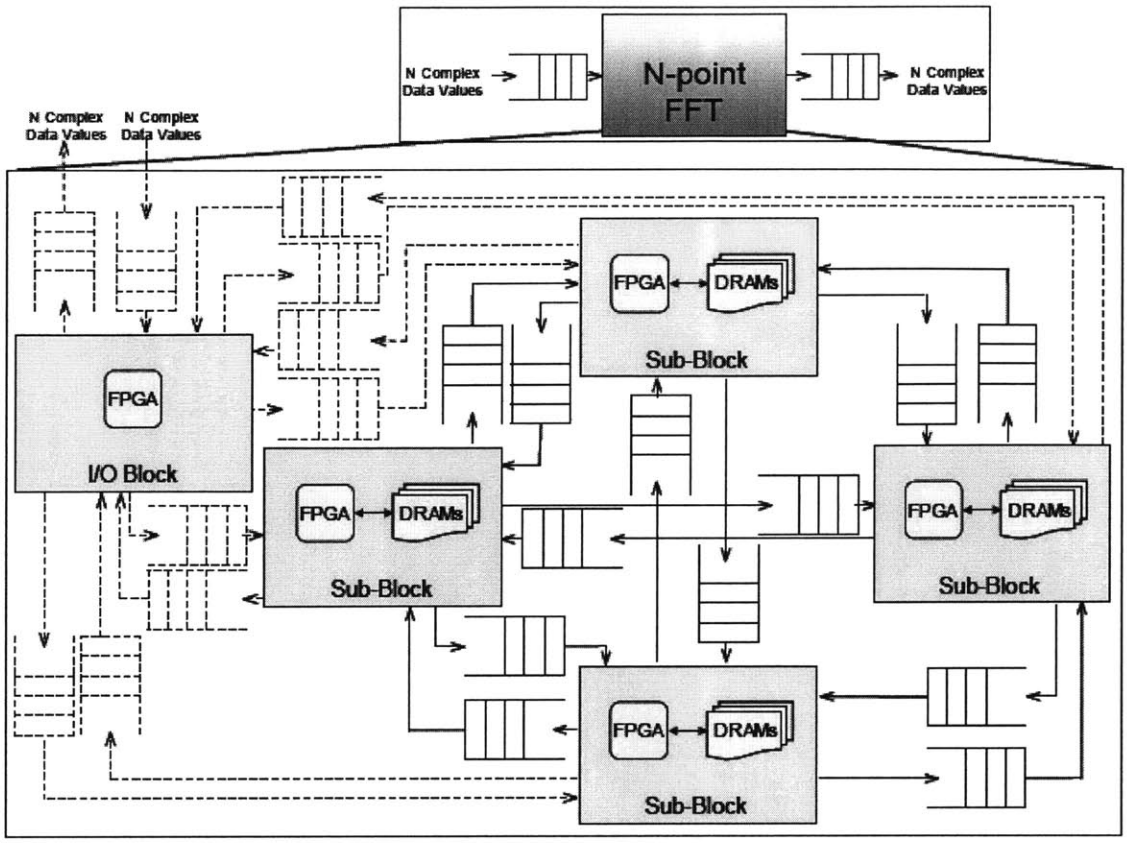

Figure 3-4: Four FPGA Micro-Architectural Block Diagram

3.5.1 Control

This module controls the flow of data and initiates transactions. It receives data and control packets from the FFT Module, and control packets from the Inter-FPGA module. Using this information as well as the current state of the calculation, the module decides where the data should be sent, what data should be requested and the overall state of the system, which it then translates into control and data packets for both the Memory and Inter-FPGA modules.

input:
- FFTControlQ{addr, current_stage, end_calc, 16 complex pairs},
- InterFPGAControlQ{op, addr, size},

output:
- ControlMemoryQ:{op, addr, size, current_stage, end_calc, x amount of data}
- ControlInterFPGAQ:{op, addr, size, x amount of data}

architectural state: Calculation State, Global Addresses

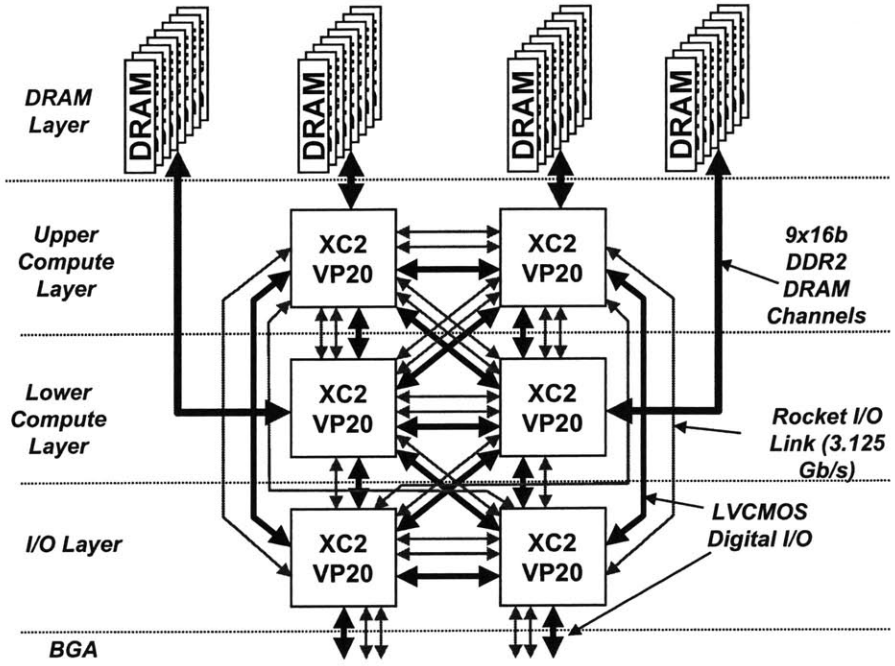

Figure 3-5: Overall Block-Level 3D FPGA System Diagram

Transactions:

1. RequestExternalData()
   (a) ControlInterFPGAQ.enq(READ, start_addr, ext_data_size);
2. SendData()
   (a) {addr, current_stage, end_calc data} = FFTControlQ.pop();
   (b) Determine data locations and split data into external_data and memory_data
   (c) ControlInterFPGAQ.enq(WRITE, start_addr, ext_data_size, external_data);
   (d) ControlMemoryQ.enq(WRITE, start_addr, memory_data_size, memory_data);
3. MemRequest()
   (a) ControlMemoryQ.enq(READ, start_addr, dram_data_size, current_stage, end_calc);
4. DetermineState()

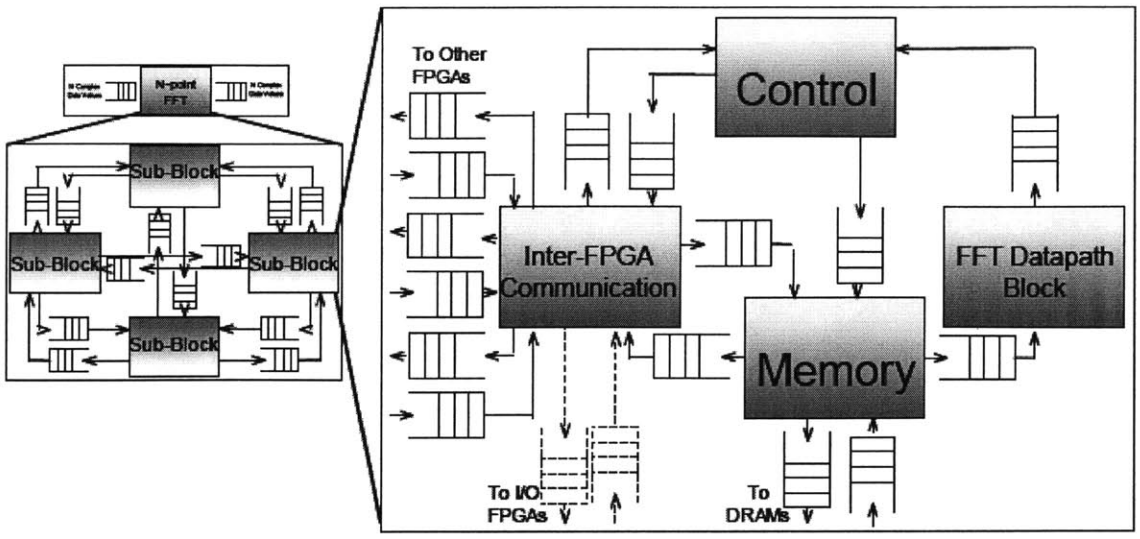

Figure 3-6: UTL Single FPGA System Block Diagram

(a) {op, addr, size} = InterFPGAControlQ.pop();

(b) Update view of mem state

Scheduler: Priority in descending order: RequestExternalData, MemRequest, DetermineState, SendData

3.5.2 FFT Datapath Block

This module performs FFT calculations when requested by the Memory module. The results of the calculation are sent along with control information to the Control module. The control information contains the address, which, while uncessary for the FFT calculation, provides enough information for the Control module to determine where the data should reside.

input: • MemoryFFTQ{current_stage, end_calc, start_addr, 16 complex pairs},

output: • FFTControlQ{start_addr, current_stage, end_calc, 16 complex pairs},

architectural state: Twiddle Multiplier Values, Twiddle Multiply State

Transactions:

1. ComputeFFT()

   (a) {current_stage, end_calc, start_addr, current_data} = MemoryFFTQ.pop();

   (b) Compute FFT

(c) if (end_calc)

i. Perform Twiddle Multiplication

(d) FFTControlQ.enq(start_addr, current_stage, end_calc, fft_data);

Scheduler: Priority in descending order: ComputeFFT

3.5.3 Memory

Memory controls both the Inter-FPGA memory and the DRAM, regulating the flow of data from Inter-FPGA and Control modules into these locations and outputting the requested blocks to either the Inter-FPGA or FFT modules as requested by the incoming control queues.

input:
- InterFPGAMemoryQ{command, addr, size, x amount of data}
- ControlMemoryQ:{op, addr, size, current_stage, end_calc, x amount of data}
- DRAMMemoryQ{addr, size, x amount of data}

output:
- MemoryFFTQ{current_stage, end_calc, start_addr, 16 complex pairs},
- MemoryInterFPGAQ{addr, size, x amount of data},
- MemoryDRAMQ{command, addr, size, x amount of data}

architectural state: DRAM State

Transactions:

1. ExternalFPGAData()

(a) {command, addr, size, data} = InterFPGAMemoryQ.pop();

(b) if (command == READ) begin

i. if (location == DRAM) begin

A. MemoryDRAMQ.enq(READ, addr, size);

ii. end else begin

A. Read Data from Memory

iii. end

iv. MemoryInterFPGAQ.enq(addr, size, data_from_memory);

(c) end else begin

i. if (location == DRAM) begin

A. MemoryDRAMQ.enq(WRITE, addr, size, data_from_memory);

ii. end else begin

A. Write data to Memory

iii. end

(d) end

2. InternalFPGAData()

   (a) {command, addr, size, current_stage, end_calc, data} = ControlMemoryQ.pop();

   (b) if (command == READ) begin

      i. if (location == DRAM) begin

         A. MemoryDRAMQ.enq(READ, addr, size);

      ii. end else begin

         A. Read Data from Memory

      iii. end

      iv. MemoryFFTQ.enq(current_stage, end_calc, addr, data_from_memory);

   (c) end else begin

      i. if (location == DRAM) begin

         A. MemoryDRAMQ.enq(WRITE, addr, size, data_from_memory);

      ii. end else begin

         A. Write data to Memory

      iii. end

   (d) end

3. DRAMData()

   (a) {addr, size, data} = DRAMMemoryQ.pop();

   (b) Write data to Memory

Scheduler: Priority in descending order: ExternalData, InternalData

3.5.4 Inter-FPGA Communication

This module communicates to other FPGAs and outside systems. It receives data from any of the Control, Memory or external Inter-FPGA data queues along with control information describing the data. It forwards this data to the appropriate module and can initiate memory transfer sequences for external Inter-FPGA modules.

input:
- ControlInterFPGAQ:{op, addr, size, x amount of data}
- OutInterFPGAQ:{command, addr, size, x amount of data}
- MemoryInterFPGAQ{addr, size, x amount of data},

output:
- InterFPGAControlQ{op, addr, size},
- InterFPGAMemoryQ{command, addr, size, x amount of data}
- InterFPGAOutQ:{command, addr, size, x amount of data}

architectural state: None

Transactions:

1. ControlRequest()
   (a) {command, addr, size, data} = ControlInterFPGAQ.pop();
   (b) if (command == READ) begin
      i. InterFPGAOutQ.enq(READ, addr, size);
   (c) end else begin
      i. InterFPGAOutQ.enq(WRITE, addr, size, data);
   (d) end
2. MemRequest()
   (a) {addr, size, data} = MemoryInterFPGAQ.pop();
   (b) InterFPGAOutQ.enq(WRITE, addr, size, data);
3. ExternalRequest()
   (a) {command, addr, size, data} = OutInterFPGAQ.pop();
   (b) if (command == READ) begin
      i. InterFPGAMemoryQ.enq(READ, addr, size);
   (c) end else begin
      i. InterFPGAMemoryQ.enq(WRITE, addr, size, data);
      ii. InterFPGAControlQ.enq(WRITE, addr, size);
   (d) end

Scheduler: Priority in descending order: ControlRequest, MemRequest, External-Request

3.6 Single FPGA Micro-Architectural Description

With an UTL single FPGA description, the micro-architectural description focuses on this same one FPGA performing a portion of the calculation, requiring control, memory, and inter-fpga communication (see Figure 3-7). Each sub-system is described below.

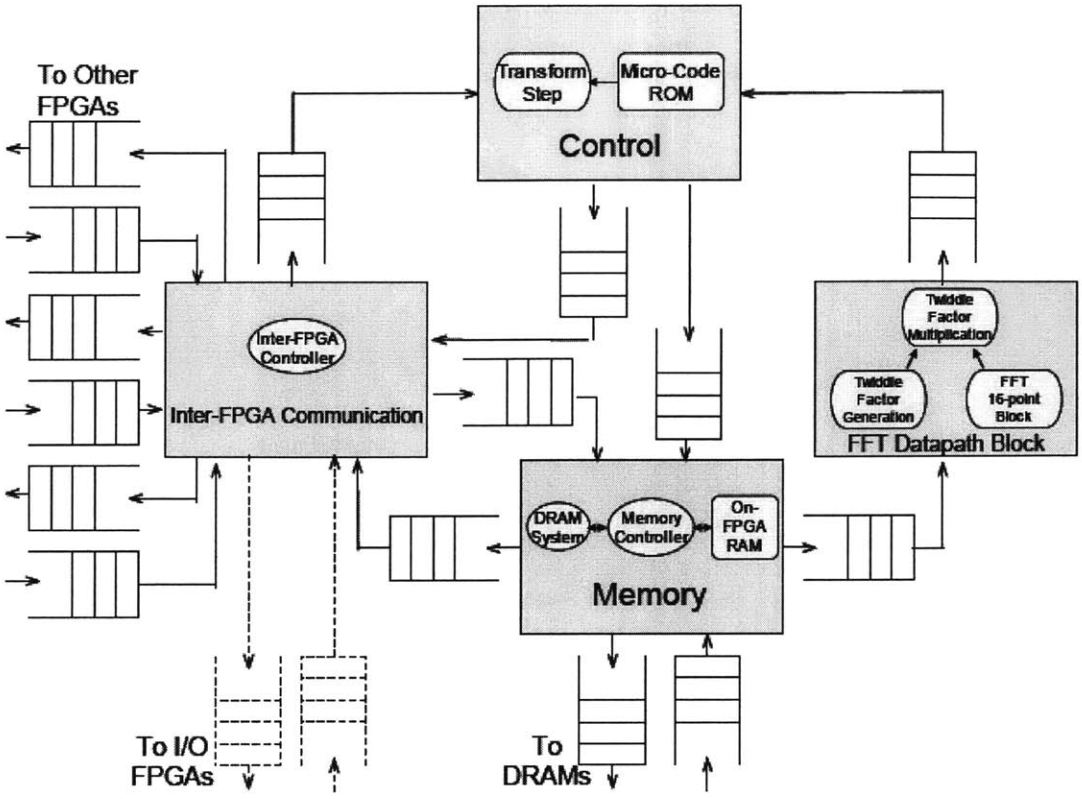

Figure 3-7: Micro-Architectural Single FPGA System Block Diagram

3.6.1 FFT Datapath Block

The FFT Datapath Block performs a Cooley-Tukey Decimation-In-Time (DIT) algorithm. This form of algorithm splits the data into subsets by even-odd patterns and performs the twiddle calculation prior to combining data inputs. As the data set is too large to perform the calculation, the Gentleman-Sande four-step approach (see Section 2.1) sub-divides the larger data set. A 16-point FFT block is the largest size that fits on the FPGA becoming the smallest factor of the 4-step algorithm. To achieve 2^{20}, the algorithm is iterated, first generating a 16x16 or 256-point FFT, then a 256x256 or 64K-point FFT, and finally a 64Kx16 or 2^{20}-point FFT, all using the fundamental 16-point block (see Figure 3-8). Within the 16-point block are 8 Radix4 Modules, with a separate module performing the multiplication step of the algorithm, including a sub-module providing the twiddle factors at each multiplication step of the calculation.

FFT 16-point Block

Combining the Radix4 modules creates the 16-point block. Each block consists of two stages of 4 Radix4 nodes each (see Figure 3-9). The inputs and outputs are

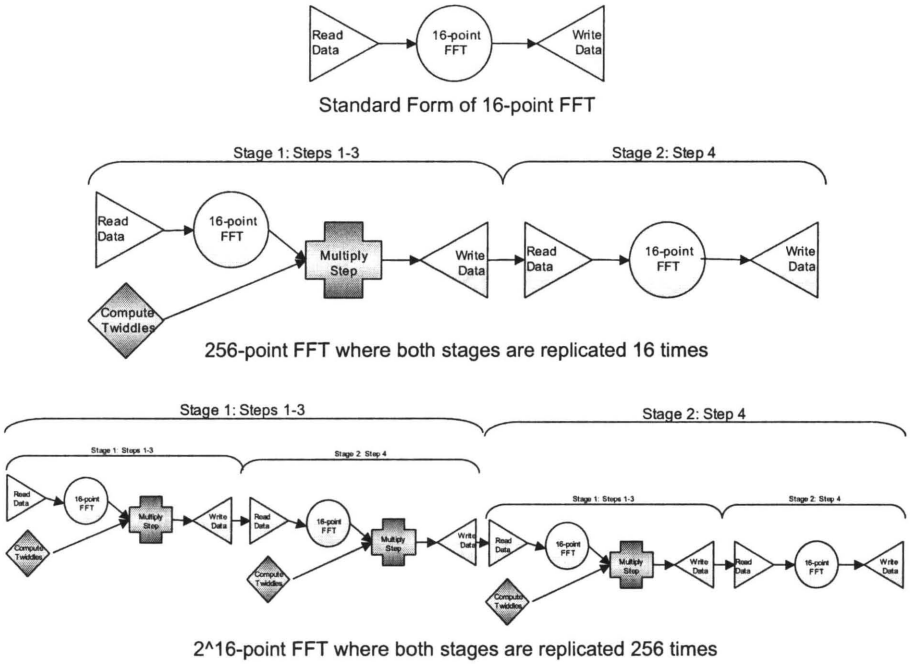

Figure 3-8: Diagram Illustrating Iterative 4-Step Approach

re-ordered, as is necessary for the algorithm, implicitly in the hardware so that the input and output to the block are in natural order and both are registered. Outputs from the first stage are directly connected to the inputs of the second stage.

Radix4 Module The Radix4 Module computes an optimized version of a 4-point FFT. This consists of 3 different stages-a twiddle multiplication and two combination stages (see Figure 3-10). Each input is factored by the twiddle values, requiring 8 multiplies and 8 additions for the stage. Each of the other two stages consists of 8 additions (subtractions consist of one inversion and an addition) resulting in a total of 32 operations per module. This results in a hardware implementation that does not meet the minimum timing requirements of the hardware system requiring registers following the multiplication stage, the most time intensive step (see Figure 3-10). The multiplication stage itself implicitly uses the on-FPGA multipliers by clearly identifying these for the synthesis tools.

Number Representation Bit width is the first design parameter for the calculation, which, for this design, matches the 18-bit width of the FPGA multipliers. In addition to this, since the calculation requires fractional values, a special number representation is required either by using fixed-point or floating-point. A fixed-point representation consists of an explicit sign bit, fixed number of bits representing a

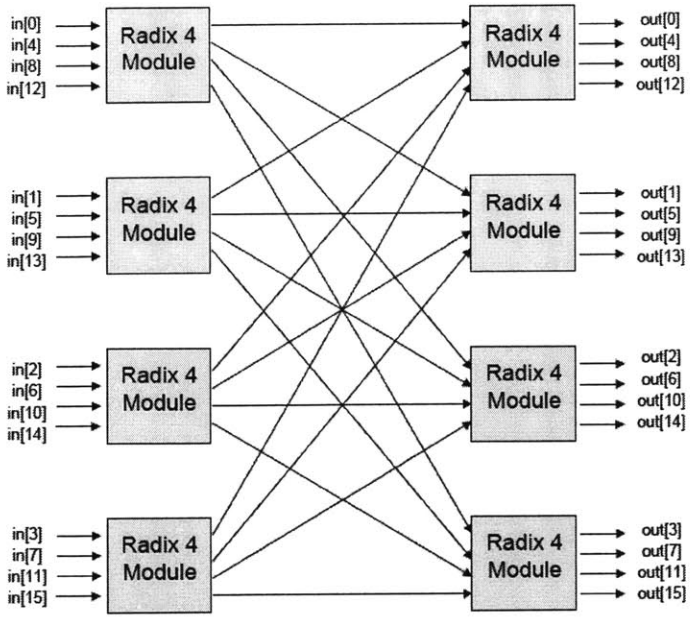

Figure 3-9: 16-point Block

range of values, implicit exponent and implicit binary point. Tracking the exponent value becomes the job of the designer. Floating-point uses an explicit sign bit, fixed number of value bits, explicit exponent and implicit binary point. Floating-point allows a larger dynamic range, but fixed-point is easier to implement and /quire less area, decreases speed and decreases power [11] and therefore represents values in this design.

Within the design, this number representation adds additional logic to each Radix4 stage. A fixed-point representation requires that the result of every calculation scale or round to require only the bit width allowed by the representation. For a multiplication, the result of 32 bits rounds at bit 14 and then masks to allow only bits $[30:15]$ to remain as the result (see Figure 3-11). A comparison determines if the special case of $-1 * -1$ exists, in which case the calculation result saturates to equal -1 requiring the value reset to the correct result of 1. As no actual scaling occurs, the exponent does not increment. Addition, on the other hand, requires that the result of 17-bits rounds at bit 0 and shift 1 left, a scaling step that increments the implicit exponent. As each number passes through three addition stages within one Radix4 step, the implicit exponent at the end of the step increments by 3. This incrementing exponent can quickly increase such that the output range does not reflect the accuracy needed, but is sufficient for lower point FFTs. To perform the larger point FFTs (2^{16} and greater), scaling needs to occur after the first step of the calculation to achieve any accuracy.

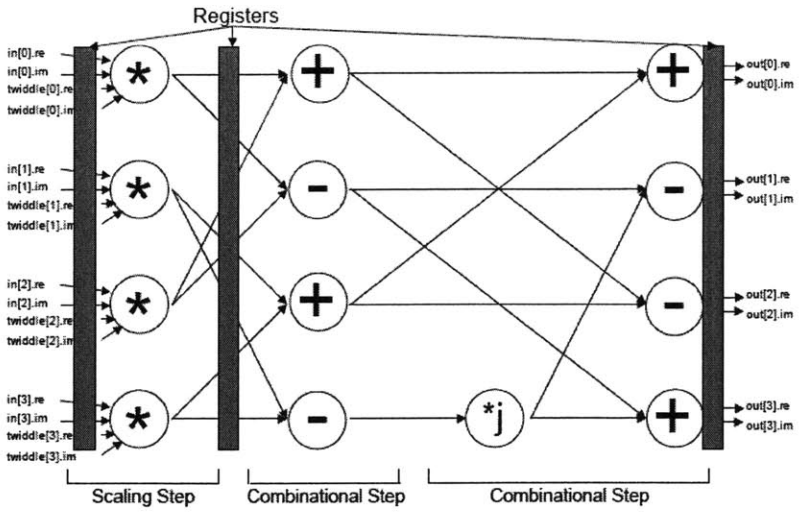

Figure 3-10: Radix4 Module with Registers. Each node represents a complex operation.

Twiddle Factor Multiplication Module

As the FFT calculation completes, each value passes through the multiplication module. If the completed calculation was the fourth step of the 4-step algorithm, the data moves through untouched. If it was the first step, the module multiplies the value by a twiddle factor calculated by the Twiddle Factor Generation Module.

Since the 4-step algorithm is iterated, three different multiplication phases exist, one for each of the 256-point, 64K-point, and 2^{20}-point iterations. This requires overlapping multiplications stages such that the fourth step of the 256-point FFT is equivalent to the first step of the 64K-point FFT and, while not normally multiplied, the result of the FFT is multiplied by the twiddle factors for the 64K-point FFT. This also occurs for the fourth step of the 64K-point FFT in relation to the 2^{20}-point FFT (see Figure 3-8).

Although the data is provided in parallel, the calculation occurs serially to reduce the hardware required. The data outputs serially as well, moving next to the Memory module.

Twiddle Factor Generation Module Twiddle Factor Generation supplies the twiddle factors necessary for each stage of the calculation. The twiddle factors follow

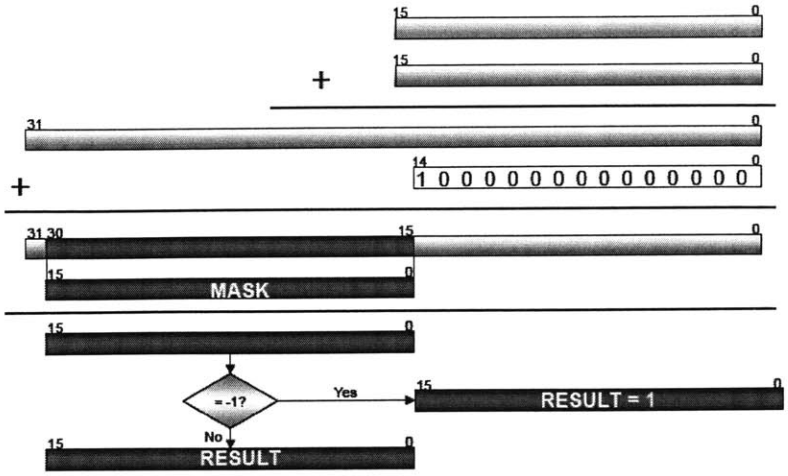

Figure 3-11: Fixed-Point Multiplication

a pattern seen in the following $[n, m]$ matrix:

$$\begin{bmatrix} W^0 & W^0 & W^0 & \dots \\ W^0 & W^1 & W^2 & \dots \\ W^0 & W^2 & W^4 & \dots \\ W^0 & W^3 & W^6 & \dots \\ W^0 & W^4 & W^8 & \dots \\ W^0 & W^5 & W^{10} & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} \tag{3.1}$$

where W represents $e^{-j2\pi/N}$ raised to $n * m$ where n is the row and m is the column.

Due to the repetition of values, the calculation stores only the second column values, relying on the first value to generate the first column and the following equation [4, 19] to generate the remaining columns:

$$W^{p*q} = W^{p*1} * W^{p*(q-1)} \tag{3.2}$$

where p is the row and q is the column of the current calculation.

This module can generate values serially, overlapping the FFT data loading cycles, reducing the hardware costs to 4 multiplies and 4 additions, and adding no additional latency to the calculation.

3.6.2 Control

The Control Module organizes the flow of data and the implementation of the algorithm. At each step, it determines which data set to use as input, which data set to read from memory, which twiddle factors to compute, where the output values should be written, what the order is for write-back of the data set to memory and what is the next stage of the calculation. It does this using a micro-code ROM and transform

step module.

Micro-code ROM

The Micro-code ROM controls the entire process, providing control signals and addresses for all other modules. It is implemented using on-FPGA Block RAMs.

Transform Step

Performing the third step of the algorithm requires translating the current $[n, m]$ FFT address location of the data to a $[m, n]$ address that may be on-FPGA memory or off-FPGA memory. The Transform Step performs this operation after the first step of the algorithm and passes the data through after the fourth step of the algorithm, writing all data to external memory in standard order.

3.6.3 Memory System

The Memory System consists of four modules: on-FPGA Memory, FPGA Memory Controller, off-FPGA DRAM, and DRAM Memory Controller; each is described briefly below.

FPGA Memory

Within the FPGA is 720Kb of RAM that can be programmed to be RAM, ROM or logic modules. Most of this acts as internal RAM for the design, holding the working input data, the next input data, the working output data, and the past output data set (see Figure 2-4). This allows space for the current computation while the Memory Controller reads the next set of input data and writes the last set of output data.

FPGA Memory Controller

This module directs the traffic from the various locations into the FPGA Memory, organizing the memory into the different sets and connecting the on-FPGA memory to the DRAM system.

DRAM

DRAM contains the entire input data set and store the output data set as well. To take advantage of the number of attached DRAMs, data stripes across the DRAMs so that 8 byte bursts access 4 inputs from each one in an interleaved manner.

DRAM Memory Controller

The DRAM Memory Controller requests data from the DRAM using a hand-shaking protocol. Design and implementation of this module are not covered in this document. See [5] for more information.

3.6.4 Inter-FPGA Communication System

The Inter-FPGA Communication system transfers data blocks between FPGAs. The communication pattern fixes the ordering such that no FPGAs interfere with the other's data transfer and a simple protocol provides the communication hand-shaking.

Chapter 4

System Model

To interpret the single FPGA results into a multi-FPGA system, a model describes the input/output (IO), area and power effects of the number of FPGAs, the dimensionality of the system, the frequency at which the system functions, and the cycles required to complete one 16-point FFT. First, the model, described in terms of IO, demonstrates the iterative quality of the 4-step algorithm, which is then elaborated into area effects and concluded with power.

4.1 IO Model

The model builds up the 2^{20}-point system, starting with a 16-point FFT and demonstrating how the 4-step algorithm iterates over the necessary values. At each step, the model describes the IO requirements in terms of on-FPGA RAMs, DRAMs and external communication.

4.1.1 16-point FFT

This model assumes that the bandwidth of the inter-communication connections exceeds the bandwidth of the DRAM communication connections.

The next key bandwidth consideration is the FFT calculation itself. The following equation represents this value:

$$B_{FFT} = f * \frac{1}{C} * (32\frac{word}{cycle} * 16\frac{bits}{word}) \tag{4.1}$$

where f is the frequency of system and C is the cycles required of 16-point FFT, decomposed into:

$$C = C_{Read} + C_{FFT} + C_{Write} \tag{4.2}$$

Figure 4-1 displays how the calculation divides into the values C_{Read}, C_{FFT}, and C_{Write}. This model then assumes that the DRAM connection limits the bandwidth of the system, although this connection can match the FFT bandwidth by increasing

the number of DRAMs used to satisfy:

$$D = \lceil \frac{B_{FFT}}{B_{DRAM}} \rceil \tag{4.3}$$

where D is the number of DRAMs connected to the system. The model assumes that D DRAMs connect to the system, leaving B_{FFT} as the dominant factor.

To better understand the connections required, Figure 4-1 demonstrates the data accesses needed for the 16-point calculation. No external FPGAs or DRAMS contribute to the data required at this stage, all data is read from one internal FPGA RAM and written into another. This block then requires

$$t_{16} = \frac{C}{f} \tag{4.4}$$

time to complete, calculation time being our metric for the IO analysis.

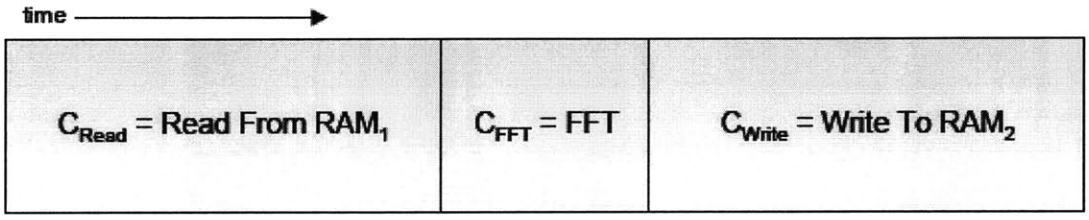

Figure 4-1: 16-point Calculation

4.1.2 256-point FFT

Figure 4-2 represents the 256-point FFT, composed of two stages where stage one consists of 16 16-point FFTs comprising the first three steps of the calculation and stage two consists of 16 16-point FFTs comprising the final step of the calculation. Again, no external FPGAs or DRAMs supply any of the data, although the access pattern for the second stage modifies to read from RAM_2 and write into RAM_2. Completion of this block requires

$$t_{256} = 2 * 16 * t_{16} + \frac{S}{f} \tag{4.5}$$

where S is the cycles required to switch between stages.

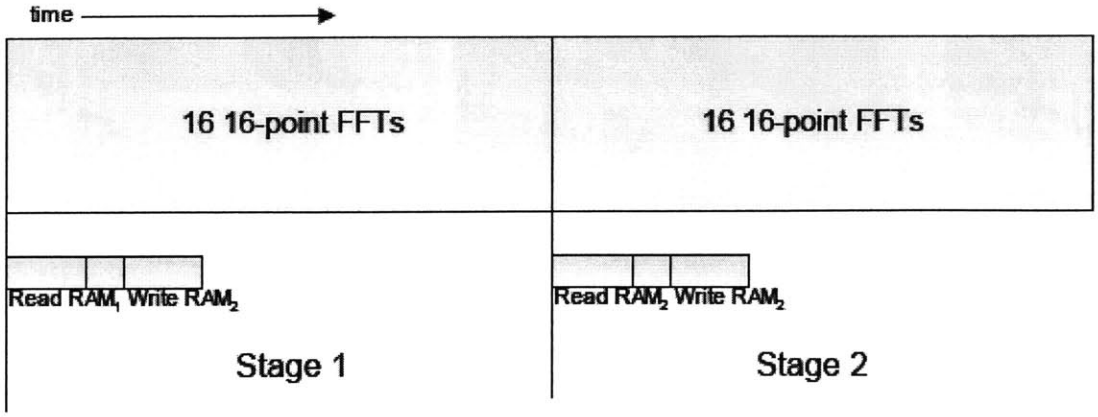

Figure 4-2: 256-point Block

4.1.3 2^{16}-point FFT

A 2^{16}-point FFT consists of two stages of 256 256-point FFTs and necessitates DRAM connections to store the data values. As Figure 4-3 shows, because the access pattern for the 256-point block modifies the RAMs used in the second stage of 16-point blocks, RAM_1 supplies all the data and RAM_2 stores all the data from the 256-point block perspective. To ensure this pattern and pipeline the DRAM accesses, two sets of D DRAMs connect to the FPGA. $DRAM_1$ loads data into RAM_1 and $DRAM_2$ stores data from RAM_2 during the first stage of the calculation, switching during the second stage to using $DRAM_2$ for both roles.

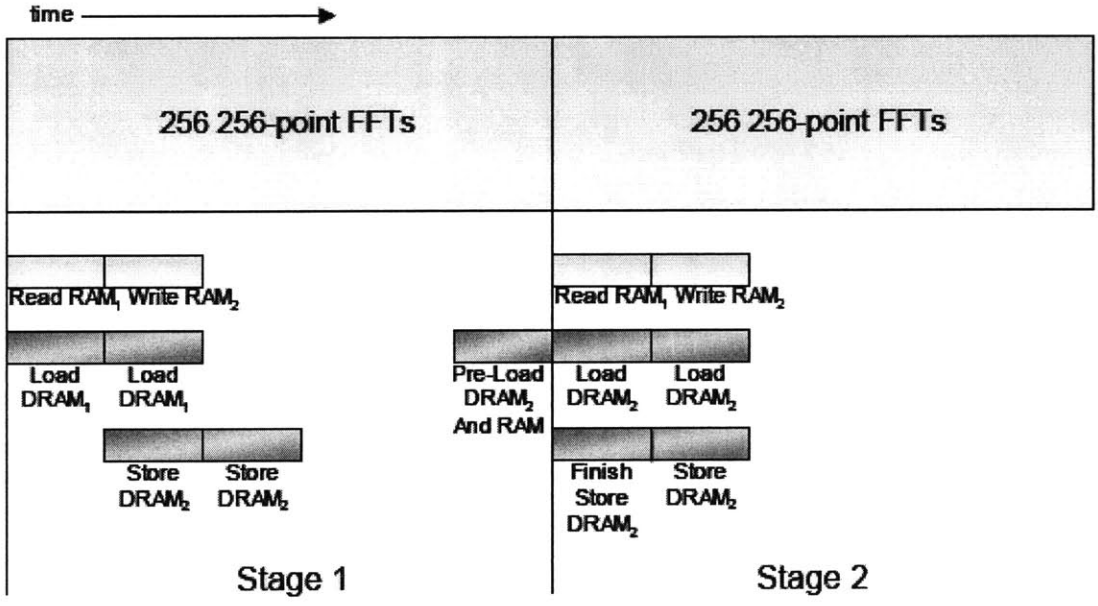

Figure 4-3: 2^{16}-point Block

Loading begins concurrent with the first 256-point block, assuming that, during the overhead of loading the data into the DRAMs, a minimum of the first 256 complex values load directly into the RAM. During the last 256-point block of the first stage, data is preloaded for the second stage from the results of this block and $DRAM_2$. Storing stalls during the first 256-point block, commencing writing the outputs during the second 256-point block and finishing with the first stage during the first 256-point block of the second stage. The results of the last 256-point block of the second stage remain in RAM_2, inserting into the output stream of the system at the correct moment. Completion of this block requires

$$t_{2^{16}} = 2 * 256 * t_{256} + \frac{S}{f} = 2 * 256 * (32 * t_{16} + \frac{S}{f}) + \frac{S}{f} \tag{4.6}$$

4.1.4 2^{20}-point FFT

While either 256-point or 2^{16} could be implemented with additional FPGAs, 2^{20}-point seems the most likely size to increase the number of FPGAs (referred to as F hereafter) in the system although, as assumed in Section 4.1.1, the bandwidth of accessing the DRAMs overshadows that of transferring data between FPGAs. Instead, the stage breakdown appears as the key variation in evaluating multi-FPGA IO. During the first stage, $\frac{16}{F}$ 2^{16}-point blocks compute, followed by $\frac{2^{16}}{F}$ 16-point calculations (see Figure 4-4). Similar to the access pattern of RAMs in the 256-point block, the first stage loads data from $DRAM_1$ and stores to $DRAM_2$, reversing the order for the second stage. This results in a total calculation time of

$$T = \frac{16}{F} * t_{2^{16}} + \frac{2^{16}}{F} * t_{16} + \frac{S}{f} = (\frac{16}{F} * 16384 + \frac{2^{16}}{F}) * t_{16} + (513 * \frac{16}{F} + 1) * \frac{S}{f} \tag{4.7}$$

an equation that is solely dependent on the number of FPGAs F, the number of cycles to switch stages S, the time to calculate one 16-point FFT t_{16}, and the frequency of the system f.

4.2 Area Model

Area increases linearly as a function of number of FPGAs, F,

$$A = F * (a_{1FPGA} + (F - 1) * a_{Inter-FPGA}) \tag{4.8}$$

where a_{1FPGA} is the area of the design on one FPGA and $a_{Inter-FPGA}$ is the area of the Inter-FPGA module. This module requests and sends data both to other FPGAs and to the DRAMs, requiring a small amount of logic and some buffers, a value estimated based on the implemented Inter-FPGA module and reasonable buffer sizes.

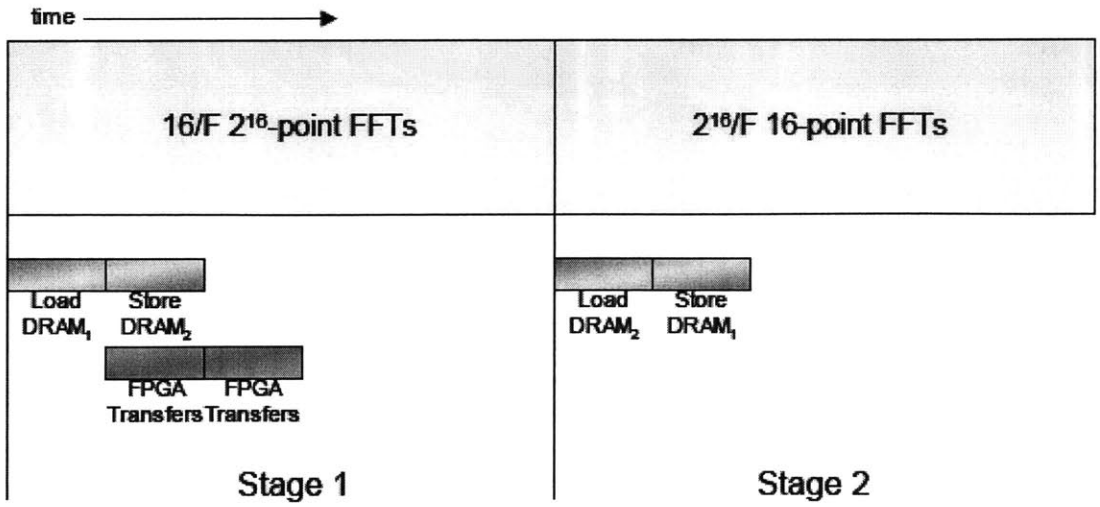

Figure 4-4: 2^{20}-point Block

4.3 Power Model

As the number of FPGAs increases, only the IO system contributes significantly to the power increase. Each FPGA connects to the same number of DRAMs and performs the same calculation, but the communication time decreases linearly with the number of FPGAs resulting in the same power usage for the calculation and the DRAM accesses within a shorter amount of time. However, the additional Inter-FPGA modules and the IO communication itself will require more power as the number of FPGAs communicating increase. Therefore, the power model focuses on the IO communication power.

If we assume that both 2D Multi-Chip Module (MCM) and 3D MCM technology use similar wire connections, the power depends on the length of the interconnect and the bandwidth at which the interconnect operates. Just focusing on one interconnection, power is:

$$pi = li * Bi \tag{4.9}$$

To determine the entire interconnect power, the number of Rocket IO compared to LVDS connections becomes critical. As long as the Inter-FPGA bandwidth is greater than or equal to the DRAM bandwidth, LVDS connections can replace Rocket IO connections. However the number of DRAMs limits the number of available LVDS connections (N_{LVDS_A}) such that

$$N_{LVDS\_A} = N_{LVDS\_Total} - D * N_{LVDS\_DRAM} \tag{4.10}$$

In order to match the bandwidth requirements, D (the number of DRAMs attached to a FPGA) LVDS connections are required. This results in a total interconnect

power of:

$$P_i = N_{Rocket} * pi_{Rocket} + D * N_{LVDS\_A} * pi_{LVDS} = N_{Rocket} * li_{Rocket} * Bi_{Rocket} + D * N_{LVDS\_A} * li_{LVDS} * Bi_{LVDS} \tag{4.11}$$

Total power then depends on the total interconnect power per FPGA (P_i) and the number of FPGAs (F):

$$P = P_i * F \tag{4.12}$$

Chapter 5

Results and Conclusions

This chapter first evaluates the model assumptions and analyzes the results of the model. Then it compares the results to other ASIC and FPGA designs followed by some conclusions.

5.1 Model Results

With the models developed and the design implemented in stages of 16-point, 256-point, 4096-point, and 2^{12}-point, we now examine the assumptions of the model and analyze the results. As with the model development, the analysis examines first the IO model, followed by area and power.

5.1.1 IO Model

The model first assumes that the DRAM communication limits the system bandwidth. As the system contains Virtex2 Pro 40 FPGAs, two methods for external communication exist, either Rocket IO or LVDS. Rocket IO operates the fastest achieving a bandwidth of 100 Gb/s for 1 module [26] although the number of modules limits the number of connections to 12 for the Virtex2 Pro 40. LVDS for this system allows 804 connections with a theoretical maximum of 840 Mb/s bandwidth although this model uses an experimentally verified value of 320 Mb/s. As the number of FPGAs in one system is unlikely to exceed 12 in the near future, this model initially describes all communication between FGPAs in terms of Rocket IO connections and uses LVDS connections for the DRAMs only. These initial IO values result in bandwidths of:

$$B_{Inter-FPGA} = 100\frac{Gb}{s} \tag{5.1}$$

$$B_{DRAM} = 320\frac{Mb}{s} \tag{5.2}$$

Therefore the DRAM bandwidth does exceed the Inter-FPGA communication bandwidth. Evaluating the FFT bandwidth requires measuring the frequency (f) at which

the 16-point block functions and the number of cycles required for one 16-point block (C). Upon implementing this block, the resulting measurements are:

$$f = 93MHz \tag{5.3}$$

$$C_{Read} = 16cycles \tag{5.4}$$

$$C_{FFT} = 7cycles \tag{5.5}$$

$$C_{Write} = 16cycles \tag{5.6}$$

$$C = C_{Read} + C_{FFT} + C_{Write} = 39cycles \tag{5.7}$$

$$B_{FFT} = f * \frac{1}{C} * 512\frac{bits}{cycle} = 1.22\frac{Gb}{s} \tag{5.8}$$

A result that also demonstrates that the DRAM limits the bandwidth and requires one set to consist of 4 DRAMs, or 8 DRAMs attached to each FPGA, which fits within the 9 DRAMs allocated per FPGA.

Before evaluating the calculation time, we need to establish the validity of using the measurements of one 16-point block for all measurements. While the IO model clearly defines the use of the number of cycles as a consistent value for each stage, the frequency can depend on the implementation of the various steps within each stage and may vary between stages, requiring anew the evaluation of the bandwidth, DRAMs connected, and 16-point block calculation time. Measurement of each implemented stage has not proven this the case and allows the usage of the 16-point block calculation time. The model also defined a parameter S, the number of cycles required to switch between stages, which measures at 2 cycles. This results in:

$$t_{16} = \frac{C}{f} = 0.42\mu s \tag{5.9}$$

$$t_{256} = 2 * 16 * t_{16} + \frac{S}{f} = 13.44\mu s \tag{5.10}$$

$$t_{2^{16}} = 2 * 256 * (32 * t_{16}) + \frac{S}{f} = 6.88ms \tag{5.11}$$

$$t_{2^{20}} = (\frac{16}{F} * 16384 + \frac{2^{16}}{F}) * t_{16} + (\frac{16}{F} + 1) * \frac{S}{f} = 34.39ms \tag{5.12}$$

where the number of FPGAs (F) is 4.

5.1.2 Area Model

Xilinx ISE Foundation supplies area measurements during the mapping process. Determining 2^{20}-point requires extending all the designs, whose linear size increase demonstrates that the design will saturate the FPGA. All other design stages use the Xilinx area numbers although determining the area of a multi-FPGA system requires the addition of the Inter-FPGA area, measured as:

$$a_{Inter-FPGA} = 68LUTs \tag{5.13}$$

resulting in a 4 FPGA area of

$$A = F * (a_{1FPGA} + (F-1) * a_{Inter-FPGA}) = 4 * (46000 + 3 * 68) \tag{5.14}$$

5.1.3 Power Model

Power reflects the difference between a 2D and 3D design therefore our result is a ratio of the total interconnect power, which depends on the power of one interconnect, a function of the length and bandwidth. From previous research [22][15] [1], the ratio of 2D interconnect length to 3D interconnect length averages to:

$$\frac{li_{3D}}{li_{2D}} = 0.80 \tag{5.15}$$

Since all other factors are equal, this translates into an average 20% total interconnect power savings or an average total power savings of 20%.

The trade-off of Rocket IO and LVDS connections affects both 2D and 3D equally in terms of power, but highlights an interesting power design point. First, given that $N_{LVDS\_DRAM} = 33$ and the maximum number of Rocket IO connections is 12, clearly the number of LVDS connections available exceeds the number necessary.

$$\frac{(N_{LVDS\_Total} - D * N_{LVDS\_DRAM})}{D} = 68 > 12 \tag{5.16}$$

This also demonstrates the relationship between the number of Rocket IO and LVDS connections as:

$$N_{LVDS\_A} = ((F+1) - N_{Rocket}) * D \tag{5.17}$$

Next, we reasonably assume that $li_{Rocket} = li_{LVDS}$ since both connections should average the same length over the various combinations of pin locations and routing paths. This then allows the power to reduce to a factor of the bandwidth and number of connections alone:

$$P = F * (N_{Rocket} * Bi_{Rocket} + D * N_{LVDS\_A} * Bi_{LVDS}) \tag{5.18}$$

Combining Equation 5.17 and Equation 5.18, varying over the number of FPGAs and Rocket IO connections, the linear relationship appears (see Figure 5-1). This demonstrates that committing all LVDS connections to both DRAM and other FPGA con-

nections allows significantly lower power while maintaining the bandwidth. Rocket IOs then can focus on communication to the external IO specialized FPGAs, allowing faster reading and writing of data between the system and external world. Alternatively, use of both LVDS and Rocket IO connections allows the number of FPGAs within the system to increase, suggesting that an internal cluster architecture may be possible with Rocket IOs connecting groups of internally LVDS connected FPGAs.

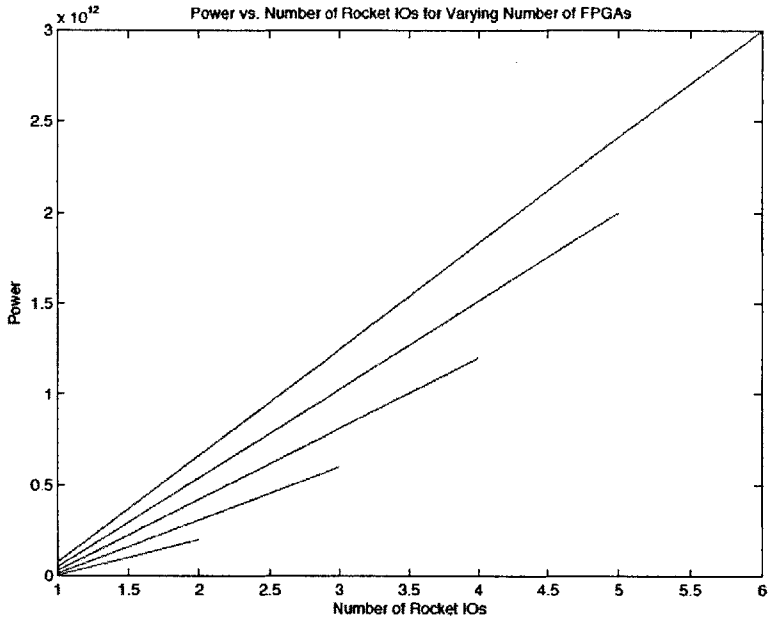

Figure 5-1: Graph Showing the Linear Power Relationship for 2-6 FGPAs

5.2 Comparison of Results to Other Systems

With the model analyzing the implementation results, we can now add to the table from Chapter 2, including each stage of the design, with single and 4 FPGA versions of 2^{20}-point design (see Table 5.1).

Designer	**FFT Points**	**ASIC Process (μm) or FPGA Type**	**Area (ASIC=mm^2, FPGA=LUTs)**	**Freq.**	**Calculation Type (μs)**
Maharatna[18]	64	0.25	13.50	20 MHz	2174.00
Cetin[8]	256	0.70	15.17	40 MHz	102.40
Baas[3] @ 3.3V	1024	0.60	24.00	173 MHz	30.00
Lenart[16]	2048	0.35	~6	76 MHz	27.00
Bidet[6]	8192	0.50	1000.00	20 MHz	400.00
Fuster[10]	64	Altera Apex	24320	33.54 MHz	1.91
Sansaloni[23] (1 CORDIC)	1024	Xilinx Virtex	626	125 MSPS	8.19
Dillon Eng.[13]	2048	Xilinx Virtex2	9510	125 MHz	4.20
RF Engines[17]	16K	Xilinx Virtex2	6292	250 MSPS*	65.53
Design	16	Xilinx Virtex2 Pro	36301	93 MHz	0.42
Design	256	Xilinx Virtex2 Pro	39079	93 MHz	13.44
Design	64K	Xilinx Virtex2 Pro	43849	93 MHz	6880.00
Design 1 FPGA	1M	Xilinx Virtex2 Pro	46000	93 MHz	137590.00
Design 4 FPGAs	1M	Xilinx Virtex2 Pro	184816	93 MHz	34390.00

Table 5.1: Comparison of Results, ASIC and FPGA Implementations
*MSPS=MegaSamplesPerSecond

In order to better understand the results, we focus on two sections: calculation time and design size. First, as Table 5.2 demonstrates, by normalizing all of the calculation times to 16-point FFTs, we see that while single FPGA and small point implementations of this design perform average to poor against the other designs, increasing the number of FPGAs increases the performance to very near the top.

Designer	FFT Points	Original Calculation Time (μs)	Normalized Calculation Time (μs)
Dillon Eng.[13]	2048	4.20	0.06
RF Engines[17]	16K	65.53	0.06
Design 4 FPGAs	1M	34390.00	0.07
Sansaloni[23]	1024	8.19	0.13
Lenart[16]	2048	27.00	0.21
Design	16	0.42	0.42
Baas[3] @ 3.3V	1024	30.00	0.46
Fuster[10]	64	1.91	0.48
Bidet[6]	8192	400.00	0.78
Design	256	13.44	0.84
Design	64K	6880.00	1.68
Design 1 FPGA	1M	137590.00	2.10
Cetin[8]	256	102.40	6.40
Maharatna[18]	64	2174.00	543.50

Table 5.2: Calculation Time Normalized to 16-Point Comparison

Table 5.3 shows the effects of gaining this speedup as the multi-FPGA system has a higher area than the single FPGA design although remains competitive with the other FPGA systems, while the other designs range across the entire spectrum. This does not show the entire picture as each design stores the values differently and may compute area differently. For this thesis, all designs stored large amounts of the data on-FPGA, containing 2 131Kb RAMs. Other system designs stored the values off-FPGA, decreasing their overall size, and not all designs stated what the size value covered, solely logic or some logic and RAMs or some other combination. However, the table does demonstrate that achieving the modularity and speedup does effect the size of the system.

Unfortunately, not enough information exists to compare power across these designs. For this design, results similar to calculation time seem likely as the power increases as the stages progress ending with almost equal power for the single compared to multi FPGA designs.

Designer	FFT Points	Original Area (LUTs)	Normalized Area (LUTs)
Design 1 FPGA	1M	46000	1
Dillon Eng.[9]	512K	12500	1
Design 4 FPGAs	1M	184816	3
RF Engines[17]	16K	6292	7
Sansaloni[23]	1024	626	10
Design	64K	43849	11
Dillon Eng.[13]	2048	9510	75
Design	256	39079	2442
Fuster[10]	64	24320	6080
Design	16	36301	36301

Table 5.3: Area Normalized to 16-Point Comparison

5.3 Conclusion

This thesis describes the design and implementation of a modular FFT algorithm, extended from a single FPGA implementation to a multi-FPGA 3-Dimensional model. With this extension, the benefit of modularity reveals a trade-off of calculation time, size, and power. Clearly, with 3D architectures, modularity also allows scalability, a feature useful with a new technology that may exhibit initial yield issues and later grow to an unexpected number of FPGAs.

While these results would likely hold for similar highly structured and repetitious algorithms, future work could explore more varied algorithms and applications with little apparent modularity to determine the feasibility of 3D benefits.

Bibliography

[1] C. Ababei, P. Maidee, and K. Bazargan. Exploring potential benefits of 3D FPGA integration. In *Field-Programmable Logic and its Applications*, pages 874–880, 2004.

[2] Syed M. Alam, Donald E. Troxel, and Carl V. Thompson. Layout-specific circuit evaluation in 3-D integrated circuits. *Analog Integrated Circuits and Signal Processing*, 35:199–206, 2003.

[3] Bevan M. Baas. A low-power, high-performance, 1024-point FFT processor. *IEEE Journal of Solid-State Circuits*, 34(3):380–387, March 1999.

[4] David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4(1):23–35, March 1990.

[5] Vimal Bhalodia. *DDR Controller Design Document.*

[6] E. Bidet, D. Castelain, C. Joanblanq, and P. Senn. A fast single-chip implementation of 8192 complex point FFT. *IEEE Journal of Solid-State Circuits*, 30(3):300–305, March 1995.

[7] Georg Bruun. z-transform DFT filters and FFTs. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26(1):56–63, 1978.

[8] Ediz Cetin, Richard C. S. Morling, and Izzet Kale. An integrated 256-point complex FFT processor for real-time spectrum analysis and measurement. In *Proceedings of IEEE Instrumentation and Measurement Technology Conference*, pages 96–101, May 1997.

[9] Tom Dillon. An efficient architecture for ultra long FFTs in FPGAs and ASICs. In *Proceedings of High Performance Embedded Computing*, September 2004. PowerPoint Poster Presentation.

[10] Joel J. Fuster and Karl S. Gugel. Pipelined 64-point fast fourier transform for programmable logic devices.

[11] Altaf Abdul Gaffar, Oskar Mencer, Wayne Luk, and Peter Y.K. Cheung. Unifying bit-width optimizations for fixed-point and floating-point designs. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, volume 12, pages 79–88, April 2004.

[12] W. M. Gentleman and G. Sande. Fast fourier transforms - for fun and profit. In *Proceedings of American Federation of Information Processing Societies*, volume 29, pages 563–578, November 1966.

[13] Dillon Engineering Inc. Ultra high-performance FFT/IFFT IP core. http://www.dilloneng.com/documents/fft_spec.pdf.

[14] T. Isshiki and Wei-Mink Dai. Field-programmable multi-chip module (FPMCM) for high-performance DSP accelerator. In *Proceedings of IEEE Asia-Pacific Conference on Circuits and Systems*, pages 139–144, December 1994.

[15] Miriam Leeser, Waleed M. Meleis, Mankuan M. Vai, and Paul Zavracky. Rothko: A three dimensional FPGA architecture, its fabrication, and design tools. *IEEE Design & Test*, 15(1):16–23, January 1998.

[16] Thomas Lenart and Viktor Owall. An 2048 complex point FFT processor using a novel data scaling approach. In *Proceedings of IEEE International Sypmposium on Circuits and Systems*, pages 45–48, May 2003.

[17] RF Engines Limited. The vectis range of pipelined FFT cores 8 to 64k points. http://www.rfel.com/products/Products_FFTs.asp.

[18] Koushik Maharatna, Eckhard Grass, and Ulrigh Jagdhold. A 64-point fourier transform chip for high-speed wireless LAN application using OFDM. *IEEE Journal of Solid-State Circuits*, 39(3):484–493, March 2004.

[19] Pronita Mehrotra, Vikram Rao, Tom Conte, and Paul D. Franzon. Optimal chip-package codesign for high performance DSP.

[20] Se Ho Park, Dong Hwan Kim, Dong Seog Han, Kyu Seon Lee, Sang Jin Park, and Jun Rim Choi. Sequential design of a 8192 complex point FFT in OFDM receiver. In *Proceedings of IEEE Asia Pacific Conference on ASIC*, pages 262–265, August 1999.

[21] C.M. Rader and N.M. Brenner. A new principle for fast fourier transformation. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 24:264–266, 1976.

[22] Arifur Rahman, Shamik Das, Anantha P. Chandrakasan, and Rafael Reif. Wiring requirement and three-dimensional integration technology for field programmable gate arrays. In *IEEE Transactions on Very Large Sale Integration (VLSI) Systems*, volume 11, pages 44–54, February 2003.

[23] T. Sansaloni, A. Perez-Pascual, and J. Vallsd. Area-efficient FPGA-based FFT processor. *Electronics Letters*, 39(19):1369–1370, September 2003.

[24] Wikipedia. Fast fourier transform. http://en.wikipedia.org/wiki/Fast_Fourier_transform.

[25] S. Winograd. On computing the discret fourier transform. *Mathematics of Computation*, 33:175–199, January 1978.

[26] Xilinx Corporation. *Virtex-II Pro Data Sheet*, 2004.