ABSTRACT

HAMLETT, MATTHEW. A Scalable Architecture for Hardware Acceleration of Large Sparse Matrix Calculations. (Under the direction of Dr. Paul Franzon).

The task of implementing the Jacobi method has been looked at from several research works over the years. The Jacobi method is considered the most ideal Iterative method for implementation on FPGAs because of its inherent parallelism and lack of data dependencies. In this work, we look specifically at solving very large matrix equations in the form of Ax = b. Here A is a sparse matrix with dimensions of 1 million x 1 million with 6 entries per row. X is the vector we are solving for, and b is a known vector. All data is in 64-bit IEEE-754 floating point format.

Previous work in this area has implemented the Jacobi method using only on chip memory accesses, greatly limiting the size of the matricies that can be solved. By using external memory, we present a design that is practical and can be used to accelerate various engineering and scientific problems today. In this design, we also implement the resources necessary for Multiple FPGAs to be used in a distributive manner so as to tackle larger problems. Our design gives a peak floating point performance of 1.8 GFLOPS and a sustained floating point performance of 1.18 GFLOPS. This is a speed up factor of around 2.95 when compared to the sustained performance that is typically seen on today's general purpose computers with this type of problem. To obtain this high peak floating point performance, we present in this paper a group of memory interfaces that are capable of supplying a total data rate of 20 Gb/sec sustained.

A SCALABLE ARCHITECTURE FOR HARDWARE ACCELERATION OF LARGE SPARSE MATRIX CALCULATIONS

by
MATTHEW HAMLETT

A thesis submitted to the Graduate Faculty of
North Carolina State University in partial
fulfillment of the requirements for the
Degree of Master of Science

COMPUTER ENGINEERING

Raleigh, North Carolina

2006

APPROVED BY:

_____

Dr. Paul Franzon
Chair of Advisory Committee

_____ _____

Dr. Michael Steer *Dr. Gianluca Lazzi*

BIOGRAPHY

Matthew Hamlett was born in 1982 in Charlotte, North Carolina (US). He was in the International Bacculaureate program until 2000 when he moved to Raleigh, North Carolina to attend North Carolina State University. He graduated with degrees in Electrical Engineering and Computer Engineering in 2000.

He stayed at North Carolina State University to pursue the Master of Science program under the guidance of Dr. Paul Franzon. He worked on his MS thesis between Fall 2005 and Fall 2006. During the summers of 2004, 2005 & 2006 he interned as a control systems engineer at Talecris Biotherapeutics.

ACKNOWLEDGEMENTS

There are many people who have provided me with encouragement and support throughout this research process. I would like to take this time to acknowledge these contributions:

I am very appreciative of the love and support of my parents, Mike and Debbie as well as my sister Kim, and brother Tyler. Their encouragement and support has helped keep me motivated through my studies and research here.

I am very appreciative of the support of my advisor, Dr. Paul Franzon who provided me the opportunity to start my thesis in this area. The research he and his team work on is a great source of inspiration in my studies and motivates me to always want to learn more. Dr. Franzon has always been willing to answer my many questions, and has provided me with many opportunities to succeed and follow my own path with this research.

I would like to thank Chris Amsink and Meeta Yadav for their help in understanding how the DDR2 interface should work and for providing me access to an interface that helped me understand things better. I would like to thank Randy Barlow for his willingness to help me understand the structure of the Matricies used in his research under Dr. Lazzi. I thank my committee members, Dr. Gianluca Lazzi and Dr. Michael Steer for taking time out of their busy schedules to listen to my research and for being supportive and encouraging with my topic.

I would like to thank Rudolf Usselmann for providing the USB core included in this design through the open source initiative opencores.org. Lastly, I would like to thank Xilinx for providing the software and the proprietary floating-point and memory cores that have allowed me to actually complete this research in a normal time span.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Description of Problem to be solved

Large matrix multiplications are found through many scientific and engineering applications. These calculations can take many forms, some of which include: Multiplying two large rectangular matrices together (Figure 1), Multiplying a Large Matrix by a Vector (Figure 2), and solving for a Large Matrix multiplied by an unknown vector equal to another vector (Figure 3). The focus of this research work is in solving Large matrix systems that take the form in Figure 3.

Engineers and Scientists mainly use software based solvers to tackle the three types of equations presented above. As engineering and science advance, the size of the matrices to be solved increases greatly. Matrices with millions of rows and columns are beginning to become common place. These computations can take days or weeks on even the most powerful of computers.

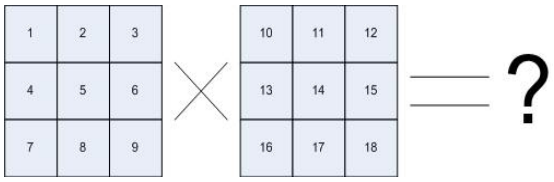


Figure 1: Matrix type AB=x

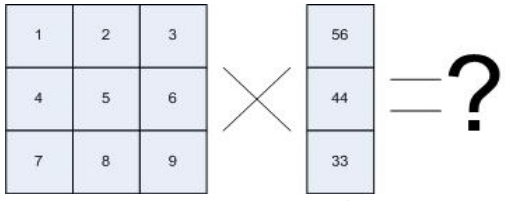


Figure 2 Matrix type Ab =x

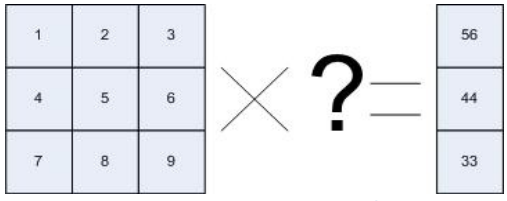


Figure 3: Matrix type Ax=b

The reason why even today's most powerful computers cannot quickly solve these equations is due to the general purpose nature of a computer. While clock frequencies are climbing to 3 and 4 Ghz, each cycle

the processor can only perform a limited amount of work. The processor may only have one or two Floating point multipliers that can be used each cycle. Another limitation of today's architectures is with the memory bandwidth available to the computer. While the processor may be able to operate at 3 Ghz, the memory can only operate at a fraction of that speed, and even then it may take multiple cycles on the memory side of things to access the needed data. All of these are reasons why solving very large matrix calculations on general purpose computers can take a very long time.

Field Programmable Gate Arrays (FPGA) are devices that can be specially configured to perform a variety of tasks. The advantage of FPGAs is that they can be quickly and easily configured multiple times to do different things. They have a quicker time to market and much cheaper initial cost than custom designed ASIC solutions. FPGA technology has been rapidly improving over the recent years and today you can find devices that can operate up to 600 Mhz and can contain the equivalent of millions of gates of logic.

The idea of using FPGAs to quickly solve matrix calculations has been a topic of a lot of research over the years. Journal articles have presented FPGA solutions to solve all three types of equations presented above with very good speed improvements over the general purpose CPU. The reason for focusing my research on solving the 3rd type of matrix equation (Figure 3) is two-fold. First, solving this type of matrix equation has been the focus of less research than the other two types, providing more of an opportunity to look at new aspects of it. The second reason is that current research efforts at NCSU require calculating the solution to matrices in this format. Software based approaches are currently being used to solve these equations, but these solutions can take days to derive. The goal of this research is to determine if a custom FPGA solution can be designed that can handle large matrices in the format Ax = b. Here, A is your square matrix, b is a known vector and x is the vector you are solving for. We focus on using an iterative method known as the Jacobi Method to solve these equations.

1.2 Related Work

Previous work on FPGA based Jacobi Iterative Solvers has shown the possibility of very good speed improvements over the general purpose CPU. In Related Work, (1), an architecture was presented for solving both dense and sparse matrix equations using the Jacobi Method. This architecture serves as the basis of our research. One limiting factor with the work was its sole use of on chip memory. This limits the size of the matrix that can be solved greatly as most FPGAs will only have somewhere between 10 and 30 Mb of integrated memory.

Another research paper by (2) looks at using the Jacobi Iterative method on FPGAs to accelerate large power calculations. This work uses a similar approach to (1) with very promising results. This research specifically analysis how the inherent parallel nature of the Jacobi method makes it the most viable for implementation on the FPGA. It argues that the performance benefits obtained with the pipelined Jacobi method on the FPGA hardware could easily overtake the more common Gauss-Seidel or Newton-Raphson methods. Both of these methods do not have the ability to take advantage of the parallel FPGA nature that the Jacobi method does.

In the work (3), the tradeoff between the size of the processing units, their pipeline depth and maximum clock frequency was analyzed. As was true in our research, the division unit in their design required a large number of resources. They looked at the trade off of using a smaller division unit with being able to fit more processing units on a single chip.

In work unrelated to FPGAs, but very related to Sparse Matrix Calculations (4), we are able to get insight into the true performance of software solutions on a variety of CPUs. In this work, Nanri takes a variety of modern, high performance CPUs such as the Itanium2, Xeon, Opteron, SPARC64, and runs software based mathematics to analyze the true Floating Point Performance being seen. Interesting enough, the result

seems to depend on the number of non-zero elements per row (from 10 to 5000). The more elements per row, the higher the sustained performance seen. In order to provide a fair comparison to our work, we use the data for sparse-matrix solutions with 10 entries per row. For the Xeon and Opteron that equates to about 370 MFLOPS, the Itanium2 is lower than that. In comparison, the peak floating point performance of the Xeon, Opteron and Itanium are 6.1 GFLOPS, 4.4 GFLOPS, and 6 GFLOPS respectively. This means that we are seeing less than 10% of the peak floating point performance in real life.

1.3 Thesis Contribution

This thesis makes the following contributions to current work on FPGA implementations of the Jacobi Iterative Method. To the best of our knowledge, no other current research work has looked at these areas.

1. We present a memory interface design which allows us to satisfy the memory bandwidth needs of the main computational unit, while allowing us to use external memory. This allows us to maintain and improve upon the performance levels of previous research while at the same time being able to use our design for much bigger matrix problems. (Our design can handle an A matrix with dimensions just over 1 million x 1 million).
2. We develop a module which computes the error of the current matrix iteration and compares it to a desired error. The controller is able to use the result of this to determine when to stop the iteration sequence.
3. We present a method for communication between multiple FPGA devices on a common system board. This allows us to tackle larger matrix calculations by simply adding more FPGA devices to this common bus interface.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 explains the Jacobi method and how it can be used iteratively to solve our matrices. Chapter 3 looks at the limitations of current FPGAs and the needs for our design. It then looks at which FPGAs and components provide the best match for us. Chapter 4 explains the implementation details of the various components of our design. Here we present a description of each major functional block and illustrations where appropriate. Chapter 5 examines the timing, bottlenecks and overall performance of our design. No tests or code are included in the main body of this paper. Waveforms and simulations results are included in the appendix for those interested in more detail on the design.

2 Solutions to Problem

2.1 Introduction to Iterative Methods

Before we begin describing the Jacobi Method, it is important to understand why there is a need to use an iterative process to solve our matrix problem. In the book *Iterative Solution of Large Sparse Systems of Equations* (5), it is mentioned that Mathematical Iterative methods go back 170 years. The first person known to use them was Carl Friedrich Gauss who derived the method because using the least squared method resulted in a system of equations much too large to use direct Gauss elimination ((5). If you take a large sparse matrix problem that is of the form Ax =B and try to find a solution directly, you quickly run into the problem of only having a limited amount of memory. This is because a sparse matrix can be stored in such a way that only non-zero values have to remain in memory. For a double precision matrix on the order of 1 million x 1 million, this means that you only need a few gigabytes of memory to hold the problem versus the unrealistic terabytes that would be needed to hold the entire matrix. When using a direct method to solve these problems you would typically take the inverse of the square matrix A, but the result of doing so is a new, non-sparse matrix on the order of 1 million x 1 million that cannot fit in main memory. Thus, the need for the iterative method arises as a way of doing the calculation without losing the sparsity of the matrix.

One may ask exactly what an Iterative method is. In General, an iterative method works by making an initial guess as to the answer of the problem. This guess is plugged back into the problem and the result with the guess is calculated. This result is compared to the desired result and an error is calculated. This error is then used to make another guess and the process then iteratively repeats until the error drops below a desired value.

For the application area of matrices, this process works as follows. You first start off with a problem to solve that resembles this:

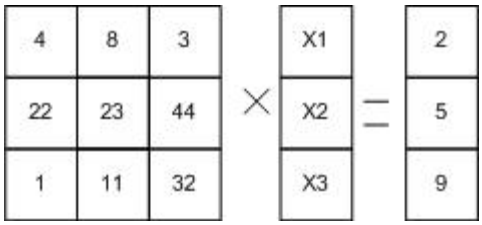


Figure 4: Initial Matrix

Step One involves coming up with a guess for the values of X1, X2 & X3. Often times solvers just use zeros for the guesses, which we will do here. Research has shown however that if you can fill in a more educated guess based on the matrix, then you can greatly reduce the number of iterations required to reach your result. As shown below we fill in zero for X1, X2 & X3 and then proceed with our calculation.

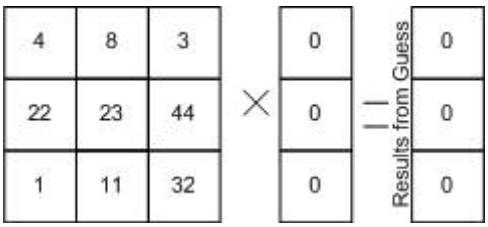


Figure 5: Result of Guess

The results from this iteration are then compared to the values that are known to be correction (2, 5 & 9). Some process is then used to come up with the best guess for the next iteration, assuming that the current iterations error is still outside of the desired bounds.

2.2 Understanding the Jacobi Method

While there are many different iterative methods available to solve the problem at hand, the Jacobi method has been the focus of many FPGA implementations because it can best take advantage of the parallel nature of the FPGA hardware. Each row in the matrix can be calculated independently of the results of the other rows leading to something that can be easily split up among multiple computation units or across multiple FPGAs. The easiest way to explain the Jacobi method is to describe the process from the viewpoint of a group of algebraic equations to solve. All matrices in the format we deal with can also be thought of as a system of equations (Figure 6: Matrix as Algebraic Equations).

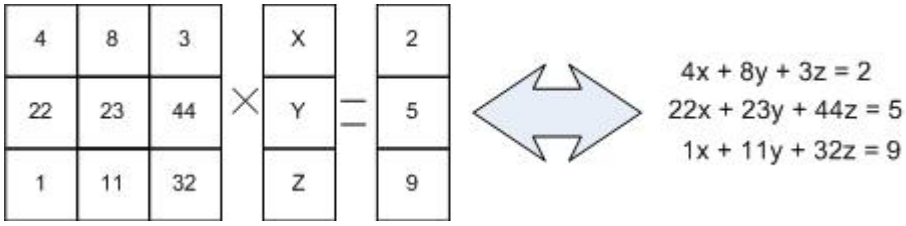


Figure 6: Matrix as Algebraic Equations

The Jacobi Algorithm transforms the above equation into the following format:

$$\begin{aligned} 4x^{n+1} + 8y^{n} + 3z^{n} &= 22 \\ 2x^{n} + 23y^{n+1} + 44z^{n} &= 5 \\ 1x^{n} + 11y^{n} + 32z^{n+1} &= 9 \end{aligned}$$

Equation 1

The meaning of the above equations is basically that for every I'th row in the matrix A, compute the corresponding I'th value in the matrix X using all the other values of vector X from the previous iteration. The superscript n represents the value from the previous iteration, while n+1 represents the value trying to be found. For our initial solution, the values of x =0, y=0 & z=0 are plugged in to find the guess for the next iteration.

The equations on the previous page can be transformed into an easier to use form as shown below:

$$4x^{n+1} + 8y^n + 3z^n = 2 \iff x^{n+1} = \frac{1}{4}(2 - 8y^n - 3z^n)$$

$$22x^n + 23y^{n+1} + 44z^n = 5 \iff y^{n+1} = \frac{1}{23}(5 - 22x^n - 44z^n)$$

$$1x^n + 11y^n + 32z^{n+1} = 9 \iff z^{n+1} = \frac{1}{32}(9 - 1x^n - 11y^n)$$

Equation 2

By filling in our initial values of 0 for x, y & z, we arrive at new values to be used for the next iteration. These values are x = ½, y=5/23, z=9/32. The transformation above can be more generically represented with the following equation (6):

$$\tilde{x}_i = \frac{1}{A_{ii}}(b_i - \sum_{j=1}^{j=N} Q_{ij}x_j)$$

Equation 3

In addition, the need arises to know when you can stop iterating the above equation. Typically an acceptable error rate is defined and once you are under that you would stop iterating and have your answer. The equation to compute the error is shown in Equation 4 and is computed at the end of every iteration, not every row computation ((6):

$$err = \frac{\max\limits_{i=1,\ldots,N}(|x_i - \tilde{x}_i|)}{\max\limits_{i=1,\ldots,N}(|b_i|)}$$

Equation 4

If the error rate as calculated above is still over the desired error rate, then you must update all x values with the values just calculated and use them for the next iteration. Other criteria could be used to determine when to stop iterating. The use of error is not something that is required with the Jacobi method. Other implementations of Iterative solutions have set a fixed number of iterations to run before stopping and displaying the results. Yet other users have implemented the use of time, so that a certain time after starting, the Iterative process stops and displays the results. We choose to implement the error calculator in our design because it is the most scientific way of determining when you get a good result.

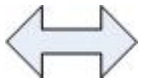
2.3 Introduction to Sparse-Matrix Storage Format

The target application for the work presented in this document is Sparse Matrices. Sparse Matrices with dimensions on the range of 100 thousand to 10s of millions are routinely encountered in a variety of scientific and engineering applications. This work also focuses on double precision floating point arithmetic where each value requires 64-bits to store it. The memory that would be required to store all values of an A matrix of size 100,000 x 100,000 in double precision format would be around 640 GB. To store a matrix of 1 million x 1 million would require 64 TB (Terabytes) of memory. Clearly it is unfeasible to use ordinary means to store these matrices. We can take advantage of the sparsity of the matrices where the majority of the entries are zeros. For this work specifically, each row in the matrix always has 6 values, no more and no less. This is because the in house work we are targeting this research towards has one entry for each geometric dimension per row. If we only store the non-zero elements, a matrix of this type with dimensions of 100,000 x 100,000 would only require 38 Mb of memory to store. Similarly, the 1 million x 1 million matrix would only require 380 Mb of memory….easily feasible with today's memory technology.

One might ask how you can store only the non-zero entries of a matrix without loosing the structure of the matrix. A format known as Compressed Sparse Row Format (CSR) was developed just for this purpose. CSR uses three vectors known as val, col and ptr to store only the non-zero values of a matrix while still retaining the original structure. The purpose of the val vector is to store the actual non-zero data value. For purposes of this work, each entry in the val vector is 64 bits and is in IEEE-754 floating point format. The col vector tells the column index of each non-zero value. This is an integer number and is 32-bits for purposes of this project. That allows for matrix dimensions of over 4 billion. There are an equal number of col and val entries, i.e. the val and col vectors are the same length. The final vector used in CSR is the ptr vector which tells the location in val where a new matrix row starts. For purposes of this research we do not need the ptr vector and can simplify the design and save memory usage by not using it. The reason for this is because of

the special nature of the matrix we are solving where every row in the A matrix has exactly 6 entries. Because of this, every 6th entry in the val vector is the start of a new row. Below we show a small sparse matrix and how it is decomposed into its 3 vectors in the CSR format.

$$A = \begin{vmatrix} 4 & 0 & 9 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 3 & 3 & 0 \\ 0 & 0 & 0 & 22 \end{vmatrix}$$



I	**1**	**2**	**3**	**4**	**5**	**6**
val	4	9	4	3	3	22
col	1	3	2	2	3	4
ptr	1	3	4	6	7	

Figure 7: CSR Format

2.4 Data Storage Format Specific to this Design

One of the major focuses of this project is on getting data into and out of external memory with the FPGA. External memory is needed in order for us to be able to handle matrices with dimensions over 10,000. Due to the nature of the Jacobi algorithm, part of the data needed for the algorithm is sequential (you read the data in order) . The other part of the algorithm needs to pull non-sequential, random numbers from memory. While the reasoning will be explained in Chapter 4, for now we will just say that the sequential data is kept separate from the random data. We also store the sequential data into memory in a special format to ease the complexity of our design. The 64-bit val entries, the 32-bit col entries and the 64-bit entries of vector B are the data points that are accessed sequentially in this design. Because of this, we store the values associated with each row in 640 bit blocks of the following format:

Table **1***: Sequential Data Format*

A1	A2	A3	A4	A5	A6	Col1	Col2	Col3	Col4	Col5	Col6	B
64-bit	64-bit	64-bit	64-bit	64-bit	64-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	64-bit
639:576	*575:512*	*511:448*	*447:384*	*383:320*	*319:256*	*255:224*	*223:192*	*191:160*	*159:128*	*127:96*	*95:64*	*63:0*

The non-sequential data (the x vector), is stored in another external memory in the order it is in the vector. This allows us to easily pull entry x_p by simply going to location P and pulling the 64-bit data value stored there.

3 Design Limitation Analysis

3.1 FPGA Design Limitations/Factors

FPGA devices have limited resources available to be configured. Unlike ASIC designs where you can just about make a design as complex as you want, an FPGA can only be made as complex as what will fit on the chip. The key resource constraints encountered throughout this project are maximum clock frequency, number of slices available, number of multiply or DSP devices available, and pin count. This section looks at these resource constraints, how they limited our design, and how based on these constraints we choose the specific FPGA that we used.

The two most constraining resources in this design were Pin count and # of available slices. Several portions of this design (the divider, memory controllers) use up a large amount of the slices available. After adding in the sequential memory controller we found we did not have enough slices to fit on the XC4VLX60 device and had to move to the VLX80. Another resource constraint we ran into at the same time was that the limited pin resources available in the 10FF668 package were not enough to allow us to interface with the 12 different memory components used in this design. By upgrading to the 10FF1148C package, we were able to fully synthesize and implement this design on our target FPGA.

3.2 Cost Analysis of FPGA Resources

As with any engineering project, it is important to minimize the cost of your design while still allowing the needed functionality. When we first started research on this project, the target device to be used was the Xilinx Virtex 2 VP20. When the area of this research moved to implementing the Jacobi method, we found that this FPGA was not large enough to handle our design. Just the divider on the VP20 took up around ½ of the available slices. We then put together data to analyze the cost of moving up to a larger FPGA vs the benefits obtained. Some of the results of this study are given in the spreadsheet below. A more detailed version is available in the appendix. We looked at the cost per 64-bit multiplier we could build, cost per MFLOP and cost per slice. The end result of this study is that we determined it to be most beneficial to use the new Virtex-4 line of products for our design. They offer higher clock frequencies and are large increase in slices available. The chip we most ideally wanted to use for purposes of this design was the XC4VLX60 because it offered the best combination of resources and low cost. Unfortunately, even using the largest package available on the VLX60, we were not able to satisfy our I/O demands, necessitating the move to the VLX80 which did offer higher pin count packages.

Table 2: FPGA Cost Analysis

FPGA	# Slices	# 18x18 Multiply Blocks or DSP Units	Top Frequency of Multiply	# 64-bit Multipliers	Cost per MFLOP	Cost per Multiply	Cost per slice
XC2VP2	1408	12	149 Mhz	0	N/A	N/A	$0.057
XC2VP4	3008	28	149 Mhz	1	$0.98	$146.30	$0.049
XC2VP20	9280	88	149 Mhz	5	$0.40	$59.84	$0.032
XC2VP40	19392	192	149 Mhz	12	$0.44	$65.82	$0.041
XC4VLX60	26624	64	228 Mhz	7	$0.37	$84.07	$0.022
XC4VLX80	35840	80	228 Mhz	8	$0.62	$142.31	$0.032

3.3 Components and Software Used

Our initial Virtex-4 FPGA we choose was the VLX60, but after running into many resource walls such as limited pin count, and limited slices, we decided to use the XC4VLX80 which fit our design very well.

The memory components used were

Qty: 6 -- MT47H64M4 Micron DDR2 SDRAM components

Qty: 6 – K7P323688M Samsung SRAM components (32 Mb each)

We used Xilinx Integrated Studio Environment 8.2 for design, we also used ModelSim as our simulation environment.

4 Design Implementation

4.1 Overview

This project was implemented on a Xilinx Virtex 4 FPGA, specifically the XC4VLX80, speed grade 11 with 1148 pins. The features available through this design include:

- Ability to handle double precision floating point numbers in IEEE-754 format
- Ability to add additional FPGA units to the design without reprogramming
- Parallel bus implemented for Multi-FPGA setup to allow result sharing.
- External Memory interfaces that are able to keep Computational Unit pipeline full and obtain peak floating point performance.
- USB 2.0 core included on the design, and pins reserved for connection to the outside world.
- Successfully synthesized to run at 300Mhz for the memory controller/ 150Mhz for other components.
- Ability to run Jacobi algorithm on one row of A matrix every clock cycle.

The primary components and primary signals for each device on the FPGA are shown on the next page in Figure 8. A brief description of the block is given on the following pages. These block units include:

- Data Processing Unit
- Computational Unit
- Main Controller
- Error Calculator
- USB Core

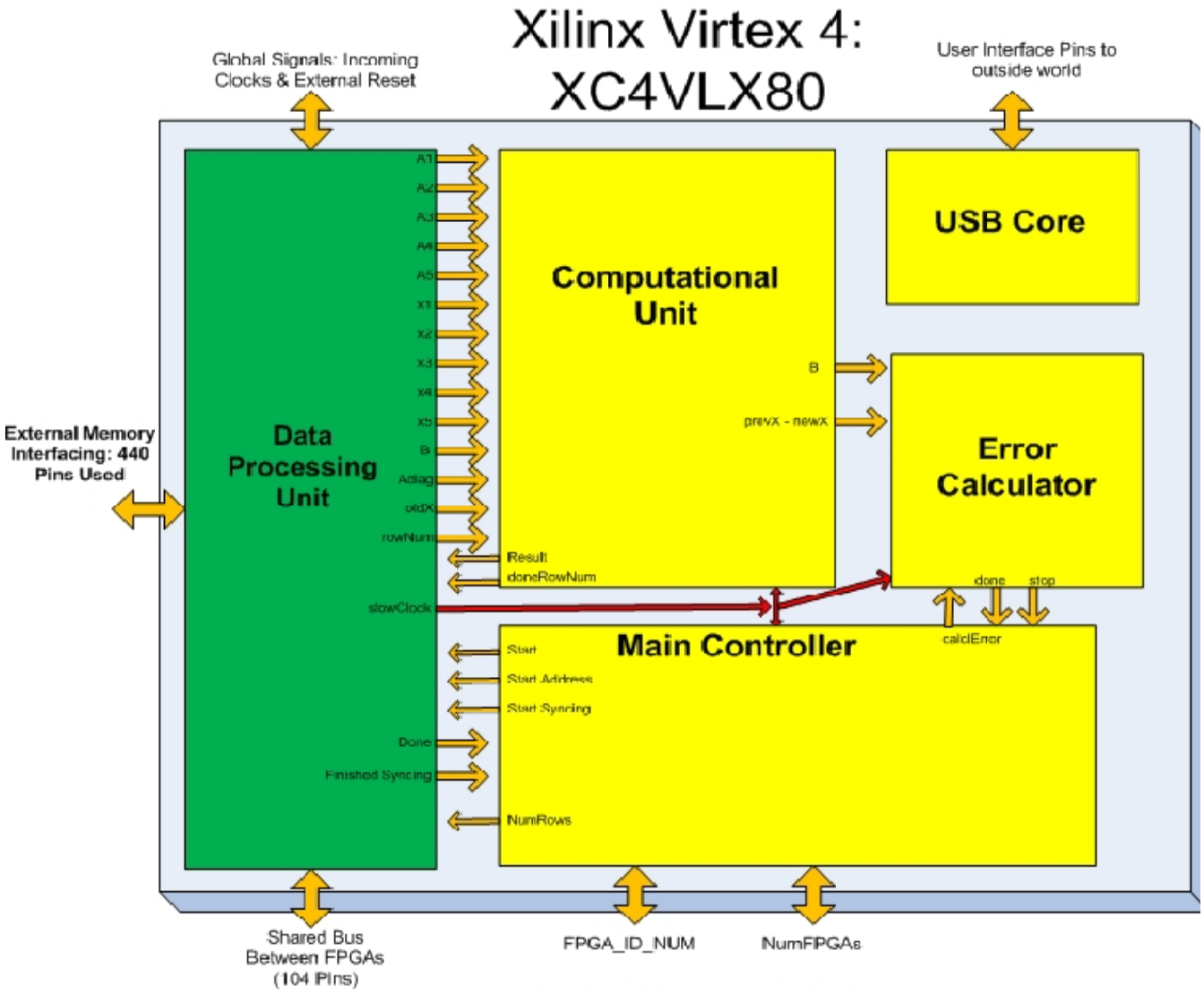


Figure 8: FPGA Block Diagram

4.2 Main Component Descriptions:

- Data Processing Unit: The job of the Data Processing Unit is to pull data from two memory controllers for the A, X & B values. This unit also takes in the global Clock signals & the reset signal from the outside world. It uses a DCM to generate a variety of clock speeds and phases which are used by the two DDR2 memory controllers to operate. The DCM also generates the K & Kbar clocks used by the SRAMS and another clock which runs at ½ the frequency of this block. This clock is passed to the other blocks in the design. The Data Processing Unit runs at 300Mhz, the remainder of the design runs at 150Mhz. After pulling the A, col & b values from Sequential memory, the Data Processing Unit must pull the correct values for X from the SRAMS. It must then resynchronize this data with the Calculation block at 150 Mhz.
- Main Controller: The main controller handles the task of starting an iteration, once an iteration is done it signals for the Error Calculator Module to calculate the error over the given iteration. At the same time the main controller begins the process of updating the old x values in the SRAM with any new x values that have been calculated. Once the controller has been signaled that all values have been updated and that the error calculation is complete, it looks at the done signal from the error Calculator to determine whether to start another iteration.
- Computational Unit: The computational unit handles solving the Jacobi equation presented earlier in this paper. This unit consists of (6) 64-bit floating point multipliers, (4) 64-bit floating point adders, (2) 64-bit floating point subtraction & (1) 64-bit floating point divide unit. There are also some delay units which help ensure that the pipeline remains even and that the correct values get multiplied at the right time.
- Error Calculator: The error calculation module takes in the B value & the difference between the x just calculated and the old x (this calculation is done in the computational unit). The error calculation

module keeps track of the largest of both of those values during an iteration. When signaled it begins a division of these two largest values to get an error which it then compares to a desired error. Based on this, the error Calculation module signals the Main Controller to start another iteration or not to.

- USB Core: The purpose of the USB Core is to simply act as a place holder to take up logic and pins on the device. The reason for needing this is to avoid using so many resources that we leave ourselves with no way of getting data into and out of our device. The USB core comes from opencores.org and has been partially verified. It has not been integrated into the data path or controller, but does serve as one way to get data into the device should we ever decide to physically implement the design. Communication between a host computer and the FPGA was beyond the scope of this research.

4.3 Description of Computational Unit

The computational unit is the heart of this design in that it is the location where all the work is done. A block diagram of this unit that examines the timing is presented in (Figure 9: Computational Unit). The error calculation module also does some of the calculation work, but all other parts of this design exist for the purpose of getting data into and out of the computational unit. The nature of the Jacobi Method allows us to use a pipelined design without having to worry about data hazards. We use this to our advantage, in Figure 9 we present our computational unit which has a Latency, or pipeline depth of 94 cycles. For every row of our Matrix we always have six values. The computational unit takes this into account by providing the appropriate number of floating point units so that every cycle an entire row of the matrix can be solved for. Because of the pipeline latency, the result of the very first row of the matrix will not appear until 94 cycles after the data enters the unit. While this may seem to be poor performing, the advantage is that every cycle after those initial 94 cycles, another row of the matrix is solved for. The goal of this research is to tackle matrices with dimensions up to 1 million x 1 million. The time needed to fill the pipeline becomes insignificant when you compare the 94 cycles needed to fill the pipeline initially to the million cycles after that.

In addition to calculating the X value for the next iteration, the computational unit in this design solve for

$$(|x_i - \tilde{x}_i|)$$

which is the top value needed by the error calculation module. We can easily calculate this value by feeding in the current X value at the beginning of the pipeline and then delaying it the appropriate number of cycles so that we can subtract the newly calculated value from it at the end of the pipeline. We then pass this value which we term “deviation”, back to the error calculation module which tracks the max deviation to determine its error at the end of an iteration.

Getting the correct data into and out of the computational unit is probably the most tricky part of this design. While this area is touched on more in the section on the Data Processing Unit, we briefly examine what is needed below. Besides the clock, there are 17 values that enter or leave this module. The currentRow, doneRow and deviation values can all be ignored because they are used internally by the FPGA and do not have to go into memory. That leaves 13, 64-bit Values that must be read into the computational unit, and 1, 64-bit value that must be written back to memory from the computational unit. The 64-bit entry that is written back to memory is easily handled, but the 13 values entering the unit are not so easily handled. The computational unit must be fed 832 bits of data from memory every clock cycle in order to make maximum use of the pipeline by keeping it full. The Computational unit if our design operates at 150 Mhz, so this translates to a needed data rate of 20.4 GB/sec of data coming from memory to the calculator.

The last issue to touch on with respect to the Computation unit is the use of Delay Units. In any pipelined design, it is important that all values that enter the pipeline together need to stay together with respect to time as they progress through the pipeline. An example in our design is the currentRow value. If we did not delay this value the same 94 cycles that the other entries go though, you would quickly get erroneous values from your design. There are a variety of delay units in this design, all of which are simply a chain of flip flops meant to emulate data passing through a pipeline. By using these delay units, we ensure that all paths of the design have equal latency.

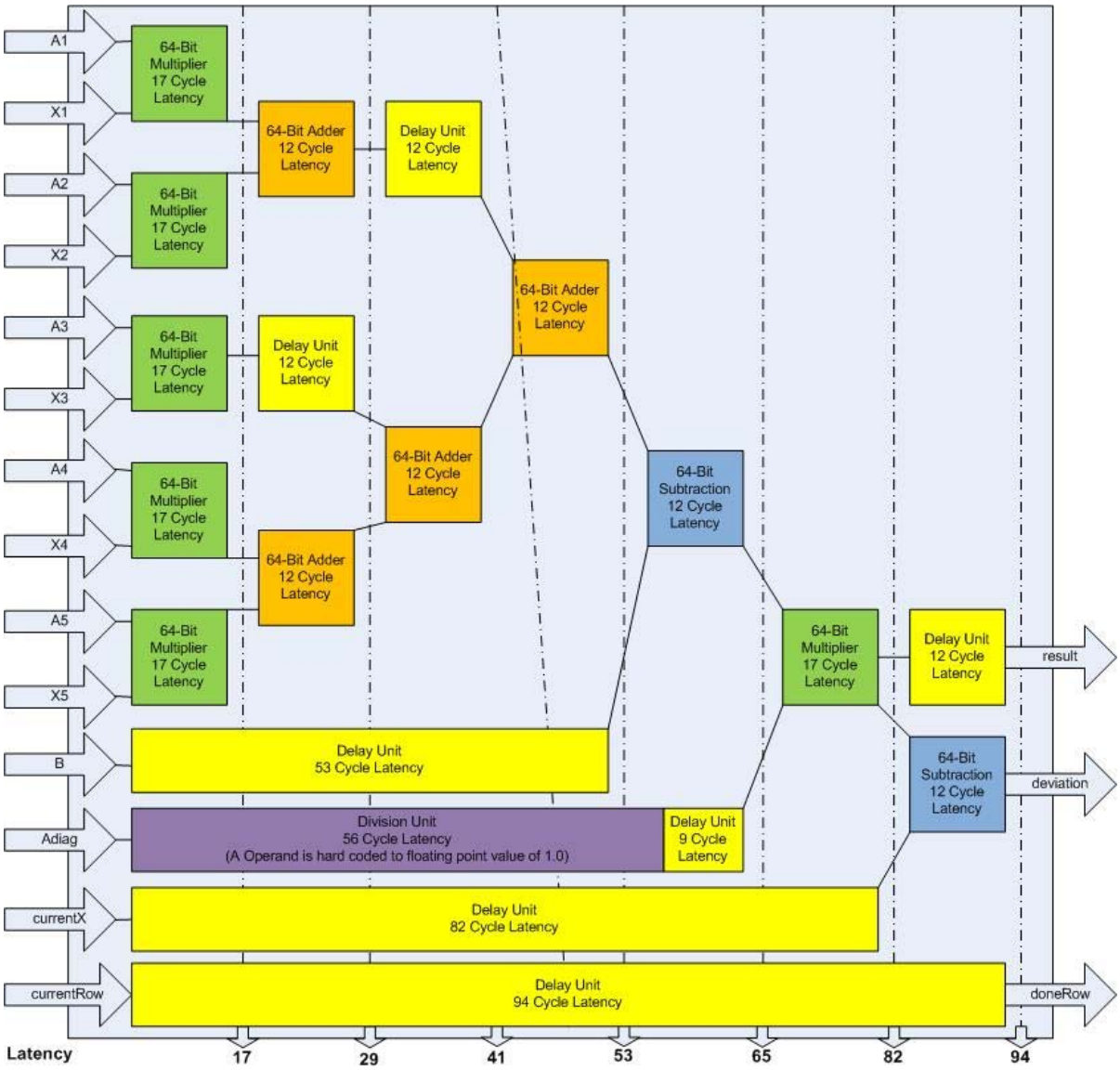


Figure 9: Computational Unit

4.4 Description of Error Calculation Module

The error calculation module is relatively simple. We pass in two 64-bit floating point numbers to this module for every clock cycle. The first of these values, B, is run through a floating point unit which keeps track of the largest value passed through since the last reset. Similarly, the value "deviation" which is calculated by the Computational unit every cycle also passes through and the largest version of it is also tracked in the module.

This module then sits and waits for a start signal, after which it will tell the Floating-Point Divider to begin dividing the largest deviation value by the largest B value. When this calculation is complete, the module compares the computed error value to a desired error value. If the computed error is smaller than the desired error, the module asserts the stop flag high to indicate that there is no need to continue iterating. Otherwise, the module leaves the stop flag low. In parallel, the module asserts the done flag which tells the controller to then look at the stop flag to determine if it should begin another iteration.

4.5 Description of Data Processing Unit

The Data Processing Unit is the most complex block of this design. We will therefore begin this section by examining what is needed by the Data Processing unit before describing the design of the unit. The Data Processing Unit exists solely to pull the needed data out of memory and to pass it on to the Computational Unit. As mentioned in the Computation section, our design needs a very large memory bandwidth of around 20 Gbps. Your typical top of the line computer today tops out at about 8 Gbps, so for purposes of this design we need to come close to doubling this result. Memory bandwidth is not the only major limiting factor in our design. The other is the type of accesses to memory we are going to be performing. In the case of our design, there are two different types of memory access we will be performing. One is sequential, the other is random. The three main parts of the Data Processing unit are the Memory Controllers, The Resynchronization Circuit and the Switchboard Circuit. The purpose of design of each of these units is described below.

4.5.1 Memory Controllers

As was mentioned in the introduction of this paper, we use a special method for storing the A values, col values and B values for every row in memory. The data for these rows is stored in memory sequentially and is accessed sequentially for every iteration through the data. Sequential access to memory is the preferred way of doing things in DDR memory because it allows you to maximize the bandwidth you actually see with your memory. The type of memory that we choose to store our sequential data in is known as DDR2. DDR2 memory stands for Dual Data Rate and is a type of dynamic memory where data can be clocked into the controller on both the rising and falling edge of the clock, essentially causing the memory itself to run at twice the clock speed. Dynamic memory is memory where the values you are storing are kept capacitively in a grid of storage cells. Because of the nature of this storage type, values must be “refreshed” at periodic intervals.

Rows must also go through a process known as precharge before they can be accessed. For Random accesses to DDR memory, it is difficult to avoid the delays due to refreshing and precharge, and therefore you typically see only a fraction of your memories peak bandwidth during these types of accesses. For sequential accesses of data however, delays due to refresh and precharge can be avoided by clever data storage techniques and by switching between the multiple banks of a memory device at the appropriate time. The sequential portion of the data for our design also takes up the largest amount of memory with 7 64-bit values and 6 32-bit values for every row of the matrix. The combination of the need for large memory storage and sequential data access makes DDR2 the best candidate for the sequential data. Our design actually uses two DDR2 controllers. One has a very large 320-bit data bus which we use to bring in our sequential data quickly. The other controller has a smaller 64-bit data bus which we use to store the resulting x values from the computation. Both of these controllers are sized exactly to match the amount of information needed to move into or out of memory each cycle.

As has been mentioned before, the majority of the Data Processing Unit runs at 2 times the clock rate of the Computational Unit This was done because with even the most powerful memory controller, we were not able to get the memory bandwidth needed due to a limited supply of user pins. When we synthesized our Computational Unit, we found that due to the nature of these floating point operations, the Computational Unit could only operate at a max frequency of 183 Mhz. We also discovered that through a Xilinx utility known as the Memory Interface Generator, we could create custom memory cores that are pre-verified to operate at a given frequency. We were then able to spec out a memory chip that was cable of running at 300 Mhz on our FPGA. The opportunity this provided was to allow the Memory to be used at 300 Mhz and then use ½ of this clock frequency to clock the rest of the design (150 Mhz). From the perspective of the Computational Unit, we just doubled the amount of data that can be pulled in from memory. After this design decision was implemented, it was then feasible to get the needed memory bandwidth given a limited number of pins and other resources.

We have mentioned the use of two DDR2 memory controllers in our design. There is another type of memory that we use in our design called SRAM (Static RAM). Static RAM is different from Dynamic Ram in that the data bits are stored in a tightly packed grid of flip flops that are not susceptible to the same kind of capacitive leaking problems that dynamic ram has. Our design requires pulling 6 completely random values from the X matrix every cycle. These x values are pointed to through use of the column vector and are 64-bit floating point numbers. Static RAM does not have the same timing restrictions that Dynamic RAM has. For the type of Static RAM that was spaced out for this design, it can successfully operate at speeds of 300 Mhz and there is a one cycle turnaround time on a read request. This guarantees that when you request a value from memory, you are able to access that value on the next clock cycle. The need to access 6 completely random values from a matrix is the primary reason behind using SRAM to hold the x vector. Disadvantages to using SRAM are that it has a cost per bit many times higher than Dynamic RAM. It is also harder/impossible to find in larger densities (i.e. 128Mb, 256Mb., etc).

In order to pull the "X" values needed for a certain row, we choose to use Static RAM (SRAM) memory. Static RAM has the advantage that it can always provide the needed data on the next clock cycle despite the random nature of the requests. Each SRAM configuration contains two 32 Mb SRAM chips accessed in parallel to give the needed 64-bit data width and 1,048,576 entry depth needed. We use 3 sets of SRAM devices configured in this manner, the purpose of each is to hold the entire "X" vector in memory, so basically there are 3 copies of the vector In memory at all times. The reason this is needed is because of the need to pull 6 random values from memory every 150Mhz clock cycle. By running the memory controllers at 2x the main clock, it allows us to pull 2 values every cycle. By accessing 3 SRAM controllers for 3 different values, we are able to access the needed 6 entries. SRAM is one of the most costly memory technologies out there, and we acknowledge that this is not the most efficient way things could be done. We examined using multi-ported SRAMs which run much slower than today's State of the art SRAMs, we also looked at using

Dynamic RAM. None of these solutions allowed us to consistently access the 6 values from memory every clock cycle.

4.5.2 Resynchronization Circuit

We take advantage of running the Memory Interfaces at 2x the base clock frequency by bringing in half of the needed data on one clock cycle, and the other half of the data on the next clock cycle. This complicates the design quite a bit because you have to get the data to the right registers depending on what clock cycle you are on, and you also have to somehow make this data where the slower 150Mhz clock can access it. This is the sole responsibility of the Resynchronization circuit. The first thing the Resynchronization circuit does is create a signal called clockingMode by ANDing together the 300Mhz clock signal and the inverse of the 150Mhz clock signal. The result of this is an alternating 0 -> 1 -> 0 pattern for every clock cycle of the 300 Mhz clock. We use this as a clock enable on registers after the memory controller to put the data in the right locations depending on what clock cycle we are on. The 6 Column values are clocked into the read address of the SRAM controllers. Again, we use the clockingMode signal to clock in half of the read addresses on one clock cycle, and the other half on the next clock cycle. As the requested data leaves the SRAM we clock in the data to the appropriate flip flops depending on the clock cycle we are on. While this is being done, the other data (A & B values) are delayed so that they remain even in the pipeline. The final part of the resynchronization circuit takes the 6 X values, the 6 A values and the B value and feeds each into its own "Synchronization" flip flop which is clocked at the 150 Mhz clock rate. This helps bring the data back into a domain useable for the Calculation Module.

4.5.3 SwitchBox Circuit

The Switchbox circuit comes after the Resynchronization circuit and is clocked at the 150 Mhz clock rate. The values needed by the Computational unit are 5 A values (A1 -> A5), 5 X values (x1 -> X5), the A value that lies

on the diagonal, the X value matching up with that diagonal A, and the B value. So far the Data Processing Unit has been able to provide 6 A & 6 X values and the B value, but has not been able to distinguish which A & X values match up with the diagonal. The Switchbox takes in the Column values for the data and compares the column values to the Row Number being processed. For one, and only one of these column values there will be a match, this is the diagonal element. The result of this comparison is fed into a series of one hot encoders which then feed the appropriate values to X1->X5, oldX, ADiag & B. This data is now read to be pushed into the Computational Unit.

4.5.4 Result Memory Controller

Another part of the Data Processing Unit is the Result Memory Controller. This controller operates at a frequency of 300 Mhz. It receives a result value from the Computational Unit every other clock cycle from its perspective. When moving data from the 150 Mhz clock domain to the 300 Mhz domain, a Resynchronization circuit is not necessary, so therefore the data gets stored directly into the controller. The data going out of the controller however is also piped to leave the circuit through the parallel bus used to connect multiple FPGAs. A small Finite State Machine controls the Result controller when the Main Controller has us in synchronization mode. This FSM allows us to synchronize the results calculated on the previous iteration with the results stored in the SRAM on all chips attached.

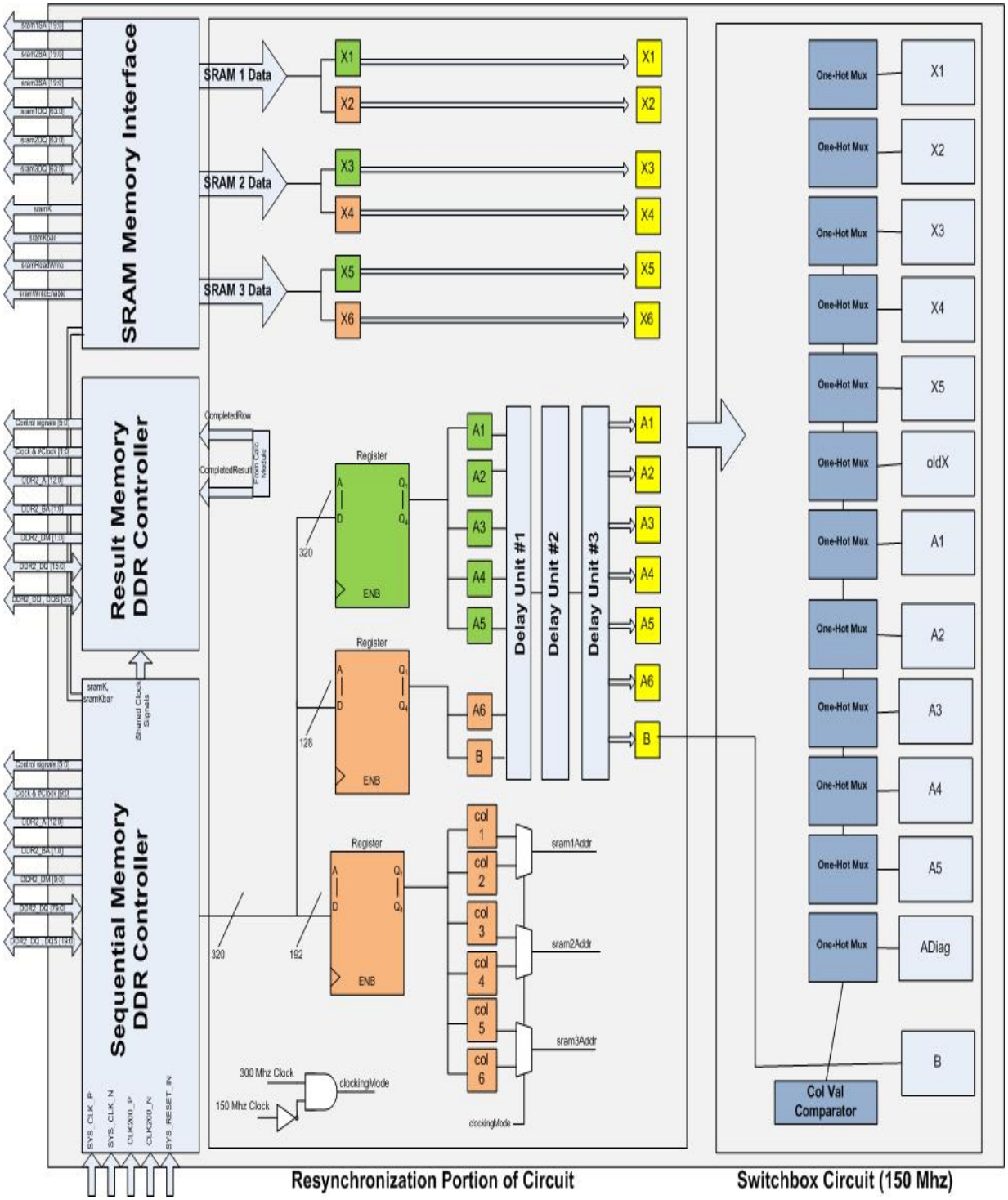


Figure 10: Data Processing Unit

4.6 Description of Main Controller:

The Main controller for this design is rather simple. It handles taking the Chip through 3 stages. The first is the Initialization stage where in a final design it would handle getting the data from the outside world into the appropriate memory locations. As was mentioned before, we have integrated a USB 2.0 Core onto the FPGA which could be used for this purpose, but the scope of which is beyond the topic of this research. For purposes of this project we assume the data is already in memory, at which point the Main Controller is in Iterate mode.

The Main Controller sends a signal to the Data Processing Unit to start iterating which causes the DPU to sequentially send the data for each row of the matrix through the Computational Unit. Once the Data Processing Unit has finished with all entries, it flags the Main Controller that it is done. The Main controller enters a Error Calculation Phase where it signals the Error Calculation module to begin calculating the error. At the same time the Controller tells the Data Processing Unit to begin synchronizing its new "X" values with the SRAMs and with other Xilinx chips through the parallel bus.

Once the Main Controller receives a signal of done from the Error Calculation Module, it then looks at the result to determine if it should continue onto another iteration or stop. In a finished product, the Controller would begin giving the data to the host computer if it reached a stop condition, but that again is beyond the scope of this research.

4.7 Multi-Chip Coordination

One of the aims of this research was to make the design as expandable as possible, allowing more values to be calculated by simply plugging in additional FPGAs. The need arises to have communication between these FPGAs which is why a parallel bus was designed to connect the units. If you have more than one FPGA working on the Jacobi method, each Chip can have a subset of the A, column & B vectors. The only values that cannot be divided up are the X values, which each FPGA needs to have its own copy of in SRAM. The purpose of this bus is to allow these values to be updated across all FPGAs at the end of every clock cycle. There are 4 different buses that make up this communication. One of the buses is 4 bits wide and is an output from the main controller and an input to all the others. This signal acts as a token, allowing whichever Chips ID is asserted communicate across the larger bus. The other bus is 4 bits wide and is tri-state to all chips, it is used as a confirmation of the chip currently transmitting. The next bus is 32 bits wide and is tri-state. The chip that is transmitting reads is result values from memory one at a time, and transmits the Row number on this bus and the data value on the other data bus which is 64-bits wide.

Because three of the buses are tri-state, the need arises for a way of coordinating use of the bus. The one way signal from the main control FPGA acts as this method of control. The buses on each FPGA are hard wired to only become a transmitter if its Chip ID is asserted on these lines, in all other instances the FPGA acts as a passive receiver.

Once a signal is received from the main controller to begin synchronization, the main FPGA asserts its Chip ID and begins sequentially transmitting its values to the other chips which receive the data and store the values in their SRAMs. The main controller knows how many controllers are present by an external signal on its pins. It then begins sequentially allowing these controllers to update their values on the bus. Once this process is complete, a signal is sent to the main controller that the next iteration can begin.

5 Results

5.1 Resource Utilization

Table 3: Resource Utilization

	Used in Proc. Unit	Memory Control Unit	Multi-Chip Sync	Total	Max Available	% Used
Slices	18857	5216	0	24073	35840	67.2
Slice Flip Flops	29314	4247	0	33561	71680	46.8
4 input Luts	24120	9222	0	33342	71680	46.5
bonded IOBs	50	448	72	570	768	74.2
FIFO/16/RAMB1s	1	14	0	15	200	7.5
GCLKs	2	14	0	16	32	50.0
DSP48s	54	0	0	54	80	67.5
BUFIOs	0	12	0	12	52	23.1
DCM_ADVs	0	3	0	3	12	25.0
ISERDES	0	96	0	96	768	12.5
OSERDES	0	132	0	132	768	17.2
PMCDs	0	2	0	2	8	25.0

The table above shows the resources available on the chosen FPGA as well as what was used. As was expected, the largest resource used in this design was I/O, this is mainly due to the huge memory bandwidth required for the design as well as the use of multiple memory controllers. The good thing is that because are are only using 75% of our I/O, it leaves some room for increasing the memory bandwidth in the future. The next most used component were DSP48 blocks, these were used up by our 6 multipliers. We only take advantage of 67.5% of these however, and using more of them will only mean a need for higher memory bandwidth. We have an adequate supply of slices still available, we are only using 67.2% of our supply. The other slices could be used for more advanced bus interfaces for multi-chip coordination or for PCI-express or PCI interfaces to a host computer. Fortunately none of our resources were used over 100% meaning this design can be implemented on our target hardware.

5.2 Timing Analysis

The Computational Unit in this design allows for 12 64-bit Floating Point calculations to occur every clock cycle. At our 150 Mhz clock frequency, this equates to a 1.8 GFLOPS rate of calculation when using just one FPGA. This number increases linearly as you add more FPGAs to do computations. This peak value of 1.8 GFLOPS is unfortunately not the sustained rate of calculation that is possible for this design. The reason for this is the delay required to synchronize the new X values with the old X values at the end of every iteration. This design is split into those two phases of operation, where in the Calculate phase you see the 1.8 GFLOPS peak value, but in the synchronize phase you see 0 GFLOPS.

If you have a problem with an "A" matrix with n rows, you will spend n clock cycles doing the computation. Fortunately the part of the design which does the synchronization operates at 2x the base clock frequency, so therefore you only spend n/2 cycles synchronizing your results with the rest of the circuit. The result of this means that for a 1 FPGA unit design, you really see only 66% of your peak GFLOPS that can be sustained. This gives us a sustained calculation rate of 1.18 GFLOPS. While it would be nice to somehow be able to avoid this performance penalty, the end result is still an improvement over Today's general purpose CPUs. (1) discuss how with a General Purpose Processor, you typically only get about 1% of its peak Floating Point performance on sparse matrix calculations such as these. On today's processors, this would equate to about 400 MFLOPS performance. This is still a speed up of 2.95 for these calculations over today's state of the art processors that run at 3 – 4 GHz.

5.3 Design Bottlenecks

Unfortunately one of the main bottle necks of this design is the need to resynchronize the X values calculated with the previous iteration with the values currently in SRAM. We end up spending 33% of our type performing this task. Another clear bottle neck with this design is the use of SRAMs in general. SRAMs are only available in smaller density packages. The 32 Mb SRAMs we specified for this project are at the high end of capacity for what is available today. If we wish to increase the size of the matrices we want to calculate, we would need to linearly increase the SRAM memory which we have already close to topped out.

Fortunately other memory technologies exists that may be able to be used to solve this problem. A technology known as RLDRAM (Reduced Latency Dynamic RAM) is targeted towards applications where random accesses to memory need to be quick. This type of memory currently gets close to SRAM performance, but would still require multiple clock cycles to get random pieces of data from memory. It is possible that future design revisions could try to incorporate multiple RLDRAM controllers in place of the SRAM controllers. This could be coupled with a type of on chip cache in hopes of being able to provide the same random access capabilities of the current SRAM modules.

The next bottle neck that has been identified is limited I/O resources available. FPGAs have some of the highest pin counts available for any ICs on the market. The Xilinx FPGA we targeted had a pin count that is on the higher end of what is available (640 User I/O pins available). This design used the majority of those pins with around 75% being used for memory interface purposes. Slice resources and DSP units (used for multiplications) are still available on this design, but if future revisions wanted to increase the calculation rate, they would also need to bring in more data from memory. With the current configuration it would be difficult to add any more to the already high memory bandwidth available.

5.4 Areas for Improvement

There are two major areas which I see that could be improved upon with this design. The first is that in order to be able to calculate matrices with larger than 1 million x 1 million dimensions, we need to choose an alternate method of retrieving random data from memory. The current method of using SRAMs for this works, and is very quick. The down side is that it is not a method that can be easily expanded by simply adding in higher capacity SRAMs. Another down side is that SRAMs tend to be many times the cost per megabyte of today's DRAMs.

The second portion of the design I would like to see improved upon is the synchronization mode of the circuit. If there existed a better way of quickly updating these memory values, that would be ideal. Currently we receive a 33% performance hit because of the way we handle the task. One way to solve both of these problems would be with a custom designed memory solution. One way to go about this would be to design a multi-ported SRAM that could operate at 300+Mhz, and could also store two values for every address. You could then add logic in the SRAM to allow a bulk update on command where all the new X values could overwrite the old X values in one clock cycle. The multi-ported feature would allow you not to have to keep multiple copies of the same data in multiple expensive SRAMs.

Finally, the design could use help in general to become a more polished product. Studies could be done on the easiest interface for hooking this type of design into the PC, then you could add the appropriate core and needed logic to make this usable in the real world.

6 Future Work

Future work that could be useful for this research would include actually designing the interface and drivers between the USB core and the Host computer as well as the USB core and the main memory. Another useful function would be a co-processor that takes in the matrix data first, analyzes it and produces a more educated first guess than our all zeros values currently are. Previous works (5) have shown that by having a good, educated first guess to the first iteration of your run, you can drastically reduce the number of iterations your design needs to run. Finally, implementing other Iterative algorithms on this platform would be useful because it would allow for different types of matrices to be calculated on this hardware. At current, the Jacobi solver is only suitable for diagonally dominant matrices. Other memory access techniques should be explored as well. As has been mentioned, SRAMs are one of the most limiting factors on expanding this design to even larger matrices. Solutions that could be explored include looking at RLDRAM and other similar technologies to offer reduced random access latency at cheap cost and high density. It would be interesting to see if you could design an on chip cache system combined with these technologies and get the same or better performance than seen with this design. The largest benefit here would be in having the capability to solve 10 million x 10 million or larger matrices.

7 Conclusion

This work has presented architecture for solving Matrices using an iterative process with the Jacobi method. This work expands upon previous research by allowing for the calculation of large matrices with this method by using external memory to store the data rather than the limited on-chip memory. The results of this design are very promising. We calculate a peak Floating Point performance of 1.8 GFLOPS for double precision arithmetic. During operation however, we only receive a sustained performance of 1.18 GFLOPS. This is still a good result as it is a speed up of around 2.95 when compared to the sustained 400 MFLOPS you would typically get with today's state of the art computers. We were also able to present a memory architecture that is capable of solving matrices over 100 times the size that previous research was able to handle.

The need for custom FPGA solutions to solve problems is something that is becoming more common rather than less. While today's general purpose computers are getting faster and faster, the memory wall problem is getting worse. This means, that for the type of calculations looked at in this paper, increasing the CPU performance just is not going to be enough to get your results faster. While power was not analyzed in this project, it is most likely only a small fraction of the power used by the power-hungry Itanium or Xeon processors, while still running almost 3 times as fast. Many vendors are now starting to offer FPGA based solutions where you can plug a card into a PC with several FPGAs onboard that can then be programmed to accelerate specific applications.. For high end scientific applications, this may become more prominent in the future as FPGA performance increases, and the cost per chip decreases.

The downside to solutions like the one presented in this project is that they only work under certain circumstances. This research focused on solution 1million x1million matrices with 6 entries per row. The General purpose computer is much more friendly if you perform a large variety of calculations in various

formats. Other lines of research look at how to make FPGAs dynamically reconfigurable so that your FPGA one second can be a matrix solver and the next a linear equation solver.

8 Bibliography

1. An FPGA-Based Floating-Point Jacobi Iterative Solver. **Morris, G R and Prasanna, V K. .** *Proceedings on the 8th International Symposium on Parallel Architectures, Algorithms and Networks.* s.l. : IEEE, 2005, 420 - 427

2. Jacobi Load Flow Accelerator using FPGA. **Foertsch, J, Johnson, J and Nagvajara, P. .** *Proceedings of the 37th Annual Power Symposium.* s.l. : IEEE, 2005, 448- 454

3. **Garcia, J. .** A FPGA-Based Jacobi Processor. *http://ccc.inaoep.mx/.* [Online] [Cited: August 5 2006.], http://ccc.inaoep.mx/~joaquingr/docs/FPGA_JacobiProcessor.pdf

4. Performance comparison of vector-calculations between Itanium2 and other processors. **Nanri, Takeshi and Watanabe, Yoshitaka. .** *Innovative Architecture for Future Generation High-Performance Processors and Systems.* s.l. : IEEE, 2005

5. **Hackbusch, W. .** *Iterative Solutions of Large Sparse Systems of Equations.* New York, NY : Springer-Verlag, 1994

6. **Warburton, T. .** Solving Scalar Linear Systems (in parallel). *http://www.caam.rice.edu/.* [Online] Fall 2003. [Cited: May 23 2006.], http://www.caam.rice.edu/~timwar/MA471F03/Lecture15b.ppt

7. **Brewer, Kevin J. .** IEEE-754 Floating-Point Conversion from 64-bit Hexadecimal to Floating-Point. [Online] Queens College, Sept 1998. , http://babbage.cs.qc.edu/IEEE-754/64bit.html

8. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. **deLorimier, M and DeHon, A. .** *Proceedings of the International Symposium on Field-Programmable Gate Arrays.* Monterey, California : s.n., 2005

9. **deLorimier, M. .** *Floating-Point Sparse Matrix-Vector Multiply for FPGAs.* Pasadena, California : California Institute of Technology, 2005, Masters Thesis

10. **Eljkhout, V. .** Overview of Iterative Linear System Solver Packages. *http://www.netlib.org/utk/papers/iterative-survey/.* [Online] 15 August 1997. [Cited: July 22 2006.], http://www.netlib.org/utk/papers/iterative-survey/

11. On Sparse Matrix-Vector Multiplication with FPGA-Based System. **ElGindy, H and Shue, Y. .** *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* s.l. : IEEE, 2002, 273-274

12. **Fithian, William S, et al. .** Viva makes programming FPGA environments absurdly easy. *http://www.starbridgesystems.com.* [Online] 5 September 2002. [Cited: July 15 2006.], http://www.starbridgesystems.com/resources/publications/Iterative%20Matrix%20Equation%20Solver.pdf

13. Parallel Block-Diagonal-Bordered Sparse Linear Solvers for Electrical Power System Applications. **Koester, D P, Ranka, S and Fox, G C. .** *Scalable Parallel Libraries Conference.* Mississippi State, MS : IEEE, 1993, 195-203

14. Pipelined Datapath for an IEEE-754 64-Bit Floating Point Jacobi Solver. **Morris, G R and Prasanna, V K. .** *Proceedings of the 9th annual High Performance Embedded Computing Workshop.* MIT Lincoln Laboratory : s.n., 2005

15. Efficient Synthesis of Array Intensive Computations Onto FPGA Based Accelerators. **Shenoy, N, et al. .** *Proceedings of the The 14th International Conference on VLSI Design .* s.l. : IEEE, 2001, 305

16. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs. **Zhuo, L and Prasanna, V K. .** *Proceedings of the 18th International Parallel and Distributed Processing Symposium.* s.l. : IEEE, 2004, 92

17. Sparse Matrix-Vector Multiplication on FPGAs. —. *13th International Symposium on Field-Programmable Gate Arrays.* Monterey, California : ACM, 2005, 63 - 74

9 Appendix

9.1 Introduction

This appendix provides RTL (Register Transfer Language) code of the design. This design consists of a variety of Verilog files and proprietary Xilinx modules. The Verilog files I created as part of this design are included here, proprietary code and files are not included in this Thesis. Also, the results of the simulations done for this design are included with a brief explanation of each test. The hand calculated expected results are presented, along with the waveforms and actual results.

9.2 Simulation Waveforms

The design was broken down and tested in blocks. We show the stimulus given in tables for each block, we also convert the decimal values used to IEEE-754 format in hex. This is thanks to a easy to use conversion program on the web . (7)

Table 4: Calculate Test Entries

currentRow	entryB	currentX	diagA	A1	A2	A3	A4	A5	X1	X2	X3	X4	X5
1	5.8	1.7	-8.6	2.99	1982	1.23	8.78	6.23	8.8	6.6	15.6	11.1	1
2	6.3	2.9	1.13	53.78	33.3	0.56	9.46	1.11	4.7	9.8	13.2	8.6	2

Table 5: Decimal-Floating Cross Reference

Decimal	IEEE-754	Decimal	IEEE-754
5.8	4017333333333333	0.56	3FE1EB851EB851EC
6.3	4019333333333333	8.78	40218F5C28F5C28F
1.7	3FFB333333333333	9.46	4022EB851EB851EC
2.9	4007333333333333	6.23	4018EB851EB851EC
-8.6	C021333333333333	1.11	3FF1C28F5C28F5C3
1.13	3FF2147AE147AE14	8.8	402199999999999A
2.99	4007EB851EB851EC	4.7	4012CCCCCCCCCCCD
53.78	404AE3D70A3D70A4	6.6	401A666666666666
1982	409EF80000000000	9.8	402399999999999A
33.3	4040A66666666666	15.6	402F333333333333
1.23	3FF3AE147AE147AE	13.2	402A666666666666
11.1	4026333333333333	8.6	4021333333333333
1	3FF0000000000000	2	4000000000000000
1537.742	409806f89e1fe923	-588.6	c08264ccccccccd
-1536.042	c098002bd1531c56	591.5	40827c0000000000

9.2.1 Calculation Block

There are two waveforms on the following pages illustrate the inputs going in and 94 cycles later, the result. Referring back to the equations in Chapter 1, the Result from these two back-to-back row calculations should come out as follows:

$$Result1 = \frac{1}{-8.6} * \big(5.8 - \ (2.99 * 8.8 + 1982 * 6.6 + 1.23 * 15.6 + 8.78 * 11.1 + 6.23 * 1)\big) = 1537.7$$

$$Result2 = \frac{1}{1.13} * \big(6.3 - \ (53.78 * 4.7 + 33.3 * 9.8 + 0.56 * 13.2 + 9.46 * 8.6 + 1.11 * 2)\big) = -588.6$$

$$Deviation1 = (1.7 - 1537.7) = \ -1536 \quad Deviation2 = (2.9 - \ -588.6) = 591.5$$

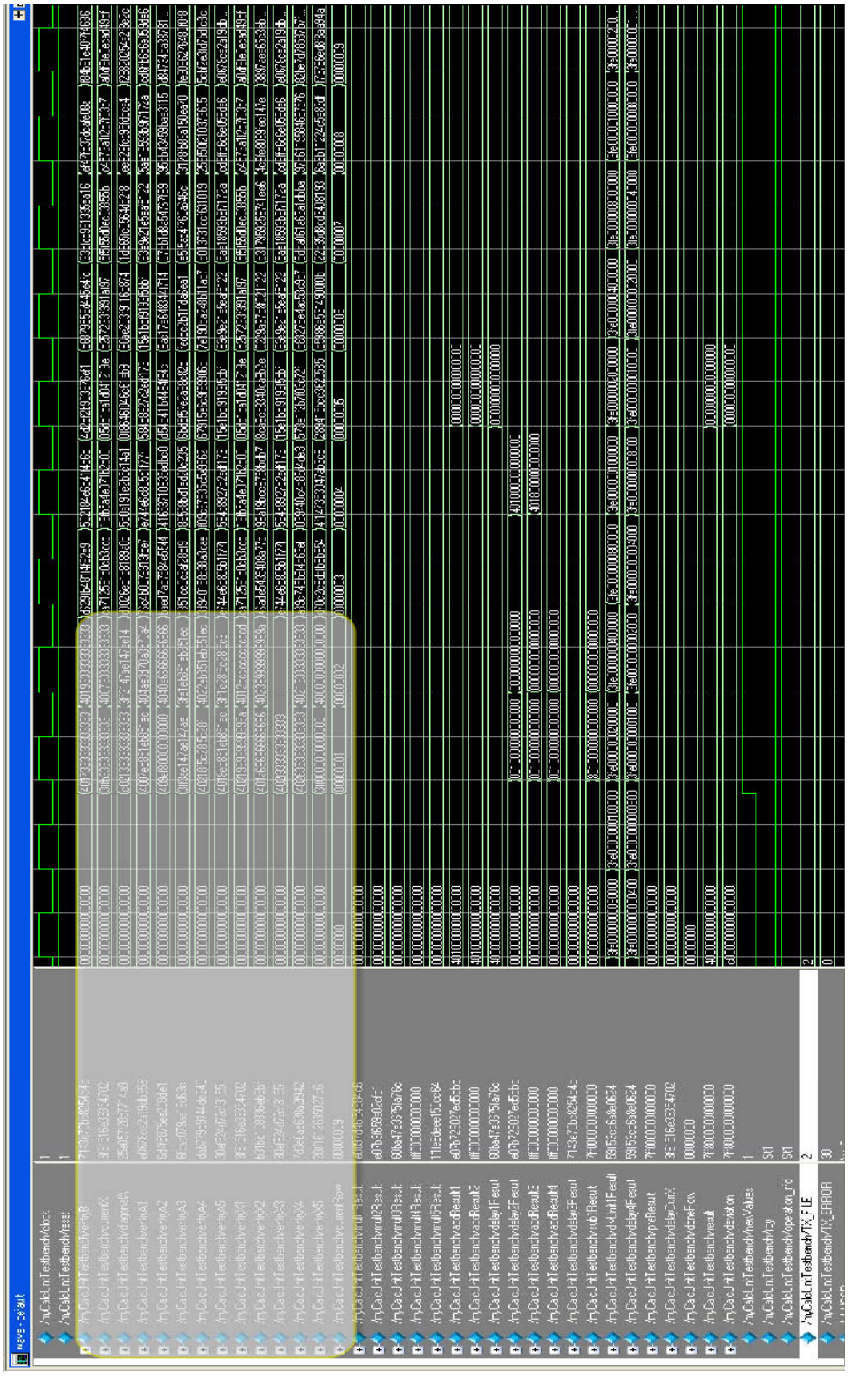


Figure 11: Calculation Module Test Inputs

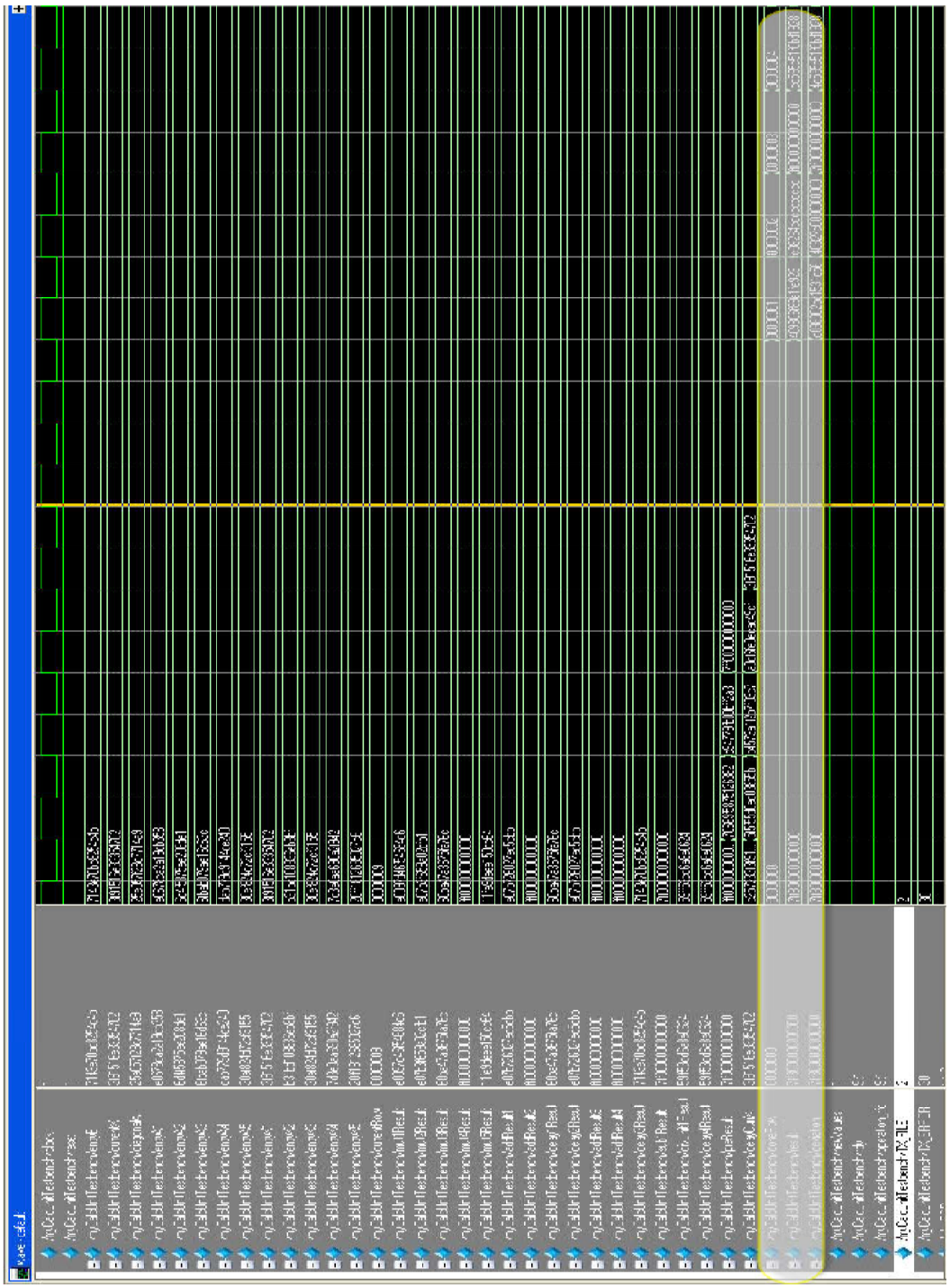


Figure 12: Calculation Test Outputs

9.2.2 Error Calculation Block

To test the error calculation block, there are 3 floating point inputs. Deviation, entryB, and desiredError. Desired error is the largest error you are willing to accept in order to stop the iterative sequence. It is usually set during initialization and stays constant for the entire process. To test this module, we feed in a series of 5 floating-point values for deviation and entryB. This module then keeps track of the largest we put in. When we signal the module to calculate the error, it then does this, compares it to the desired error and flags yes, conditions met or no they are not. There is one waveform showing the inputs, and two for the output under two conditions. One for a desired Error greater than the actual error and the other for one less than the actual error.

Table 6: Error Module Test Bench

Deviation	5.8	1.3	6.8	0.3	9.6
entryB	1.3	1.0	3.2	1.7	2.4

Table 7: Conversion Chart

Decimal	IEEE-754
5.8	4017333333333333
1.3	3FF4CCCCCCCCCCC
1.0	3FF0000000000000
6.8	401B333333333333
3.2	400999999999999A
0.3	3FD3333333333333
1.7	3FFB333333333333
9.6	4023333333333333
2.4	4003333333333333
5.0	4014000000000000
2.0	4000000000000000

The resulting error after the above inputs are pushed through is 3.0. We run two tests with these inputs, one with a desiredError of 5.0 and another with a desired error of 2.0. The first test we expect and see a high criteriaFlag indicating we can stop iterating. The other test we expect and see the criteriaMet flag to stay low.

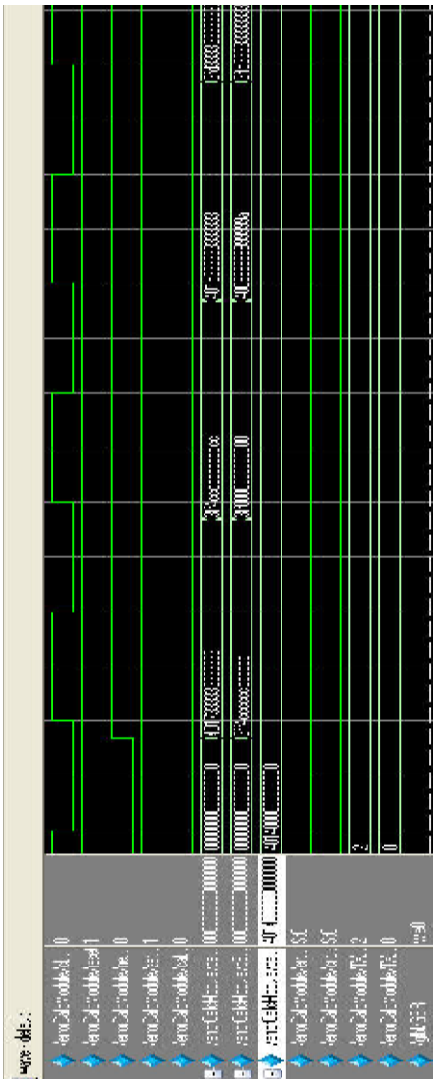


Figure 13: ErrorCalc Inputs

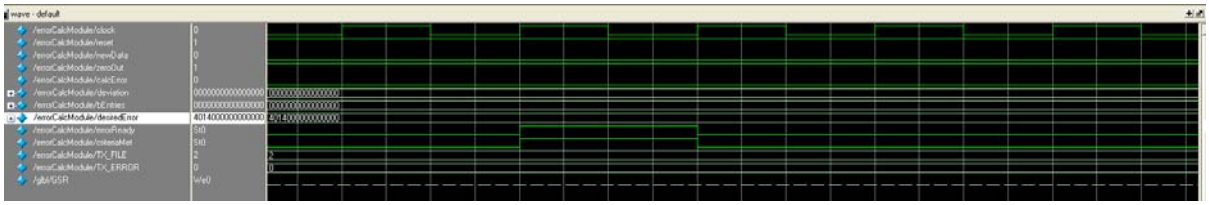


Figure 14: criteriaMet Result Error Module

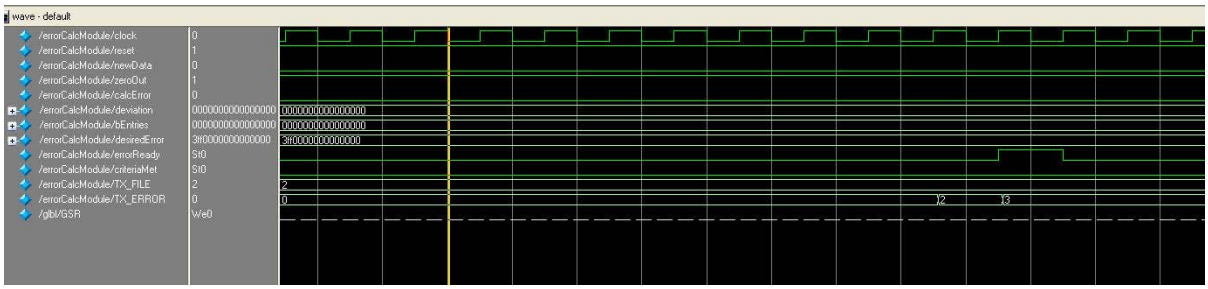


Figure 15: criteria not met Error Module

9.3 Design RTL Code

9.3.1 Organization

The verilog code of the design is included in the next set of pages. The code is broken down by which major logic block it is a part of. Please note the following:

- The code for the two DDR2 memory controller modules is not included in this paper. These controllers are proprietary property of Xilinx and cannot be included because of copyright reasons.
- It was decided to opt out of including the code for the USB 2.0 controller. The controller was used as-is without modification from the opencores.org site. Because of this we decided to give the link to where the code can be downloaded.
    - http://www.opencores.org/projects.cgi/web/usb/overview
- Black-box items are used throughout this code, these are portions of the design that have been generated using the Xilinx IP Cores interface. This mainly includes the DCM modules and the floating point arithmetic modules. No verilog code is available for these portions of the designs.

9.3.2 Computational Unit Top Level Files

topLevel.v

```
module topLevel(extern_reset, clock, reset, phy_clk_pad_i, TxReady_pad_i, RxValid_pad_i, RxActive_pad_i,
RxError_pad_i,
usb_vbus_pad_i, DataIn_pad_i, LineState_pad_i, VStatus_pad_i, phy_rst_pad_o, TxValid_pad_o,
XcvSelect_pad_o, TermSel_pad_o, SuspendM_pad_o, VControl_Load_pad_o, DataOut_pad_o,
OpMode_pad_o, VControl_pad_o, CLK200_P, CLK200_N, SYS_CLK_P, SYS_CLK_N,
MentryBToCalc, McurrentXToCalc, MdiagonalAToCalc, MentryA1ToCalc, MentryA2ToCalc, MentryA3ToCalc,
MentryA4ToCalc, MentryA5ToCalc,
MentryX1ToCalc, MentryX2ToCalc, MentryX3ToCalc, MentryX4ToCalc, MentryX5ToCalc, McurrentRowToCalc);

        //Global IO Related I/O to/from FPGA
  input CLK200_P;
  input CLK200_N;
        input SYS_CLK_P;
        input SYS_CLK_N;
        input extern_reset;
        input clock, reset;

        //USB Controller Type I/O to/from FPGA
  input phy_clk_pad_i;
  input TxReady_pad_i;
  input RxValid_pad_i;
  input RxActive_pad_i;
  input RxError_pad_i;
  input usb_vbus_pad_i;
  input [7:0] DataIn_pad_i;
  input [1:0] LineState_pad_i;
  input [7:0] VStatus_pad_i;
  output phy_rst_pad_o;
  output TxValid_pad_o;
  output XcvSelect_pad_o;
  output TermSel_pad_o;
  output SuspendM_pad_o;
  output VControl_Load_pad_o;
  output [7:0] DataOut_pad_o;
  output [1:0] OpMode_pad_o;
  output [3:0] VControl_pad_o;

        //connection between sram and usb controller
        wire locked_out;
        wire wb_we_i, wb_stb_i, wb_cyc_i;
        wire resume_req_i, wb_ack_o, inta_o, intb_o, susp_o, sram_re_o, sram_we_o, done, start;
        wire [17:0] wb_addr_i;
        wire [31:0] wb_data_i;
```

```
        wire [31:0] wb_data_o;
        wire [15:0] dma_req_o;
        wire [9:0] sram_adr_o;
        wire [31:0] sram_data_o;
        wire [15:0] dma_ack_i;
        wire [31:0] sram_data_i;

        wire [63:0] deviation;

        reg [63:0] entryBToCalc, currentXToCalc, diagonalAToCalc, entryA1ToCalc;
        reg [63:0] entryA2ToCalc, entryA3ToCalc, entryA4ToCalc, entryA5ToCalc;
        reg [63:0] entryX1ToCalc, entryX2ToCalc, entryX3ToCalc, entryX4ToCalc, entryX5ToCalc;
        reg [31:0] currentRowToCalc;

        input [63:0] MentryBToCalc, McurrentXToCalc, MdiagonalAToCalc, MentryA1ToCalc;
        input [63:0] MentryA2ToCalc, MentryA3ToCalc, MentryA4ToCalc, MentryA5ToCalc;
        input [63:0] MentryX1ToCalc, MentryX2ToCalc, MentryX3ToCalc, MentryX4ToCalc, MentryX5ToCalc;
        input [31:0] McurrentRowToCalc;

/*      always@(posedge clock) begin
                entryBToCalc <= MentryBToCalc;
                currentXToCalc <= McurrentXToCalc;
                diagonalAToCalc <= MdiagonalAToCalc;
                entryA1ToCalc <= MentryA1ToCalc;
                entryA2ToCalc <= MentryA2ToCalc;
                entryA3ToCalc <= MentryA3ToCalc;
                entryA4ToCalc <= MentryA4ToCalc;
                entryA5ToCalc <= MentryA5ToCalc;
                entryX1ToCalc <= MentryX1ToCalc;
                entryX2ToCalc <= MentryX2ToCalc;
                entryX3ToCalc <= MentryX3ToCalc;
                entryX4ToCalc <= MentryX4ToCalc;
                entryX5ToCalc <= MentryX5ToCalc;
                currentRowToCalc <= McurrentRowToCalc;
        end*/
        //connections between controller & memInterface
        wire resetMemAccess, enableMemAccess;
        wire [31:0] fetchRow, doneRow;

        iterationController itCont1 (
  .clock(clock),
  .reset(reset),
  .newData(newData),
  .zeroOut(zeroOut),
  .calcError(calcError),
  .desiredError(desiredError),
  .errorReady(errorReady),
  .criteriaMet(criteriaMet),
```

```
.doneRow(doneRow),
.fetchRow(fetchRow),
.enableMemAccess(enableMemAccess),
.resetMemAccess(resetMemAccess)
);

        usbf_top usbController (
.clk_i(clock),
.rst_i(reset),
.wb_addr_i(wb_addr_i),
.wb_data_i(wb_data_i),
.wb_data_o(wb_data_o),
.wb_ack_o(wb_ack_o),
.wb_we_i(wb_we_i),
.wb_stb_i(wb_stb_i),
.wb_cyc_i(wb_cyc_i),
.inta_o(inta_o),
.intb_o(intb_o),
.dma_req_o(dma_req_o),
.dma_ack_i(dma_ack_i),
.susp_o(susp_o),
.resume_req_i(resume_req_i),
.phy_clk_pad_i(phy_clk_pad_i),
.phy_rst_pad_o(phy_rst_pad_o),
.DataOut_pad_o(DataOut_pad_o),
.TxValid_pad_o(TxValid_pad_o),
.TxReady_pad_i(TxReady_pad_i),
.RxValid_pad_i(RxValid_pad_i),
.RxActive_pad_i(RxActive_pad_i),
.RxError_pad_i(RxError_pad_i),
.DataIn_pad_i(DataIn_pad_i),
.XcvSelect_pad_o(XcvSelect_pad_o),
.TermSel_pad_o(TermSel_pad_o),
.SuspendM_pad_o(SuspendM_pad_o),
.LineState_pad_i(LineState_pad_i),
.OpMode_pad_o(OpMode_pad_o),
.usb_vbus_pad_i(usb_vbus_pad_i),
.VControl_Load_pad_o(VControl_Load_pad_o),
.VControl_pad_o(VControl_pad_o),
.VStatus_pad_i(VStatus_pad_i),
.sram_adr_o(sram_adr_o),
.sram_data_i(sram_data_i),
.sram_data_o(sram_data_o),
.sram_re_o(sram_re_o),
.sram_we_o(sram_we_o)
);

        usbmem usbInterfaceSRAM (
```

```
            .addr(sram_adr_o),
            .clk(clock),
            .din(sram_data_o),
            .dout(sram_data_i),
            .en(sram_re_o),
            .we(sram_we_o));

            myCalcModule myCalcUnit (
   .clock(clock),
   .entryB(entryBToCalc),
            .currentX(currentXToCalc),
   .diagonalA(diagonalAToCalc),
   .entryA1(entryA1ToCalc),
   .entryA2(entryA2ToCalc),
   .entryA3(entryA3ToCalc),
   .entryA4(entryA4ToCalc),
   .entryA5(entryA5ToCalc),
   .entryX1(entryX1ToCalc),
   .entryX2(entryX2ToCalc),
   .entryX3(entryX3ToCalc),
   .entryX4(entryX4ToCalc),
   .entryX5(entryX5ToCalc),
   .currentRow(currentRowToCalc),
   .result(),
   .deviation(deviation),
   .doneRow()
   );

           errorModule errorCalc (
   .clock(clock),
   .calcError(),
   .deviation(deviation),
   .bEntries(entryBToCalc),
   .desiredError(),
   .errorReady(),
   .criteriaMet()
   );
endmodule
```

9.3.3 Calculation Module Files

myCalcModule.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    20:38:07 10/03/2006
// Design Name:
// Module Name:    myCalcModule
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module myCalcModule(clock, reset, entryB, currentX, diagonalA, entryA1, entryA2, entryA3, entryA4, entryA5,
entryX1, entryX2, entryX3, entryX4, entryX5, currentRow, result, deviation, doneRow, newValues, rdy,
operation_rfd);

   input clock;
   input reset;
   input newValues;
   input [63:0] entryB;
   input [63:0] currentX;
   input [63:0] diagonalA;
   input [63:0] entryA1;
   input [63:0] entryA2;
   input [63:0] entryA3;
   input [63:0] entryA4;
   input [63:0] entryA5;
   input [63:0] entryX1;
   input [63:0] entryX2;
   input [63:0] entryX3;
   input [63:0] entryX4;
   input [63:0] entryX5;
   input [31:0] currentRow;
   output [63:0] result;
   output [63:0] deviation;
   output [31:0] doneRow;
```

```
wire [63:0] mult1Result, mult2Result, mult3Result, mult4Result, mult5Result;
 wire [63:0] addResult1, addResult2, addResult3, addResult4;
 wire [63:0] delay1Result, delay2Result, delay3Result, delay4Result;
 wire [63:0] sub1Result;
 wire [63:0] divUnit1Result, delayCurrX, preResult;
 output operation_rfd, rdy;
 wire [31:0] preDoneRow;
 wire sclr, rdy1, rdy2, rdy3, rdy4, rdy5, rdy6, operation_rfd1, operation_rfd2;
 wire operation_rfd3, operation_rfd4, operation_rfd5, operation_rfd6;
 assign operation_rfd = (operation_rfd1 && operation_rfd2 && operation_rfd3 && operation_rfd4 &&
operation_rfd5 && operation_rfd6);
 assign rdy = (rdy1 && rdy2 && rdy3 && rdy4 && rdy5 && rdy6);

 assign sclr = ~reset;

    float64mult multiplyUnit1(.a(entryA1), .rdy(rdy1), .operation_rfd(operation_rfd1), .sclr(sclr), .b(entryX1),
.clk(clock), .result(mult1Result), .operation_nd(newValues));

float64mult multiplyUnit2(.a(entryA2), .rdy(rdy2), .operation_rfd(operation_rfd2), .sclr(sclr), .b(entryX2),
.clk(clock), .result(mult2Result), .operation_nd(newValues));

 float64mult multiplyUnit3(.a(entryA3), .rdy(rdy3), .operation_rfd(operation_rfd3), .sclr(sclr), .b(entryX3),
.clk(clock), .result(mult3Result), .operation_nd(newValues));

 float64mult multiplyUnit4(.a(entryA4), .rdy(rdy4), .operation_rfd(operation_rfd4), .sclr(sclr), .b(entryX4),
.clk(clock), .result(mult4Result), .operation_nd(newValues));

 float64mult multiplyUnit5(.a(entryA5), .rdy(rdy5), .operation_rfd(operation_rfd5), .sclr(sclr), .b(entryX5),
.clk(clock), .result(mult5Result), .operation_nd(newValues));

 float64add addUnit1(.a(mult1Result), .b(mult2Result), .clk(clock), .result(addResult1));

 float64add addUnit2(.a(mult4Result), .b(mult5Result), .clk(clock), .result(addResult2));

 delay12cycle delay1 (.clock(clock), .reset(reset), .myin(mult3Result), .myout(delay1Result));

 delay12cycle delay2 (.clock(clock), .reset(reset), .myin(addResult1), .myout(delay2Result));

 float64add addUnit3(.a(delay1Result), .b(addResult2), .clk(clock), .result(addResult3));

 float64add addUnit4(.a(delay2Result), .b(addResult3), .clk(clock), .result(addResult4));

 delay53cycle delay3 (.clock(clock), .reset(reset), .myin(entryB), .myout(delay3Result));

 float64subtract subUnit1(.a(delay3Result), .b(addResult4), .clk(clock), .result(sub1Result));

 float64divide divUnit1(.a(64'h3ff0000000000000), .b(diagonalA), .clk(clock), .result(divUnit1Result));
```

```
 delay9cycle delay4 (.clock(clock), .reset(reset), .myin(divUnit1Result), .myout(delay4Result));

 float64mult multiplyUnit6(.a(sub1Result), .rdy(rdy6), .operation_rfd(operation_rfd6), .sclr(sclr),
.b(delay4Result), .clk(clock), .result(preResult), .operation_nd(1'b1));

 delay82cycle32 delay5 (.clock(clock), .reset(reset), .myin(currentRow), .myout(preDoneRow));
 delay12cycle32 delay8 (.clock(clock), .reset(reset), .myin(preDoneRow), .myout(doneRow));
 delay82cycle delay6 (.clock(clock), .reset(reset), .myin(currentX), .myout(delayCurrX));

 float64subtract subUnit2(.a(delayCurrX), .b(preResult), .clk(clock), .result(deviation));

 delay12cycle delay7 (.clock(clock), .reset(reset), .myin(preResult), .myout(result));


endmodule
```

delay12cycle.v

```
`timescale 1ns / 1ps

module delay12cycle(clock, reset, myin, myout);
   input clock, reset;
   input [63:0] myin;
   output [63:0] myout;

    wire [63:0] delay1, delay2, delay3, delay4, delay5, delay6, delay7, delay8;
    wire [63:0] delay9, delay10, delay11;

    register64wide delayUnit1 (.clock(clock), .reset(reset), .D(myin), .Q(delay1));
    register64wide delayUnit2 (.clock(clock), .reset(reset), .D(delay1), .Q(delay2));
    register64wide delayUnit3 (.clock(clock), .reset(reset), .D(delay2), .Q(delay3));
    register64wide delayUnit4 (.clock(clock), .reset(reset), .D(delay3), .Q(delay4));
    register64wide delayUnit5 (.clock(clock), .reset(reset), .D(delay4), .Q(delay5));
    register64wide delayUnit6 (.clock(clock), .reset(reset), .D(delay5), .Q(delay6));
    register64wide delayUnit7 (.clock(clock), .reset(reset), .D(delay6), .Q(delay7));
    register64wide delayUnit8 (.clock(clock), .reset(reset), .D(delay7), .Q(delay8));
    register64wide delayUnit9 (.clock(clock), .reset(reset), .D(delay8), .Q(delay9));
    register64wide delayUnit10 (.clock(clock), .reset(reset), .D(delay9), .Q(delay10));
    register64wide delayUnit11 (.clock(clock), .reset(reset), .D(delay10), .Q(delay11));
    register64wide delayUnit12 (.clock(clock), .reset(reset), .D(delay11), .Q(myout));

endmodule
```

register64wide.v

```
module register64wide(clock, reset, D, Q);
   input clock, reset;
   input [63:0] D;
   output [63:0] Q;
```

```
  reg [63:0] Q;

  always @(posedge clock or negedge reset) begin
            if (!reset) begin
                Q <= 64'b0;
            end
            else begin
                Q <= D;
            end
  end


endmodule
```

delay53cycles.v

```
module delay53cycle(clock, reset, myin, myout);
  input clock, reset;
  input [63:0] myin;
  output [63:0] myout;

   wire [63:0] delay1, delay2, delay3, delay4, delay5, delay6, delay7, delay8;

   delay12cycle delayUnit1 (.clock(clock), .reset(reset), .myin(myin), .myout(delay1));
   delay12cycle delayUnit2 (.clock(clock), .reset(reset), .myin(delay1), .myout(delay2));
   delay12cycle delayUnit3 (.clock(clock), .reset(reset), .myin(delay2), .myout(delay3));
   delay12cycle delayUnit4 (.clock(clock), .reset(reset), .myin(delay3), .myout(delay4));
   register64wide delayUnit5 (.clock(clock), .reset(reset), .D(delay4), .Q(delay5));
   register64wide delayUnit6 (.clock(clock), .reset(reset), .D(delay5), .Q(delay6));
   register64wide delayUnit7 (.clock(clock), .reset(reset), .D(delay6), .Q(delay7));
   register64wide delayUnit8 (.clock(clock), .reset(reset), .D(delay7), .Q(delay8));
   register64wide delayUnit9 (.clock(clock), .reset(reset), .D(delay8), .Q(myout));
endmodule
```

delay9cycles.v

```
module delay9cycle(clock, reset, myin, myout);
  input clock, reset;
  input [63:0] myin;
  output [63:0] myout;

  wire [63:0] delay1, delay2, delay3, delay4, delay5, delay6, delay7, delay8;

  register64wide delayUnit1 (.clock(clock), .reset(reset), .D(myin), .Q(delay1));
  register64wide delayUnit2 (.clock(clock), .reset(reset), .D(delay1), .Q(delay2));
  register64wide delayUnit3 (.clock(clock), .reset(reset), .D(delay2), .Q(delay3));
  register64wide delayUnit4 (.clock(clock), .reset(reset), .D(delay3), .Q(delay4));
  register64wide delayUnit5 (.clock(clock), .reset(reset), .D(delay4), .Q(delay5));
  register64wide delayUnit6 (.clock(clock), .reset(reset), .D(delay5), .Q(delay6));
```

```
   register64wide delayUnit7 (.clock(clock), .reset(reset), .D(delay6), .Q(delay7));
   register64wide delayUnit8 (.clock(clock), .reset(reset), .D(delay7), .Q(delay8));
    register64wide delayUnit9 (.clock(clock), .reset(reset), .D(delay8), .Q(myout));
endmodule
```

register32wide.v

```
module register32wide(clock, reset, D, Q);
   input clock, reset;
   input [31:0] D;
   output [31:0] Q;

   reg [31:0] Q;

   always @(posedge clock or negedge reset) begin
                if (!reset) begin
                  Q <= 32'b0;
                end
                else begin
                  Q <= D;
                End
   end


endmodule
```

delay12cycle32.v

```
module delay12cycle32(clock, reset, myin, myout);
   input clock, reset;
   input [31:0] myin;
  output [31:0] myout;

   wire [31:0] delay1, delay2, delay3, delay4, delay5, delay6, delay7, delay8;
   wire [31:0] delay9, delay10, delay11;

  register32wide delayUnit1 (.clock(clock), .reset(reset), .D(myin), .Q(delay1));
  register32wide delayUnit2 (.clock(clock), .reset(reset), .D(delay1), .Q(delay2));
  register32wide delayUnit3 (.clock(clock), .reset(reset), .D(delay2), .Q(delay3));
  register32wide delayUnit4 (.clock(clock), .reset(reset), .D(delay3), .Q(delay4));
  register32wide delayUnit5 (.clock(clock), .reset(reset), .D(delay4), .Q(delay5));
  register32wide delayUnit6 (.clock(clock), .reset(reset), .D(delay5), .Q(delay6));
  register32wide delayUnit7 (.clock(clock), .reset(reset), .D(delay6), .Q(delay7));
  register32wide delayUnit8 (.clock(clock), .reset(reset), .D(delay7), .Q(delay8));
  register32wide delayUnit9 (.clock(clock), .reset(reset), .D(delay8), .Q(delay9));
  register32wide delayUnit10 (.clock(clock), .reset(reset), .D(delay9), .Q(delay10));
  register32wide delayUnit11 (.clock(clock), .reset(reset), .D(delay10), .Q(delay11));
  register32wide delayUnit12 (.clock(clock), .reset(reset), .D(delay11), .Q(myout));
```

```
endmodule
```

delay82cycle32.v

```
module delay82cycle32(clock, reset, myin, myout);
   input clock, reset;
   input [31:0] myin;
   output [31:0] myout;

         wire [31:0] delay1, delay2, delay3, delay4, delay5, delay6, delay7;

         delay12cycle32 delayUnit1 (.clock(clock), .reset(reset), .myin(myin), .myout(delay1));
         delay12cycle32 delayUnit2 (.clock(clock), .reset(reset), .myin(delay1), .myout(delay2));
         delay12cycle32 delayUnit3 (.clock(clock), .reset(reset), .myin(delay2), .myout(delay3));
         delay12cycle32 delayUnit4 (.clock(clock), .reset(reset), .myin(delay3), .myout(delay4));
         delay12cycle32 delayUnit5 (.clock(clock), .reset(reset), .myin(delay4), .myout(delay5));
         delay12cycle32 delayUnit6 (.clock(clock), .reset(reset), .myin(delay5), .myout(delay6));
         delay9cycle32 delayUnit7  (.clock(clock), .reset(reset), .myin(delay6), .myout(delay7));
         register32wide delayUnit8 (.clock(clock), .reset(reset), .D(delay7), .Q(myout));

endmodule
```

delay82cycle.v

```
module delay82cycle(clock, reset, myin, myout);
   input clock, reset;
   input [63:0] myin;
   output [63:0] myout;

   wire [63:0] delay1, delay2, delay3, delay4, delay5, delay6, delay7;

  delay12cycle delayUnit1 (.clock(clock), .reset(reset), .myin(myin), .myout(delay1));
  delay12cycle delayUnit2 (.clock(clock), .reset(reset), .myin(delay1), .myout(delay2));
  delay12cycle delayUnit3 (.clock(clock), .reset(reset), .myin(delay2), .myout(delay3));
  delay12cycle delayUnit4 (.clock(clock), .reset(reset), .myin(delay3), .myout(delay4));
  delay12cycle delayUnit5 (.clock(clock), .reset(reset), .myin(delay4), .myout(delay5));
  delay12cycle delayUnit6 (.clock(clock), .reset(reset), .myin(delay5), .myout(delay6));
  delay9cycle delayUnit7  (.clock(clock), .reset(reset), .myin(delay6), .myout(delay7));
  register64wide delayUnit8 (.clock(clock), .reset(reset), .D(delay7), .Q(myout));

endmodule
```

9.3.4 Error Module Files

errorModule.v

```
module errorModule(clock, reset, newData, zeroOut, calcError, deviation, bEntries, desiredError, errorReady,
criteriaMet);
   input clock, reset, zeroOut, newData;
   input calcError;
   input [63:0] deviation;
   input [63:0] bEntries;
   input [63:0] desiredError;
   output errorReady;
   output criteriaMet;

            wire [63:0] largestDev, largestB;
            wire [63:0] myError;
            wire [63:0] compareError;
            wire preErrorReady, preErrorReady2;
            wire clearRegs;

            reg preErrorReady3;
            reg errorReady, criteriaMet;
            reg [63:0] myErrorReg;

            assign clearRegs = reset && zeroOut;

            floatGreatestValue largestDeviation(.clock(clock), .newData(newData), .reset(clearRegs),
.myInput(deviation), .myOutput(largestDev));
            floatGreatestValue largestBEntry(.clock(clock), .newData(newData), .reset(clearRegs),
.myInput(bEntries), .myOutput(largestB));

            smalldivider64bit divideUnitb(.a(largestDev), .b(largestB), .operation_nd(calcError), .operation_rfd(),
           .clk(clock),
                                    .result(myError), .rdy(preErrorReady));
            float64subtract subUnit1(.a(myErrorReg), .b(desiredError), .clk(clock), .result(compareError));

            delay12cycle1Bit delayMe1A(.clock(clock), .reset(clearRegs), .myin(preErrorReady),
.myout(preErrorReady2));

            always@(posedge clock or negedge reset) begin
                  if(!reset)
                              myErrorReg <= 64'b0;
                  else if(preErrorReady==1)
                              myErrorReg <= myError;
            end

            always@(posedge clock or negedge reset) begin
```

```
		if(!reset) begin
			errorReady <= 0;
			criteriaMet <= 0;
			preErrorReady3 <= 0;
		end
		else begin
			preErrorReady3 <= preErrorReady2;
			errorReady <= preErrorReady3;
			if(compareError[63] && preErrorReady3)
				criteriaMet <= 1;
			else
				criteriaMet <= 0;
		end
	end

endmodule
```

floatGreatestValue.v

```
module floatGreatestValue(clock, newData, reset, myInput, myOutput);
  input clock, reset, newData;
  input [63:0] myInput;
	output [63:0] myOutput;

  wire out1, out2, out3, out4;
	wire thirdCycle;
	wire [63:0] regInput, delayed1Cycle, delayed2Cycle;
	wire [63:0] delayCompA, delayComp1B, delayComp2B;
	wire [63:0] compareUnit1Result, compareUnit2Result, delayComp1Result, delayComp2Result,
delayComp3Result, delayComp4Result;
	wire [63:0] compareUnit3Result, compareUnit4Result;
	wire rdy1, rdy2, rdy3, rdy4;

	reg [1:0] count1;
	reg compareEnable;
	reg [63:0] greatestValue;
	reg [63:0] compA, comp1B, comp2B;


	register64wide registerMyInput(.clock(clock), .reset(reset), .D(myInput), .Q(regInput));
	register64wide delayUnit1(.clock(clock), .reset(reset), .D(regInput), .Q(delayed1Cycle));
	register64wide delayUnit2(.clock(clock), .reset(reset), .D(delayed1Cycle), .Q(delayed2Cycle));
```

```
        float64greater compareUnit1(.a(compA), .operation_nd(compareEnable), .rdy(rdy1), .b(comp1B),
.clk(clock), .result(out1));
        float64greater compareUnit2(.a(compA), .operation_nd(compareEnable), .rdy(rdy2), .b(comp2B),
.clk(clock), .result(out2));

        delay3cycle delayMe1(.clock(clock), .reset(reset), .myInput(compA), .myOutput(delayCompA));
        delay3cycle delayMe2(.clock(clock), .reset(reset), .myInput(comp1B), .myOutput(delayComp1B));
        delay3cycle delayMe3(.clock(clock), .reset(reset), .myInput(comp2B), .myOutput(delayComp2B));
        delay3cycle delayMe4(.clock(clock), .reset(reset), .myInput(compareUnit1Result),
.myOutput(delayComp1Result));
        delay3cycle delayMe5(.clock(clock), .reset(reset), .myInput(compareUnit2Result),
.myOutput(delayComp2Result));
        delay3cycle delayMe6(.clock(clock), .reset(reset), .myInput(compareUnit3Result),
.myOutput(delayComp3Result));
        delay3cycle delayMe7(.clock(clock), .reset(reset), .myInput(compareUnit4Result),
.myOutput(delayComp4Result));

        assign compareUnit1Result = out1 ? delayCompA : delayComp1B;
        assign compareUnit2Result = out2 ? delayCompA : delayComp2B;
        assign compareUnit3Result = out3 ? delayComp1Result : delayComp2Result;
        assign compareUnit4Result = out4 ? delayComp3Result : delayComp4Result;
        assign myOutput = greatestValue;

        float64greater compareUnit3(.a(compareUnit1Result), .operation_nd(rdy1), .rdy(rdy3),
.b(compareUnit2Result), .clk(clock), .result(out3));
        float64greater compareUnit4(.a(compareUnit3Result), .operation_nd(rdy3), .rdy(rdy4),
.b(compareUnit4Result), .clk(clock), .result(out4));

        //main greatest value storage flip flop
        always@(posedge clock or negedge reset)
         if(!reset) begin
            greatestValue <= 0;
         end
         else begin
           if(rdy4) begin
                      greatestValue <= compareUnit4Result;
                    end
         end

        //registers with enables & reset
        always@(posedge clock or negedge reset)
         if(!reset) begin
                      compA <= 0;
                              comp1B <= 0;
                              comp2B <= 0;
                              compareEnable <= 0;
         end
         else begin
```

```
            if(thirdCycle) begin
                        compA <= regInput;
                        comp1B <= delayed1Cycle;
                        comp2B <= delayed2Cycle;
                        compareEnable <= 1;
                end
                else
                  compareEnable <= 0;
          end

          //counter
  always @(posedge clock or negedge reset) begin
   if (!reset)
     count1 <= 0;
   else begin
                  if (newData || (count1 !=0)) begin
                              if(count1 == 2'b11)
                                        count1 <= 2'b01;
                              else
                                        count1 <= count1 + 1;
                        end
                end
          end
           assign thirdCycle = (count1 == 2'b11);

endmodule
```

delay12cycle1Bit.v

```
module delay12cycle1Bit(clock, reset, myin, myout);
  input clock, reset;
  input  myin;
  output myout;

          reg delay1, delay2, delay3, delay4, delay5, delay6, delay7, delay8;
          reg delay9, delay10, delay11, delay12;

          always@(posedge clock or negedge reset) begin
                  if(!reset) begin
                              delay1 <= 0;
                              delay2 <= 0;
                              delay3 <= 0;
                              delay4 <= 0;
                              delay5 <= 0;
                              delay6 <= 0;
                              delay7 <= 0;
                              delay8 <= 0;
                              delay9 <= 0;
```

```
                delay10 <= 0;
                delay11 <= 0;
                delay12 <= 0;
            end
            else begin
                delay1 <= myin;
                delay2 <= delay1;
                delay3 <= delay2;
                delay4 <= delay3;
                delay5 <= delay4;
                delay6 <= delay5;
                delay7 <= delay6;
                delay8 <= delay7;
                delay9 <= delay8;
                delay10 <= delay9;
                delay11 <= delay10;
                delay12 <= delay11;
            end
        end

        assign myout = delay12;

endmodule
```

9.3.5 Iterative Controller Files

iterationController.v

```
module iterationController(clock, reset, newData, zeroOut, calcError, desiredError, errorReady, criteriaMet,
doneRow, fetchRow, enableMemAccess, resetMemAccess);
  input clock;
  input reset;
  output newData;
  output zeroOut;
  output calcError;
  output [63:0] desiredError;
  input errorReady;
  input criteriaMet;
  input [31:0] doneRow;
  output [31:0] fetchRow;
  output enableMemAccess;
  output resetMemAccess;

  reg newData, zeroOut, calcError, enableMemAccess, resetMemAccess;
        reg [31:0] fetchRow;
        reg [63:0] desiredError;
  reg [4:0] state, next_state;
        reg [7:0] timer1, timer2;
        reg [31:0] rowNumber;
        parameter S0=4'b0000, S1=4'b0001, S2=4'b0010, S3=4'b0011, S4=4'b0100, S5=4'b0101, S6=4'b0110,
S7=4'b0111, S8=4'b1000, S9=4'b1001, S10=4'b1010;
        reg resetValue, countUp;
        reg [31:0] numTotalRows;
        wire matrixEnd;
        reg timer1Enable, timer2Enable;
        wire timer1Expire, timer2Expire;

        always@(posedge clock)
          begin
                case(state)
                        S0: begin
                                if(!reset)
                                        next_state <= S1;
                                else
                                        next_state <= S0;
                                end
                        S1: begin
                                calcError <=0;
                                zeroOut <=0;
                                newData <=0;
                                enableMemAccess <=0;
                                resetMemAccess <=1;
```

```
                countUp <=1;
                resetValue <=0;
                next_state <= S2;
                timer1Enable <= 0;
                timer2Enable <= 0;
                end
S2: begin
                calcError <=0;
                zeroOut <=1;
                newData <=0;
                enableMemAccess <=1;
                resetMemAccess <=0;
                countUp <=1;
                resetValue <=0;
                timer1Enable <= 0;
 timer2Enable <= 0;

                next_state <= S3;
                end
S3: begin
                calcError <=0;
                zeroOut <=1;
                newData <=1;
                enableMemAccess <=1;
                resetMemAccess <=0;
                countUp <=1;
                resetValue <=0;
                timer1Enable <= 0;
                timer2Enable <= 0;

                if(matrixEnd && (doneRow==numTotalRows))
                        next_state <= S4;
                else
                        next_state <= S3;
                end
S4: begin
                calcError <= 0;
                zeroOut <=1;
                newData <=0;
                enableMemAccess <= 0;
                resetMemAccess <=0;
                countUp <=0;
   resetValue <=1;
                timer1Enable <= 1;
                timer2Enable <= 0;
                if(timer1Expire==1)
                        next_state <= S5;
                else
```

```
                                        next_state <= S4;
                                end
                        S5: begin
                                calcError <= 1;
                                zeroOut <=1;
                                newData <=0;
                                enableMemAccess <=0;
                                resetMemAccess <=0;
                                countUp <=0;
                                resetValue <=0;
                                timer1Enable <=0;
                                timer2Enable <=0;
                                if(errorReady==1) begin
                                        if(criteriaMet==1)
                                                next_state <=S10;
                                        else
                                                next_state <=S6;
                                        end
                                        else
                                                next_state <= S5;
                                end
                        S6: begin
                                calcError <= 0;
                                zeroOut <= 0;
                                newData <=0;
                                enableMemAccess <=0;
                                resetMemAccess <=1;
                                countUp <=0;
                                resetValue <=0;
                                timer1Enable <=0;
                                timer2Enable <=0;
                                next_state <= S1;
                                end
                        S10:    begin
                                  next_state <= S0;                     //Normally this state would
signal the computer
                                                                                                        //that we
are finished with matrix calculation, but since we have no interface tothe

        //computer set up, we simply reset our system for now
                                         end
                                default: next_state <= S0;
                        endcase
                end
        always@(posedge clock)
                if(!reset) begin
                        state <=0;
                        desiredError <= 64'h4014000000000000;
```

```
                numTotalRows <= 32'd5000;
        end
        else
                state <= next_state;


assign matrixEnd = (numTotalRows==rowNumber) ? 1 : 0;
assign timer1Expire = (timer1 == 8'd15);
assign timer2Expire = (timer2 == 8'd60);

always@(posedge clock)
        if(!reset)
                rowNumber <= 32'b0;
        else begin
                if(countUp)
                        rowNumber <= rowNumber +1;
                else
                        if(resetValue)
                                rowNumber <= 0;
 end

 always@(posedge clock)
  if(!reset)
                timer1 <= 8'b0;
        else begin
                if(timer1Enable)
                        timer1 <= timer1 + 1;
                else
                        timer1 <= 0;
        end

        always@(posedge clock)
  if(!reset)
                timer2 <= 8'b0;
        else begin
                if(timer2Enable)
                        timer2 <= timer1 + 1;
                else
                        timer2 <= 0;
        end


endmodule
```

9.3.6 Memory Interface Controller Files

memControl.v

```
module memControl(SYS_CLK_P, SYS_CLK_N, CLK200_P, CLK200_N, SYS_RESET_IN, SlowerClock,
 cntrl1_DDR2_RAS_N, cntrl1_DDR2_CAS_N, cntrl1_DDR2_WE_N, cntrl1_DDR2_CS_N, cntrl1_DDR2_ODT,
cntrl1_DDR2_CKE, cntrl1_DDR2_CK, cntrl1_DDR2_CK_N,
 cntrl1_DDR2_A, cntrl1_DDR2_BA, cntrl1_DDR2_DM, cntrl1_DDR2_DQ, cntrl1_DDR2_DQS,
cntrl1_DDR2_DQS_N,
 cntrl0_DDR2_RAS_N, cntrl0_DDR2_CAS_N, cntrl0_DDR2_WE_N, cntrl0_DDR2_CS_N, cntrl0_DDR2_ODT,
cntrl0_DDR2_CKE, cntrl0_DDR2_CK, cntrl0_DDR2_CK_N,
 cntrl0_DDR2_A, cntrl0_DDR2_BA, cntrl0_DDR2_DM, cntrl0_DDR2_DQ, cntrl0_DDR2_DQS,
cntrl0_DDR2_DQS_N, sramK, sramKbar, sramReadWrite, sramWriteEnable,
 sram1DQ, sram2DQ, sram3DQ, sram1SA, sram2SA, sram3SA, logicReset, sramAddressSource);

 // delay3forA1, delay3forA2, delay3forA3, delay3forA4, delay3forA5, delay3forA6, delay3forB, ranDataX1,
ranDataX2, ranDataX3, ranDataX4, ranDataX5, ranDataX6
input SYS_CLK_P;
input SYS_CLK_N;
input CLK200_P;
input CLK200_N;
input SYS_RESET_IN;
output SlowerClock;
output logicReset;

//SRAM Signals
inout [63:0] sram1DQ, sram2DQ, sram3DQ;
output [19:0] sram1SA, sram2SA, sram3SA;
output sramK, sramKbar;
output sramReadWrite, sramWriteEnable;    //sramReadWrite=0 means Read, 1=Write, but only if
sramWriteEnable is high
input sramAddressSource;              //0=internally generated (i.e. from col pointers), 1=comes from main
controller to write
wire [63:0] sram1DQ_IN, sram1DQ_OUT, sram2DQ_IN, sram2DQ_OUT, sram3DQ_IN, sram3DQ_OUT;


output cntrl1_DDR2_RAS_N;
output cntrl1_DDR2_CAS_N;
output cntrl1_DDR2_WE_N;
output cntrl1_DDR2_CS_N;
output cntrl1_DDR2_ODT;
output cntrl1_DDR2_CKE;
output cntrl1_DDR2_CK;
output cntrl1_DDR2_CK_N;
output [12:0] cntrl1_DDR2_A;
output [1:0] cntrl1_DDR2_BA;
output [1:0] cntrl1_DDR2_DM;
output [15:0] cntrl1_DDR2_DQ;
```

```
output [1:0] cntrl1_DDR2_DQS;
output [1:0] cntrl1_DDR2_DQS_N;

output cntrl0_DDR2_RAS_N;
output cntrl0_DDR2_CAS_N;
output cntrl0_DDR2_WE_N;
output cntrl0_DDR2_CS_N;
output cntrl0_DDR2_ODT;
output cntrl0_DDR2_CKE;
output [4:0] cntrl0_DDR2_CK;
output [4:0] cntrl0_DDR2_CK_N;
output [12:0] cntrl0_DDR2_A;
output [1:0] cntrl0_DDR2_BA;
output [9:0] cntrl0_DDR2_DM;
output [79:0] cntrl0_DDR2_DQ;
output [9:0] cntrl0_DDR2_DQS;
output [9:0] cntrl0_DDR2_DQS_N;

wire clocking_clk0_dcm, clocking_clk90_dcm, clocking_clkdiv2_dcm, clocking_dcm_lock,
clocking_REF_CLK200_IN;

wire cntrl0_CLK_TB, cntrl0_RESET_TB, cntrl0_WDF_ALMOST_FULL, cntrl0_AF_ALMOST_FULL,
cntrl0_READ_DATA_VALID, cntrl0_APP_WDF_WREN, cntrl0_APP_WAF_WREN;
wire[2:0] cntrl0_BURST_LENGTH;
wire [35:0] cntrl0_APP_WAF_ADDR;
wire [39:0] cntrl0_APP_MASK_DATA;
wire [79:0] cntrl0_READ_DATA0_FIFO_OUT, cntrl0_READ_DATA1_FIFO_OUT, cntrl0_READ_DATA2_FIFO_OUT,
cntrl0_READ_DATA3_FIFO_OUT;
wire [319:0] cntrl0_APP_WDF_DATA;
wire [319:0] cntrl0_APP_READ_DATA;
wire clock;

wire cntrl1_CLK_TB, cntrl1_RESET_TB, cntrl1_WDF_ALMOST_FULL, cntrl1_AF_ALMOST_FULL,
cntrl1_READ_DATA_VALID, cntrl1_APP_WDF_WREN, cntrl1_APP_WAF_WREN;
wire[2:0] cntrl1_BURST_LENGTH;
wire [35:0] cntrl1_APP_WAF_ADDR;
wire [39:0] cntrl1_APP_MASK_DATA;
wire [79:0] cntrl1_READ_DATA0_FIFO_OUT, cntrl1_READ_DATA1_FIFO_OUT, cntrl1_READ_DATA2_FIFO_OUT,
cntrl1_READ_DATA3_FIFO_OUT;
wire [319:0] cntrl1_APP_WDF_DATA;
wire [79:0] cntrl0_DATAIN_WORD0, cntrl0_DATAIN_WORD1, cntrl0_DATAIN_WORD2,
cntrl0_DATAIN_WORD3;
wire [159:0] cntrl0_LG_DATAIN_WORD0, cntrl0_LG_DATAIN_WORD1, cntrl0_LG_DATAIN_WORD2,
cntrl0_LG_DATAIN_WORD3;
reg [319:0] seqDataPhase0;
reg [127:0] seqDataPhase1a;
reg [191:0] seqDataPhase1b;
```

```
reg [63:0] seqDataPhase0RegA1, seqDataPhase0RegA2, seqDataPhase0RegA3, seqDataPhase0RegA4,
seqDataPhase0RegA5;
reg [63:0] seqDataPhase1RegA6, seqDataPhase1RegB;
reg [31:0] seqDataPhase1RegCol1, seqDataPhase1RegCol2, seqDataPhase1RegCol3, seqDataPhase1RegCol4,
seqDataPhase1RegCol5, seqDataPhase1RegCol6;
reg [63:0] ranDataX1, ranDataX2, ranDataX3, ranDataX4, ranDataX5, ranDataX6;
reg [63:0] delay1forA1, delay1forA2, delay1forA3, delay1forA4, delay1forA5, delay1forA6, delay1forB;
reg [63:0] delay2forA1, delay2forA2, delay2forA3, delay2forA4, delay2forA5, delay2forA6, delay2forB;
reg [63:0] delay3forA1, delay3forA2, delay3forA3, delay3forA4, delay3forA5, delay3forA6, delay3forB;

wire [19:0] sram1ReadAddressA, sram2ReadAddressA, sram3ReadAddressA;
wire [19:0] sram1ReadAddressB, sram2ReadAddressB, sram3ReadAddressB;
wire clockingMode;

assign sram1ReadAddressA = clockingMode ? (seqDataPhase1RegCol1[19:0]) : (seqDataPhase1RegCol2[19:0]);
assign sram2ReadAddressA = clockingMode ? (seqDataPhase1RegCol3[19:0]) : (seqDataPhase1RegCol4[19:0]);
assign sram3ReadAddressA = clockingMode ? (seqDataPhase1RegCol5[19:0]) : (seqDataPhase1RegCol6[19:0]);
assign sram1SA = sramAddressSource ? sram1ReadAddressA : sram1ReadAddressB;
assign sram2SA = sramAddressSource ? sram2ReadAddressA : sram2ReadAddressB;
assign sram3SA = sramAddressSource ? sram3ReadAddressA : sram3ReadAddressB;

assign sram1DQ = (sramAddressSource) ? 64'hzzzzzzzzzzzzzzzz : sram1DQ_IN;
assign sram2DQ = (sramAddressSource) ? 64'hzzzzzzzzzzzzzzzz : sram2DQ_IN;
assign sram3DQ = (sramAddressSource) ? 64'hzzzzzzzzzzzzzzzz : sram3DQ_IN;
assign sram1DQ_OUT = sram1DQ;
assign sram2DQ_OUT = sram2DQ;
assign sram3DQ_OUT = sram3DQ;

assign sramK = clocking_clk0_dcm;
assign clock = cntrl0_CLK_TB;
assign SlowerClock = clocking_clkdiv2_dcm;
assign cntrl0_APP_READ_DATA = {cntrl0_READ_DATA0_FIFO_OUT, cntrl0_READ_DATA1_FIFO_OUT,
cntrl0_READ_DATA2_FIFO_OUT, cntrl0_READ_DATA3_FIFO_OUT};
assign logicReset = cntrl0_RESET_TB;

always @(posedge clock or negedge cntrl0_RESET_TB) begin
    if (!cntrl0_RESET_TB) begin
      seqDataPhase0 = 320'b0;
    end
    else if (!clockingMode) begin
      seqDataPhase0 = cntrl0_APP_READ_DATA;
    end
end

always@(posedge clock or negedge cntrl0_RESET_TB) begin
          if(!cntrl0_RESET_TB) begin
                    delay1forA1 = 64'b0;
                    delay2forA1 = 64'b0;
```

```
                delay3forA1 = 64'b0;
                delay1forA2 = 64'b0;
                delay2forA2 = 64'b0;
                delay3forA2 = 64'b0;
                delay1forA3 = 64'b0;
                delay2forA3 = 64'b0;
                delay3forA3 = 64'b0;
                delay1forA4 = 64'b0;
                delay2forA4 = 64'b0;
                delay3forA4 = 64'b0;
                delay1forA5 = 64'b0;
                delay2forA5 = 64'b0;
                delay3forA5 = 64'b0;
                delay1forA6 = 64'b0;
                delay2forA6 = 64'b0;
                delay3forA6 = 64'b0;
                delay1forB = 64'b0;
                delay2forB = 64'b0;
                delay3forB = 64'b0;
        end
        else begin
                delay1forA1 = seqDataPhase0RegA1;
                delay1forA2 = seqDataPhase0RegA2;
                delay1forA3 = seqDataPhase0RegA3;
                delay1forA4 = seqDataPhase0RegA4;
                delay1forA5 = seqDataPhase0RegA5;
                delay1forA6 = seqDataPhase1RegA6;
                delay1forB = seqDataPhase1RegB;
                delay2forA1 = delay1forA1;
                delay2forA2 = delay1forA2;
                delay2forA3 = delay1forA3;
                delay2forA4 = delay1forA4;
                delay2forA5 = delay1forA5;
                delay2forA6 = delay1forA6;
                delay2forB = delay1forB;
                delay3forA1 = delay2forA1;
                delay3forA2 = delay2forA2;
                delay3forA3 = delay2forA3;
                delay3forA4 = delay2forA4;
                delay3forA5 = delay2forA5;
                delay3forA6 = delay2forA6;
                delay3forB = delay2forB;
        end
end

always@(posedge clock or negedge cntrl0_RESET_TB) begin
        if(!cntrl0_RESET_TB) begin
                seqDataPhase1a = 128'b0;
```

```
                seqDataPhase1b = 192'b0;
        end
        else if(clockingMode) begin
                seqDataPhase1a = {cntrl0_APP_READ_DATA[319:256], cntrl0_APP_READ_DATA[63:0]};
                seqDataPhase1b = cntrl0_APP_READ_DATA[255:64];
        end
end

always@(posedge clock or negedge cntrl0_RESET_TB) begin
          if(!cntrl0_RESET_TB) begin
                        seqDataPhase0RegA1 = 64'b0;
                        seqDataPhase0RegA2 = 64'b0;
                        seqDataPhase0RegA3 = 64'b0;
                        seqDataPhase0RegA4 = 64'b0;
                        seqDataPhase0RegA5 = 64'b0;
                end
                else if(!clockingMode) begin
                        seqDataPhase0RegA1 = seqDataPhase0[319:256];
                        seqDataPhase0RegA2 = seqDataPhase0[255:192];
                        seqDataPhase0RegA3 = seqDataPhase0[191:128];
                        seqDataPhase0RegA4 = seqDataPhase0[127:64];
                        seqDataPhase0RegA5 = seqDataPhase0[63:0];
                end
end

always@(posedge clock or negedge cntrl0_RESET_TB) begin
                if(!cntrl0_RESET_TB) begin
                        seqDataPhase1RegA6 = 64'b0;
                        seqDataPhase1RegB = 64'b0;
                        seqDataPhase1RegCol1 = 32'b0;
                        seqDataPhase1RegCol2 = 32'b0;
                        seqDataPhase1RegCol3 = 32'b0;
                        seqDataPhase1RegCol4 = 32'b0;
                        seqDataPhase1RegCol5 = 32'b0;
                        seqDataPhase1RegCol6 = 32'b0;
                end
                else if(clockingMode) begin
                        seqDataPhase1RegA6 = seqDataPhase1a[127:64];
                        seqDataPhase1RegB = seqDataPhase1a[63:0];
                        seqDataPhase1RegCol1 = seqDataPhase1b[191:160];
                        seqDataPhase1RegCol2 = seqDataPhase1b[159:128];
                        seqDataPhase1RegCol3 = seqDataPhase1b[127:96];
                        seqDataPhase1RegCol4 = seqDataPhase1b[95:64];
                        seqDataPhase1RegCol5 = seqDataPhase1b[63:32];
                        seqDataPhase1RegCol6 = seqDataPhase1b[31:0];
                end
end
```

```
always @(posedge clock or negedge cntrl0_RESET_TB) begin
    if (!cntrl0_RESET_TB) begin
      ranDataX1 = 64'b0;
                              ranDataX2 = 64'b0;
                              ranDataX3 = 64'b0;
                              ranDataX4 = 64'b0;
                              ranDataX5 = 64'b0;
                              ranDataX6 = 64'b0;
    end
    else begin
                              if (!clockingMode) begin
                                        ranDataX2 = sram1DQ_OUT;
                                        ranDataX4 = sram2DQ_OUT;
                                        ranDataX6 = sram3DQ_OUT;
                              end
                              if (clockingMode) begin
                                        ranDataX1 = sram1DQ_OUT;
                                        ranDataX3 = sram2DQ_OUT;
                                        ranDataX5 = sram3DQ_OUT;
                              end
                    end
end

ddr2controller sequentialDataController (
   .cntrl0_DDR2_DQ(cntrl0_DDR2_DQ),
   .cntrl0_DDR2_A(cntrl0_DDR2_A),
   .cntrl0_DDR2_BA(cntrl0_DDR2_BA),
   .cntrl0_DDR2_RAS_N(cntrl0_DDR2_RAS_N),
   .cntrl0_DDR2_CAS_N(cntrl0_DDR2_CAS_N),
   .cntrl0_DDR2_WE_N(cntrl0_DDR2_WE_N),
   .cntrl0_DDR2_CS_N(cntrl0_DDR2_CS_N),
   .cntrl0_DDR2_ODT(cntrl0_DDR2_ODT),
   .cntrl0_DDR2_CKE(cntrl0_DDR2_CKE),
   .cntrl0_DDR2_DM(cntrl0_DDR2_DM),
   .SYS_CLK_P(SYS_CLK_P),
   .SYS_CLK_N(SYS_CLK_N),
   .CLK200_P(CLK200_P),
   .CLK200_N(CLK200_N),
   .SYS_RESET_IN(SYS_RESET_IN),
   .cntrl0_CLK_TB(cntrl0_CLK_TB),
   .cntrl0_RESET_TB(cntrl0_RESET_TB),
   .cntrl0_WDF_ALMOST_FULL(cntrl0_WDF_ALMOST_FULL),
   .cntrl0_AF_ALMOST_FULL(cntrl0_AF_ALMOST_FULL),
   .cntrl0_READ_DATA_VALID(cntrl0_READ_DATA_VALID),
   .cntrl0_APP_WDF_WREN(cntrl0_APP_WDF_WREN),
   .cntrl0_APP_WAF_WREN(cntrl0_APP_WAF_WREN),
   .cntrl0_BURST_LENGTH(cntrl0_BURST_LENGTH),
   .cntrl0_APP_WAF_ADDR(cntrl0_APP_WAF_ADDR),
```

```
   .cntrl0_READ_DATA0_FIFO_OUT(cntrl0_READ_DATA0_FIFO_OUT),
   .cntrl0_READ_DATA1_FIFO_OUT(cntrl0_READ_DATA1_FIFO_OUT),
   .cntrl0_READ_DATA2_FIFO_OUT(cntrl0_READ_DATA2_FIFO_OUT),
   .cntrl0_READ_DATA3_FIFO_OUT(cntrl0_READ_DATA3_FIFO_OUT),
   .cntrl0_APP_WDF_DATA(cntrl0_APP_WDF_DATA),
   .cntrl0_APP_MASK_DATA(cntrl0_APP_MASK_DATA),
   .cntrl0_DDR2_DQS(cntrl0_DDR2_DQS),
   .cntrl0_DDR2_DQS_N(cntrl0_DDR2_DQS_N),
   .cntrl0_DDR2_CK(cntrl0_DDR2_CK),
   .cntrl0_DDR2_CK_N(cntrl0_DDR2_CK_N),
   .clocking_clk0_dcm(clocking_clk0_dcm),
   .clocking_clk90_dcm(clocking_clk90_dcm),
   .clocking_clkdiv2_dcm(clocking_clkdiv2_dcm),
   .clocking_dcm_lock(clocking_dcm_lock),
   .clocking_REF_CLK200_IN(clocking_REF_CLK200_IN),
            .clocking_clk180_dcm(sramKbar)
   );

mem_interface_top resultStorageController (
   .cntrl0_DDR2_DQ(cntrl1_DDR2_DQ),
   .cntrl0_DDR2_A(cntrl1_DDR2_A),
   .cntrl0_DDR2_BA(cntrl1_DDR2_BA),
   .cntrl0_DDR2_RAS_N(cntrl1_DDR2_RAS_N),
   .cntrl0_DDR2_CAS_N(cntrl1_DDR2_CAS_N),
   .cntrl0_DDR2_WE_N(cntrl1_DDR2_WE_N),
   .cntrl0_DDR2_CS_N(cntrl1_DDR2_CS_N),
   .cntrl0_DDR2_ODT(cntrl1_DDR2_ODT),
   .cntrl0_DDR2_CKE(cntrl1_DDR2_CKE),
   .cntrl0_DDR2_DM(cntrl1_DDR2_DM),
   .SYS_RESET_IN(SYS_RESET_IN),
   .cntrl0_CLK_TB(cntrl1_CLK_TB),
   .cntrl0_RESET_TB(cntrl1_RESET_TB),
   .cntrl0_WDF_ALMOST_FULL(cntrl1_WDF_ALMOST_FULL),
   .cntrl0_AF_ALMOST_FULL(cntrl1_AF_ALMOST_FULL),
   .cntrl0_READ_DATA_VALID(cntrl1_READ_DATA_VALID),
   .cntrl0_APP_WDF_WREN(cntrl1_APP_WDF_WREN),
   .cntrl0_APP_WAF_WREN(cntrl1_APP_WAF_WREN),
   .cntrl0_BURST_LENGTH(cntrl1_BURST_LENGTH),
   .cntrl0_APP_WAF_ADDR(cntrl1_APP_WAF_ADDR),
   .cntrl0_READ_DATA0_FIFO_OUT(cntrl1_READ_DATA0_FIFO_OUT),
   .cntrl0_READ_DATA1_FIFO_OUT(cntrl1_READ_DATA1_FIFO_OUT),
   .cntrl0_READ_DATA2_FIFO_OUT(cntrl1_READ_DATA2_FIFO_OUT),
   .cntrl0_READ_DATA3_FIFO_OUT(cntrl1_READ_DATA3_FIFO_OUT),
   .cntrl0_APP_WDF_DATA(cntrl1_APP_WDF_DATA),
   .cntrl0_APP_MASK_DATA(cntrl1_APP_MASK_DATA),
   .cntrl0_DDR2_DQS(cntrl1_DDR2_DQS),
   .cntrl0_DDR2_DQS_N(cntrl1_DDR2_DQS_N),
   .cntrl0_DDR2_CK(cntrl1_DDR2_CK),
```

```
   .cntrl0_DDR2_CK_N(cntrl1_DDR2_CK_N),
   .clocking_clk0_dcm(clocking_clk0_dcm),
   .clocking_clk90_dcm(clocking_clk90_dcm),
   .clocking_clkdiv2_dcm(clocking_clkdiv2_dcm),
   .clocking_dcm_lock(clocking_dcm_lock),
   .clocking_REF_CLK200_IN(clocking_REF_CLK200_IN)
   );

Endmodule
```

multiChipSync.v

```
module multiChipSync(clock, reset, numRows, startSync, myChipID, maxChips, token, transmitID, rowNumber,
dataValue, sramAddress, dataToSRAM, dataFromSRAM);
   input clock;
   input reset;
   input startSync;
   input [3:0] myChipID;
   input [3:0] maxChips;
   output [3:0] token;
   inout [3:0] transmitID;
            input [31:0] numRows;
   inout [31:0] rowNumber;
   inout [63:0] dataValue;
   output [27:0] sramAddress;
   output [63:0] dataToSRAM;
   input [63:0] dataFromSRAM;

            reg [31:0] rowsPerChip;
            reg [31:0] myCount;
            reg [3:0] state;
            reg [3:0] tokenOut;
            reg incToken;
            reg [3:0] next_state;

            wire countUp;
            wire dataDirection;          //determines whether writing the incoming data to sram or reading from
0=output, 1=input

            always@(posedge clock or negedge reset)
            begin
                  if(!reset) begin
                                    rowsPerChip <= 32'b0;
```

```
                        state <= 4'b0;
                end
                else begin
                        rowsPerChip <= (numRows / maxChips);
                        state <= next_state;
                end

        always@(posedge clock or negedge reset)
        begin
                if(!reset) begin
                        myCount <= 32'b0;
                end
                else
                        begin
                        if(countUp==1)
                                myCount <= myCount +1;
                        end
        end

        always@(posedge clock or negedge reset)
begin
                if(!reset) begin
                        tokenOut <= 4'b0;
                end
                else
                        begin
                                if(incToken==1)
                                        tokenOut <= tokenOut+1;
                        end
        end
                case(state)
                        4'b0000: begin
                                        token <= 4'b0000;
                                                dataValue <= 64'b0;
                                                sramAddress <= 28'b0;
                                                dataToSRAM <= 64'b0;
                                                next_state <= 4'b0001;
                                        end
                 4'b0001:       begin
                                    if(startSync==1)
                                                        next_state <= 4'b0010;
                                                else
                                                        next_state <= 4'b0001;
                                                end
                 4'b0010:  begin
                                if(myChipID==4'b0000)
                                                  begin
                                                                token <= 4'b0000;
```

```
                                        sramAddress = 28'b0;
                                        dataDirection <= 0;
                                        next_state <= 4'b0011;
                                        countUp <= 1;
                                    end
                                else begin
                                        dataDirection <= 1;
                                        next_state <= 4'b1000;
                                end
                4'b0011 : begin
                                rowNumber <= myCount;
                                dataValue <= dataFromSRAM;
                                transmitID <= 4'b0000;
                                countUp <= 1;
                                if(myCount == rowsPerChip)
                                        next_state <= 4'b0100;
                                else
                                        next_state <= 4'b0011;
                                end
                4'b0100:   begin
                                incToken <= 1;
                                next_state <= 4'b0101;
                                end
                4'b0101:   begin
                                if(rowNumber == (token*rowsPerChip))
                                        incToken <= 1;
                                else
                                        incToken <= 0;
                                token <= tokenOut;
                                if(token==maxChips)
                                        next_state <= 4'b0110;
                                else
                                        next_state <= 4'b0101;
                                sramAddress <= rowNumber;
                                dataToSRAM <= dataValue;
                                end
        4'b0110 :    begin
                                //sync complete, go back to state 0
                                next_state <= 4'b0000;
                                end

endmodule
```

9.4 Xilinx Timing Reports/Device Utilization Reports

This section shows the most important part of the synthesis report for this project. There is a device utilization report giving how much of all the FPGAs resources are used for each side. The timing information is split into two sides, the processing side of the FPGA and the memory Interface side. This was necessary in the tools because of the different clocks driving these two parts of the circuit.

9.4.1 Timing report for Processing side of FPGA (Calculation module, error module, controller…)

Timing Summary:

Speed Grade: -11

Minimum period: 5.623ns (Maximum Frequency: 177.841MHz)
Minimum input arrival time before clock: 3.073ns
Maximum output required time after clock: 5.066ns
Maximum combinational path delay: 5.775ns

9.4.2 .Timing report for memory Interface portion of FPGA

Timing Summary:

Speed Grade: -11

Minimum period: 3.297ns (Maximum Frequency: 303.275MHz)
Minimum input arrival time before clock: No path found
Maximum output required time after clock: 5.050ns
Maximum combinational path delay: 6.093ns

9.4.3 Device Utilization Report for Memory Interface portion of design

```
=========================================================================
*                      Final Report                     *
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 4vlx80ff1148-11

 Number of Slices:               5216  out of  35840    14%
 Number of Slice Flip Flops:     4247  out of  71680     5%
 Number of 4 input LUTs:         9222  out of  71680    12%
   Number used as logic:         8438
   Number used as Shift registers:    16
   Number used as RAMs:          768
 Number of IOs:                  450
 Number of bonded IOBs:          448  out of    768    58%
   IOB Flip Flops:               36
 Number of FIFO16/RAMB16s:       14  out of    200     7%
   Number used as FIFO16s:       14
 Number of GCLKs:                14  out of     32    43%
 Number of BUFIOs:               12  out of     52    23%
 Number of DCM_ADVs:              3  out of     12    25%
 Number of ISERDESs:             96  out of    768    12%
 Number of OSERDESs:             132  out of    768    17%
 Number of PMCDs:                2  out of      8    25%
```

9.4.4 Device Utilization for Processing portion of design

```
=========================================================================

=========================================================================
*                      Final Report                     *
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 4vlx80ff1148-11

 Number of Slices:               18857  out of  35840    52%
 Number of Slice Flip Flops:     29314  out of  71680    40%
 Number of 4 input LUTs:         24120  out of  71680    33%
   Number used as logic:         20719
```

```
   Number used as Shift registers:   3401
Number of IOs:                       915
Number of bonded IOBs:                38  out of    768     4%
Number of FIFO16/RAMB16s:              1  out of    200     0%
   Number used as RAMB16s:             1
Number of GCLKs:                      2  out of     32     6%
Number of DSP48s:                     54  out of     80    67%


=========================================================================
```