AN IMPLEMENTATION OF A THREE DIMENSIONAL COMPUTATIONAL PIPELINE WITH MINIMAL LATENCY AND MAXIMUM THROUGHPUT FOR LU FACTORIZATION USING FIELD PROGRAMMABLE GATE ARRAYS

by

Edward Thomas Henciak

B.S.E.E, University of Pittsburgh, 1998

Submitted to the Graduate Faculty of

The Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

University of Pittsburgh

2008

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Edward Thomas Henciak

It was defended on

April 4, 2008

and approved by

Dr. Ronald G. Hoelzeman, Associate Professor, Department of Electrical and Computer Engineering

Dr. J. Robert Boston, Professor, Department of Electrical and Computer Engineering

Thesis Co-advisor: Dr. James T. Cain, Professor, Department of Electrical and Computer Engineering

Thesis Co-advisor: Dr. Marlin Mickle, Professor, Department of Electrical and Computer Engineering

AN IMPLEMENTATION OF A THREE DIMENSIONAL COMPUTATIONAL PIPELINE WITH MINIMAL LATENCY AND MAXIMUM THROUGHPUT FOR LU FACTORIZATION USING FIELD PROGRAMMABLE GATE ARRAYS

Edward Thomas Henciak, M.S.

University of Pittsburgh, 2008

Traditionally, computationally intense algebraic functions such as LU factorization are solved using complex systems such as supercomputers, parallel processing systems, and non-dedicated computing clusters. While these solutions are adequate for some problems, they typically suffer from classic parallel processing issues such as communication overhead, complex scheduling algorithms, and cost. Moreover, they are not feasible for embedded applications.

Extremely high performance solutions are sometimes implemented using costly, custom hardware such as Application Specific Integrated Circuits (ASICs). Unfortunately, the design, implementation, and verification of ASICs has become cost prohibitive and such solutions are only feasible if the end design is to be manufactured in very high volumes. As a result, many proposed architectures to solve specific problems lie dormant because they are simply too expensive to realize.

In recent years, advancements in Field Programmable Gate Array (FPGA) technology allow engineers to map complex algorithms to logic gates while achieving performance similar to ASIC technology. This thesis demonstrates the feasibility of the implementation of a three dimensional pipeline designed to solve LU factorization using FPGAs based on an architecture

proposed nearly 10 years ago when a technology to implement such an architecture either did not exist or was too costly to implement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 OVERVIEW OF THE PROBLEM

As demand for computational performance increases, new technologies and data processing architectures are required. While traditional, programmable CPUs obviously work extremely well for many real world applications, some problems are only “realistically” solved using custom hardware solutions.

A programmable CPU is not designed to be the optimal solution for *application specific* problems but rather flexibility. Programmable CPUs, at best, can only emulate what an application specific solution could provide without the resulting performance gains. As a result, engineers develop custom hardware solutions for these application specific problems when warranted.

Unfortunately, custom hardware solutions tend to have problems of their own. A countless number of custom computing architectures have been proposed over the years. However, the underlying technologies required to implement these computing architectures are either too primitive, cost prohibitive or simply do not exist at the time they are devised. As a result, most of these architectures lie dormant simply waiting to be realized so that their full potential can be unleashed.

Fortunately, the past ten years has seen explosive growth in Field Programmable Gate Array (FPGA) technology. In the past, the need for high speed digital circuits typically translated into a requirement for an Application Specific Integrated Circuit (ASIC). Not only do ASICs have significant costs in terms of manufacturing, they also require a significant amount of manpower for realization of the device. Modern ASICs require teams of engineers to design, implement, and verify a complex digital system.

FPGAs lighten this manpower requirement in that they are reconfigurable computing platforms: they do not require the verification efforts ASICs require nor do they require the up front costs involved with ASIC manufacturing. If a problem is discovered in an FPGA design, the problem can be fixed without having to remanufacture the device.

More importantly, FPGAs have achieved a level of performance on par with ASICs. For example, the Xilinx Virtex 5 SXT device provides the end user with DSP MAC functions capable of running up to 550MHz as well as high speed serial transceivers running at 3.2Gbps [1]. Similar performance is available by competing products from other FPGA vendors such as Altera and Lattice Semiconductor. The end result is that many custom, high performance computing platforms can now be realized with little risk and lower cost.

Despite this technology providing vast, new computing resources, skeptics may still question the overall feasibility of such a technology. As a result, this thesis aims to demonstrate the application of FPGAs on an architecture proposed almost ten years ago [1] when a technology for implementing the proposed architecture either did not exist, or existed but was far too cost prohibitive to implement at the time. More importantly, this particular application and architecture has significant potential for use in modern, real time, embedded systems.

1.2 THESIS OBJECTIVE

Solutions to complex mathematical problems have always been a driving force behind the explosive growth in computing platforms. Obviously, as more computation power is available to end users, more complex problems can be solved. However, not all of these problems are best solved on traditional, programmable CPUs. Some problems, especially those that require solutions in real time, are best solved using other means.

This thesis examines the problem of a complex mathematical problem, LU factorization, as well computational solutions that exist for this problem. Subsequently, this thesis also demonstrates the implementation of a proposed, high performance architecture that solves the LU factorization problem using a 3D systolic array in a modern FPGA device.

In the past, solutions that previously relied on parallel, Von Neumann processing systems were typically implemented on distributed computing platforms where a mathematical problem is decomposed into several smaller tasks. These smaller tasks are subsequently assigned to various processing elements available in the system. Once the processing elements have completed their task, the results are reported back to a host. Once the host assembles the results, a final solution is reported and the system is free to work on the next problem.

Problems exist with distributed computing methodology with the first being communication bottlenecks. Typically, distributed systems of the past tend to use slower communications protocols relative to their potential computing power. For example, a modern CPU might appear to have more than enough processing bandwidth to solve the partial solutions of any given parallel processing problem. However, the communications bandwidth available to the system might restrict the usefulness of the CPU.

Next, distributed systems tend to suffer from communication overhead. For example, a typical Ethernet packet contains several bytes of header information and requires several bus transactions with the Ethernet peripheral interface on the system bus. The end result is degraded performance due to the communication link since the Ethernet peripheral is typically running at a speed far less than the processing power of the CPU. Moreover, the bus where the peripheral resides is typically shared with other peripherals thus reducing overall system bandwidth.

Finally, the linear CPUs in the distributed systems typically execute some kind of operating system. Some of the processor's time is interrupted to service other tasks required to maintain CPU operation. While most of these tasks are taken into account when evaluating communication performance, some potential performance is lost simply due to the operating system requirement.

Overall, the objective of this thesis is to demonstrate that a radically different solution to the LU factorization problem can be implemented using modern, custom hardware technology. Instead of mapping the LU factorization algorithm to a distributed, linear computing system, it is now possible to create a custom hardware pipeline using FPGA technology. Not only can this architecture be implemented, an LU factorization problem can be solved in an optimal number of cycles.

In the past, this solution was not feasible given the low gate densities of FPGAs, but modern FPGA devices are well suited for the computationally intense task of LU factorization. This thesis demonstrates the implementation of an algorithm proposed by Dr. Joann Paul and Dr. Marlin Mickle of the University of Pittsburgh [1] that is virtually impossible to implement with significant performance gains on linear CPUs, but is straightforward to implement on a modern FPGA, and discusses performance compared to distributed, linear CPUs as well as other LU factorization solutions.

1.3 OUTLINE OF THE THESIS

Chapter 2 of this thesis reviews literature leading to the support of FPGA technology as a viable candidate for implementing the massively parallel processing solution the 3D Pipeline for LU factorization requires. It is intended to give the reader an idea of various solutions that have been explored over the past 20 years. Chapter 3 defines the problem and details the approach used to solve the problem using FPGA technology. Chapter 4 details the results of the work described and outlined in Chapter 3. Finally, Chapter 5 presents conclusions and details potential future work.

2.0 THE LU FACTORIZATION PROBLEM ON COMPUTING PLATFORMS

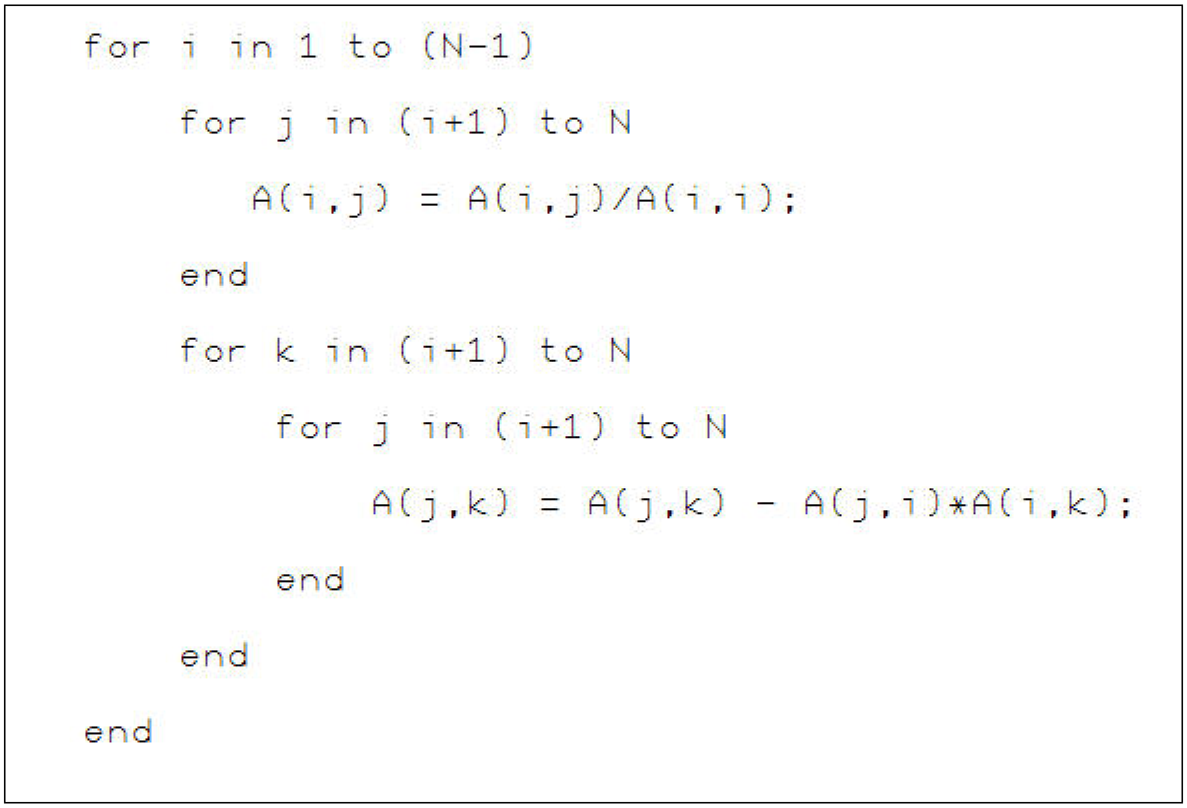
LU factorization of matrices is computationally intense. The process is highly iterative in nature when implemented on a traditional, Von Neumann central processing unit. Consider the following mathematical representation of LU factorization.

$$t_{ij} = \frac{1}{s_{ii}}\left(a_{ij} - \sum_{k=1}^{i-1} s_{ik} * t_{kj}\right)$$

$$s_{ij} = a_{ij} - \sum_{k=1}^{j-1} s_{ik} * t_{kj}.$$

Figure 1: Mathematical Representation of LU Factorization as detailed in [1]

Overall, the following mathematical function can be represented in computer wording using the following pseudo code. The variable N is equal to the size of the matrix (i.e. if N = 4, then the matrix is 4x4).



```
for i in 1 to (N-1)
    for j in (i+1) to N
       A(i,j) = A(i,j)/A(i,i);
    end
    for k in (i+1) to N
        for j in (i+1) to N
            A(j,k) = A(j,k) - A(j,i)*A(i,k);
        end
    end
end
```

Figure 2: LU Factorization (Crout Elimination) pseudocode

The end result of this procedure is a square matrix with the resulting s and t matrices both above and below the diagonal. The s and t matrices are subsequently used to solve sets of linear simultaneous equations through back substitution. Unfortunately, the above algorithm requires a substantial number of cycles to process matrices on traditional CPUs. Each multiply, subtract, and divide requires time. The amount of time is variable depending on the precision of the data being processed. The loops also introduce conditionals which require the CPU to execute many compare operations to determine if branching is required.

However, the potential for parallelism exists in the algorithm and researchers have exploited this parallelism in various projects. This chapter aims to review the work that has been done with implementing parallel solutions to address the LU factorization problem. Moreover, this chapter demonstrates how this parallelism has migrated from banks of CPUs to single, programmable logic devices over the past two decades.

2.1 PARALLEL PROCESSING SYSTEM WITH SHARED MEMORY

One potential application for LU factorization algorithms is circuit simulation. Back in 1988, researchers at the Southern Methodist University as well as the University of Wisconsin examined the use of a Sequent Balance 21000 computing cluster for circuit simulation [2]. This particular system uses up to 10 processors running Unix with a single, shared memory. Overall, various circuits are used to evaluate the performance of the system.

The system is responsible for assembling a solution matrix based on the circuit that requires simulation. LU Factorization is used to solve this solution matrix. Once the solution matrix is created, a data structure (see Figure 3) is created in memory that represents the diagonal entries vector (DIAG), nonzero off-diagonal entries subscripts (NZSUB), the nonzero upper triangular entries vector row by row (UNZ), the nonzero lower triangular entries vector column by column (LNZ), and the index of the start of nonzero values in each column/row of LNZ/UNZ and NZSUB.

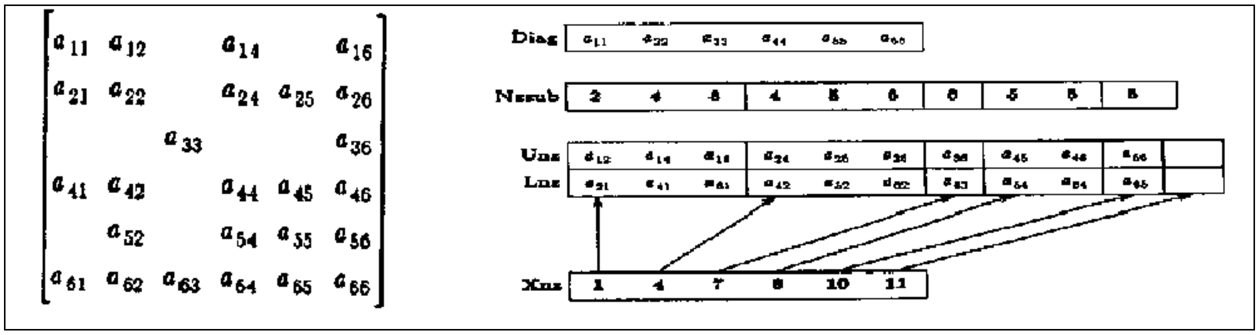


Figure 3: Example matrix from the Chen-Hu architecture and resulting data structure

Once the data structure is assembled, the CPU must now label the tasks required to solve the matrix and map these tasks to various processors in the system. It is important to note that, even at this point, the solution of the matrix has not commenced! The data structure described in Figure 3 only facilitates scheduling of tasks that are candidates for parallel execution. After the graph of these tasks is created, the scheduler can now instruct the processors to begin computing the final solution.

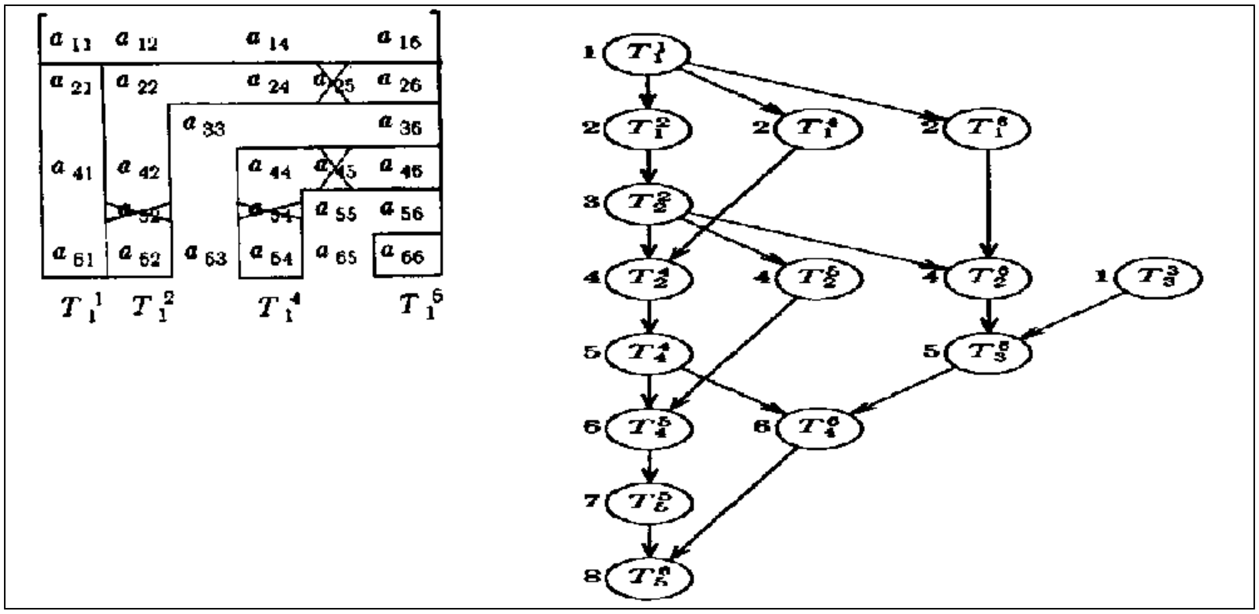


Figure 4: Task model and resulting task graph that is mapped to processors in the system.

Overall, the researchers found significant performance increase when mapping the LU factorization algorithm to parallel processing elements. For example, in the case of a 150 node simulation, performance was increased by a factor of 6.5 when using a 10 processor system when compared to a sequentially executing algorithm. The same algorithm running on a five processor configuration yielded an improvement factor of 4.16.

While a performance increase is observed, it is important to consider all the tasks involved to solve the matrix. To recap, the system must assemble the data structure based on the matrix, assign task clusters to processors, label each task in the task graph, and form a task queue. This is a significant amount of overhead. Moreover, the researchers admit that "the critical path scheduling, in this case, will require a large amount of synchronizing semaphores and hence result in large overhead." [2] The system suffers from other communication overhead such as interrupts, memory contention, and indirect accesses to nonzero entries. Finally, given the shared memory architecture of the system, it is virtually impossible to pipeline seeing that a new solution cannot commence until the previous solution has been solved. Throughput is rather low in such a system for these reasons.

2.2 PARALLEL PROCESSING WITH DISTRIBUTED MEMORY

The solution described in the previous section suffered from a serious bottleneck: memory contention and communication issues. Of course, alleviating these problems will yield better system performance. In 1992, researchers at the Supercomputing Research Center in Bowie, MD created a system to solve LU factorization on a MasPar MP-1 parallel processing system [3]. Using this system, a new way of mapping data and computations to processors is used, and reasonable processor utilization is obtained even for “unstructured” sparse matrices. The sparse problem is decomposed into many smaller, dense sub-problems, with low overhead for communications and memory access.

The MP-1 is a distributed memory system. Overall, this system alleviates one of the bottlenecks seen in the previous chapter: shared memory. Instead of multiple CPUs competing for memory access, each CPU uses its own memory space. Of course, some communication bandwidth is required to load each processor’s shared memory; however, this bandwidth is virtually negligible when compared with the daunting task of LU factorization.

The processor array in the MP-1 is a rectangular array. Each processor has a low latency link to communicate with its eight nearest neighbors as well as a broadcast function to send specified row (column) elements across the entire row (column) [3]. This “mesh based” communication is available to drastically reduce any inter-processor communication. Obviously, so long as a processor is communicating with a nearest neighbor processor, communication overhead is kept to a minimum. In the event a processor needs to communicate with a non-neighbor, a router is available, but the communication cost associated with the router is higher than the local link.

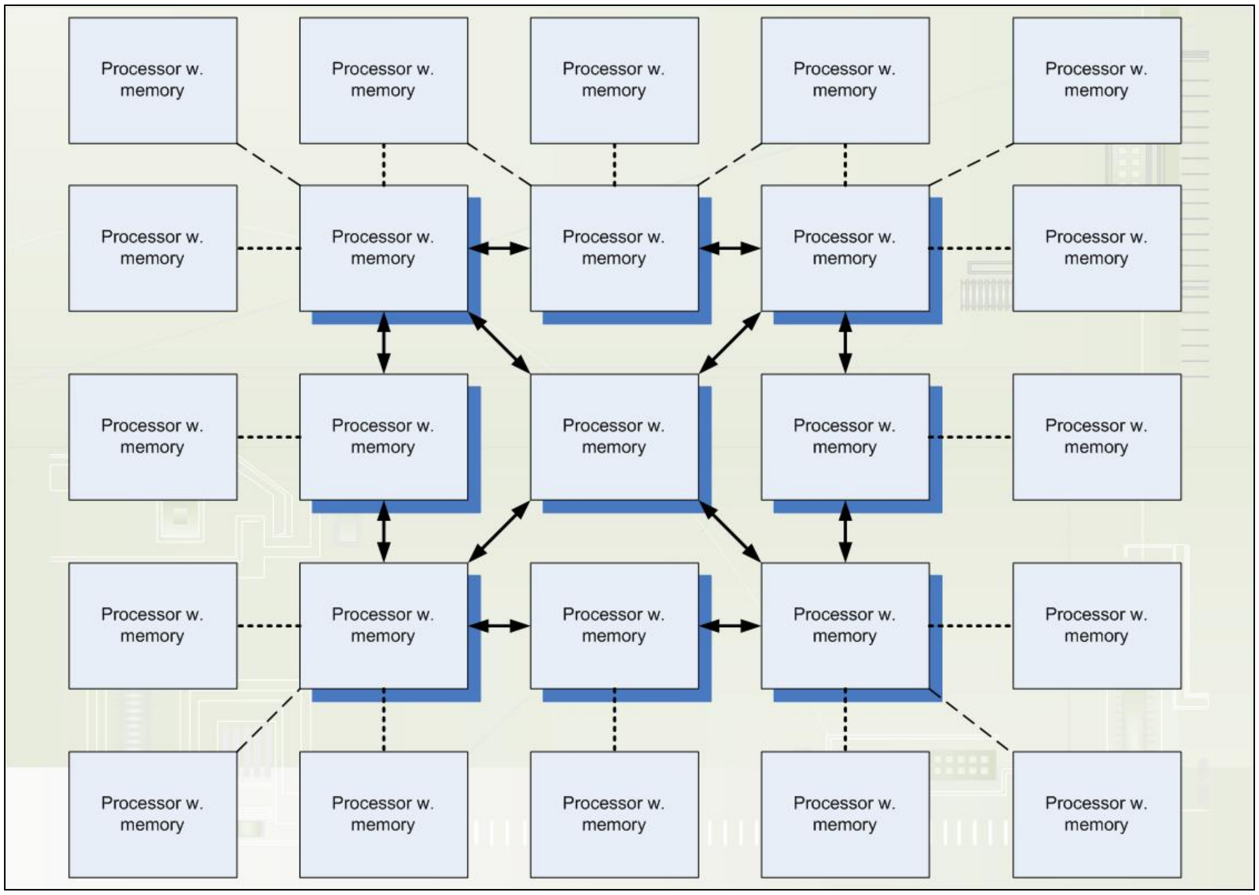


Figure 5: Rectangular processing array

Similar to the architecture discussed in the previous section, the system requires the LU Factorization problem to be broken down into a graph. In the case of this research, the tasks are mapped to an elimination tree. Moreover, tasks are mapped such that all inter-processor communication requirements are constrained to the high speed nearest neighbor and do not require use of the router mechanism.

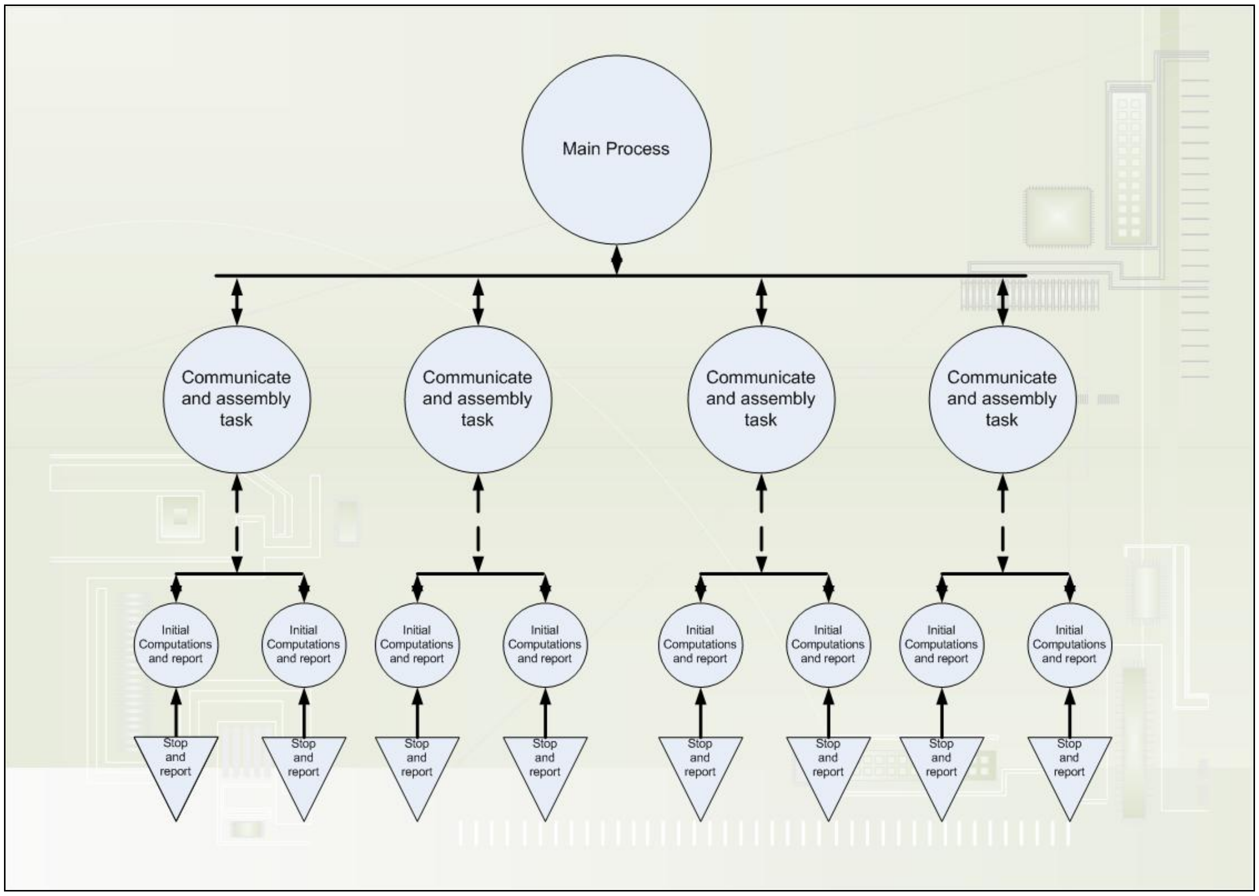

Main Process
Communicate and assembly task
Communicate and assembly task
Communicate and assembly task
Communicate and assembly task
Initial Computations and report
Stop and report

Figure 6: Tree example

In the tree example seen in Figure 6, one processor is responsible for decomposing the matrix into subtasks. Navigating down the tree, these tasks are assigned to processors in the rectangular processing array. In LU factorization, some results are required from other tasks before computation can commence. These required computations as well as matrix elements are forwarded down the tree. Eventually, the tasks are decomposed such that a processor is able to perform computations that are not dependent on previous solutions being available. As a result, these child tasks stop processing and begin reporting their results back to their parent tasks. As the solutions begin navigating up the tree, the intermediate calculations are performed until all results are communicated to the main process. At this point, the main task has the solution to the factorization.

The researchers indicate that their method was tested on a 64x64 processor array. Overall, they indicate that the decomposition function requires very little overhead and that most of the CPU cycles are spent on LU Factorization and not communication requirements thanks to the local link feature of the grid architecture. However, similar to the architecture in the previous section, a major problem is throughput. Again, despite the distributed nature of this system, a new matrix cannot be processed until the previous matrix is complete. Moreover, this architecture requires a rather large number of general purpose processors, 4096 to be exact. As a result, this approach would require extensive cost to develop.

2.3 NETWORK DISTRIBUTED CLUSTER PROCESSING

The previous research discussed involved solutions to LU factorization using static sized processing arrays. In 2004, research in parallel processing performed by researchers at the University of Tokyo yielded high performance solutions to LU factorization on non-dedicated computing clusters where the number of available computing resources may be arbitrary and even dynamically changing. The researchers observed 130Gflops with 128 processes running on a 70-node, dual 2.4GHz Intel Xeon cluster with a matrix size of 46,080 by 46,080 [4]. However, all 70 nodes are not available at all times.

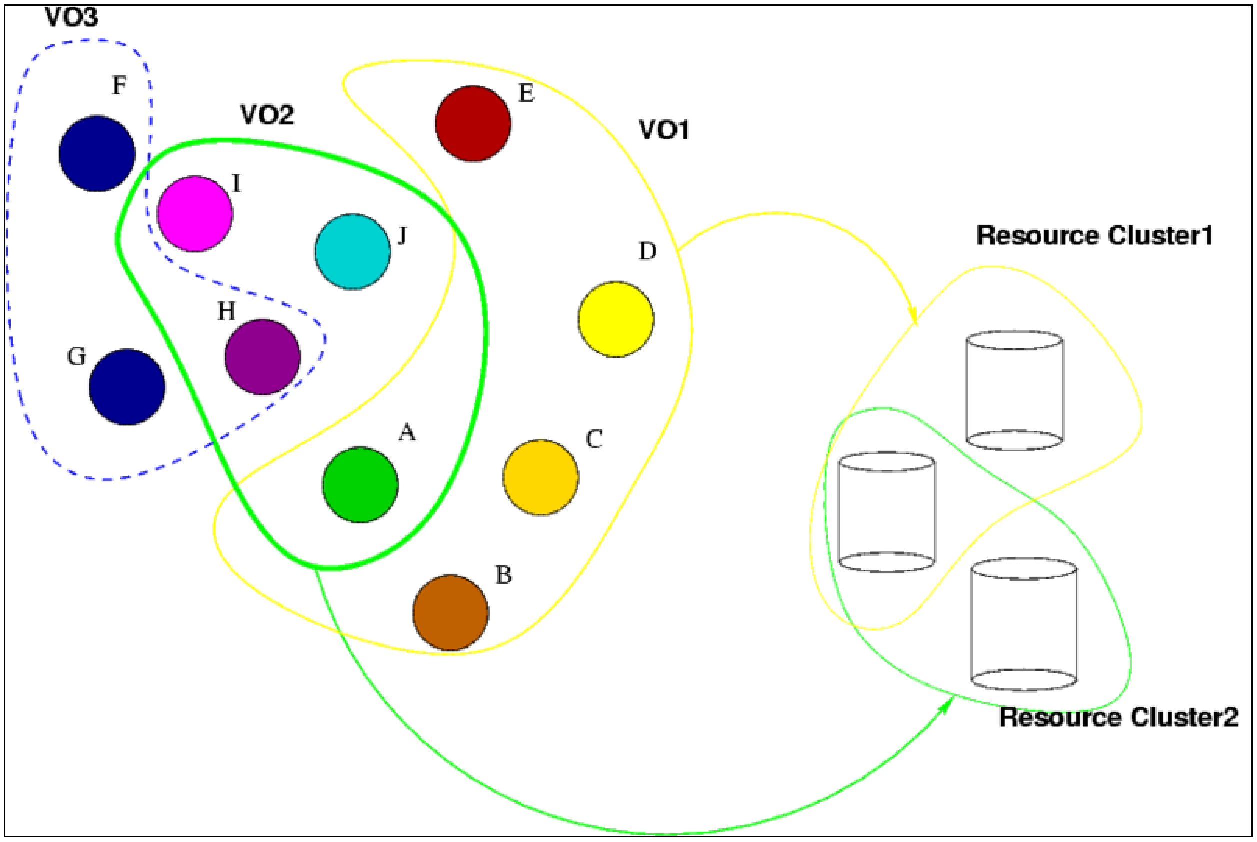


Figure 7: Distributed Computing Cluster [5]

Overall, distributed cluster computing (also known as “Grid Computing” or “Cloud Computing today) takes advantage of networked computers in an effort to create a “virtual supercomputer” to perform parallel operations via network links such as Ethernet as opposed to traditional parallel processing systems interconnected by a high speed computer bus [5]. Overall, a topology is created that breaks all available systems down into various resource clusters known as Virtual Organizations (VOs). Tasks are distributed among these VOs and results are communicated as needed.

Solutions using distributed clusters are rather attractive in that numerous, networked processors exist that sit idle during down times. With the advent of the Internet, literally millions of computers are now globally networked. In a report examining the use of broadband connectivity in the world, over 180 million active DSL lines, 60 million active cable lines, and 23 million active fiber lines connecting as many or more computers worldwide are in place [6]. This clearly indicates a substantial number of computing resources with broadband connections available at any given time on the Internet.

As a result, researchers are looking to exploit this untapped power that exists. Researchers at Stanford University, under the Folding@home project, utilize idle, networked Sony Playstation 3 gaming consoles, as well as other powerful computers, to perform complex, mathematical functions in an effort to understand protein folding, mis-folding, and related diseases [7]. Researchers at Berkeley are using the same distributed cluster approach to distribute the analysis of radio telescope data in an effort to locate intelligent radio signals under the SETI@home project [8].

The research performed at the University of Tokyo is similar to the projects mentioned above only this project focuses on LU factorization. The 70 node system is partitioned into various virtual organizations. A programming model known as “Phoenix” is developed to facilitate communications between VOs as well as construct an algorithm for task scheduling and load balancing. Also, methods are created to handle the communication protocol, the dynamic nature of nodes in the system, as well as the nondeterministic time involved with inter-processor communication. Keep in mind that parallel processing systems discussed in the other research have deterministic interconnect.

Overall, this system yields amazing results given the number of mathematical calculations required. However, this system requires very complex algorithms to schedule, distribute, and collect results. Moreover, the number of computing resources required is very large. A solution such as distributed computing clusters, while interesting, is not feasible for a straightforward solution to the LU factorization problem to say nothing of embedded system applications. Also, the system cannot perform a subsequent solution until the current solution is computed. Still, it is a glimpse at the future of parallel processing and how it pertains to the resources required for solving complex mathematical functions such as LU factorization. Some of the aspects of this architecture that may be used in the future pertaining to the 3D LU factorization pipeline are discussed in Section 5.1.5.

2.4 PARALLEL PROCESSING SYSTEM-ON-A-CHIP

While most research involved with solving LU factorization has been performed using large computer systems, some research has been performed at implementing a solution to LU factorization on a chip level as opposed to a large scale, discrete processor system. The benefits of using chips as opposed to large workstations are cost, reduced power consumption, and the potential to use these solutions in embedded systems. Researchers at the New Jersey Institute of Technology successfully implemented LU factorization at the chip level using FPGA technology [9].

The researchers employed a solution that is not unlike the shared memory solution discussed in Section 2.1. To create a computational platform, six, 32-bit RISC processors are instantiated in an Altera FPGA. These processors are connected to a shared memory using a customized block of logic on the FPGA fabric. The soft core IP processor from Altera, Nios, is used. The Nios RISC processor is a fully configurable soft IP that offers over 125MHz in an

Altera Stratix FPGA. Since floating point operations are desired for LU factorization, a single precision, IEEE754 floating point unit is used so that floating point operations are handled in hardware as opposed to software thus improving performance.

An algorithm is developed to partition the solution among five processors. The sixth processor acts as a "local controller" governing the operation of the entire process. These processors post results via the shared memory and the local controller broadcasts results to other processors as needed. These algorithms are not unlike those examined in Section 2.1. After one matrix is processed, the system is free to begin processing another matrix.

The researchers demonstrate that performance for non-trivial matrix sizes is rather impressive. In the case of a 5x5 LU factorization, a software solution takes 45,168 system clock cycles on a single CPU. When their parallel processing solution is implemented, the performance increases significantly as only 4,583 clock cycles are required [9]. Therefore, this research proves that application acceleration similar to that seen in similar architectures using workstations is possible on FPGAs.

The same researchers took this concept a step farther and implemented another system designed to solve LU factorization not unlike the system seen in Section 2.2. The system is implemented on an Altera FPGA. The algorithms to partition data in [9] are modified to handle load balancing. The feasibility of scaling the architecture is also presented. Also, distributed memory using on-chip SRAM is used as opposed to a single shared memory among all processors.

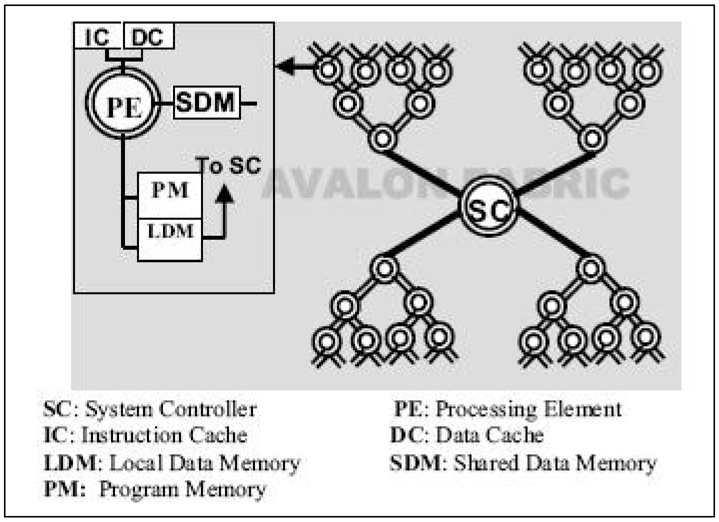


Figure 8: Binary tree of processors from [11]

One processor is assigned "system controller" to provide a mechanism for both partitioning tasks taking into account the load balancing of both processing communications. Each node of the tree consists of both instruction and data caches, both data and program memory for the Harvard Architecture Nios processor, and a shared data memory for communicating results to children in the tree. Each node also has a path to communicate results back to the system controller via Altera's "Avalon" bus fabric [10] [11].

The researchers indicate a significant improvement in the matrix operations reported in [11] of more than 20%. Still, even with these enhancements, overhead exists similar to that seen in their workstation-based relatives. Matrix processing time is sacrificed since inter-processor communication is required. However, the research in [9], [10], and [11] clearly demonstrates that LU factorization of non-trivial matrices in an FPGA is possible.

2.5 DEDICATED PROCESSING LOGIC

So far, every solution to the LU factorization problem presented involves the use of a traditional Von Neumann or Harvard CPU to perform calculations. In an effort to increase performance, multiple CPUs are used. When partitioning the design across multiple CPUs, some time must be dedicated to communication of intermediate results. This indicates that some processing time must be sacrificed so that communication results are broadcast to interested processors.

A potential solution to the communication bottleneck is the use of dedicated hardware to solve LU factorization. While the 3D Paul-Mickle pipeline proposes such custom hardware, researchers at the University of Southern California actually implemented a similar solution in 2004 in that a custom pipeline is used to solve LU factorization for any size of matrix. The linear pipeline is implemented such that it assumes data is being sent to the pipeline in a word by word format since the research assumes that hardware accelerators are fed streaming data from external CPUs or external memory [12].

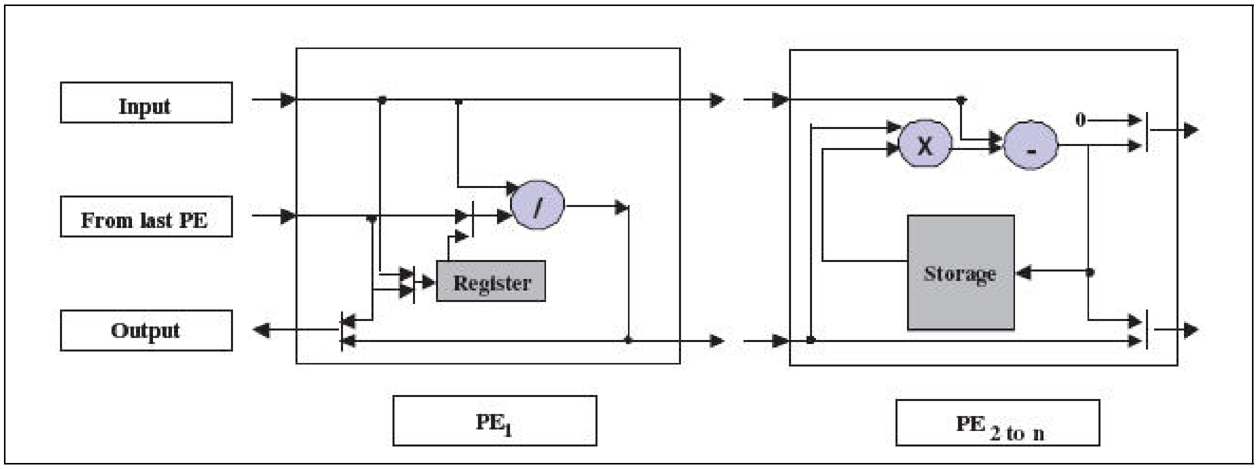


Figure 9: Linear array solution presented in [12]

The logic accepts data in a word by word, streaming format that arrives at the node labeled "Input" in Figure 9. The first processing element (PE_1) is responsible for the division required at each stage of the factorization algorithm. These quotients, as well as the initial values of the matrix are forwarded to the subsequent processing elements denoted PE $_{2\ \text{to}\ n}$ in the linear array. Control logic in PE_1 directs matrix data to either the divide function or directly to the output. Control logic in subsequent stages directs data to either the multiply or subtraction function as well as fetching of previous results from local storage when needed. Output that is needed for the next stage of the algorithm is fed back into the pipeline as needed. Also, the pipeline can begin to accept a new input matrix when the final stage of the current matrix being processed is performed. The following diagram from [12] details the operations that occur in each processing element of a pipeline that solves LU factorization for a 4x4 matrix.

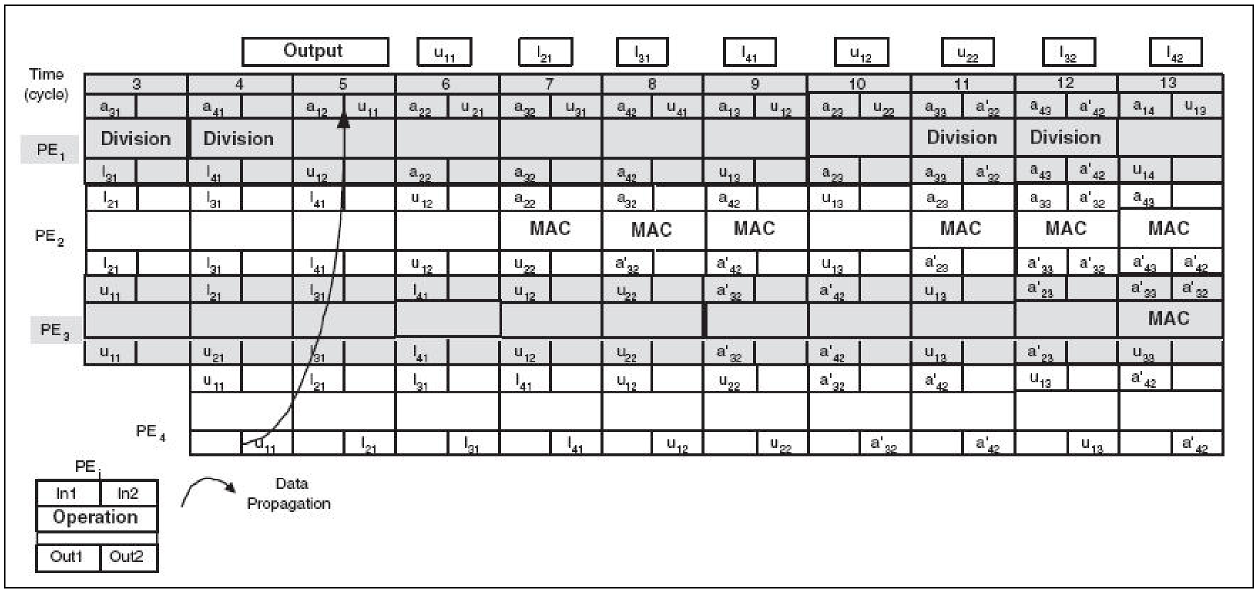


Figure 10: Dataflow of architecture presented in [12]

This architecture clearly demonstrates that implementing a custom pipeline via the use of a device like an ASIC or FPGA is feasible as opposed to a system using distributed CPUs. Each mathematical function is designed to support IEEE single precision format floating point operations and the design is functioning on a Xilinx Virtex-II Pro-125 device. Performance is exactly as expected. The latency achieved by this solution is $n^2 + n$ cycles [12]. So, when comparing these results to the results in [9] and [11] a substantial performance increase is observed. The 5X5 case in this architecture would only require 30 arithmetic cycles as opposed to the 4,583 cycles observed in the other FPGA research. Even if one factors in the amount of latency in the divide, multiply, and subtract functions, the end result is well over three quarters the number cycles required by the Nios based parallel processing system. While the latency reduction is impressive, it most certainly is improvable with a different, custom logic solution.

3.0 THE THREE DIMENSIONAL LU FACTORIZATION PIPELINE

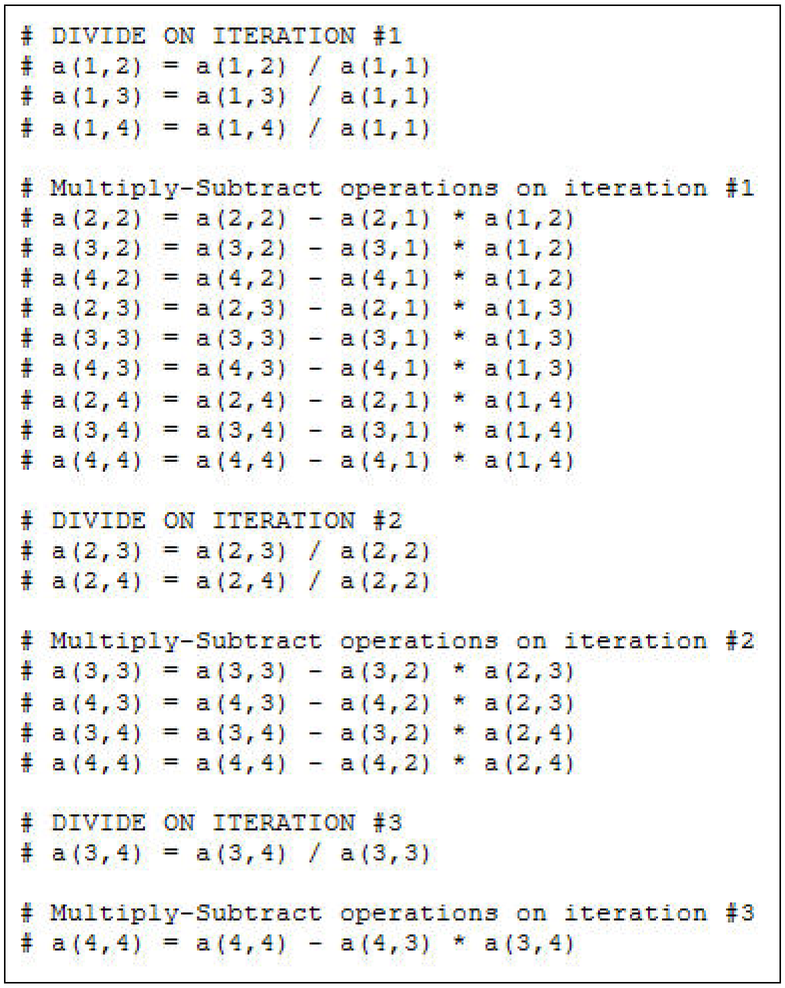
In the previous chapter, various architectures were explored from prior research that details various schemes used to solve LU factorization. Earlier implementations used banks of parallel processors to perform the function while later solutions migrated similar solutions performed on large scale workstations to single-chip implementations. Parallel processing solutions involving CPUs, as well as the more recent, distributed cluster architecture, provide a high-speed solution to the LU factorization problem with the tradeoff of significant computing resources being required, not to mention the classic communication overhead problem associated with parallel processing and overhead associated with the complex scheduling and load balancing algorithms. The later solutions utilizing FPGA technology reduce the physical requirements to a single device, but suffer from sub-optimal latency. A solution must exist that eliminates the communication overhead issue and effectively minimizes latency.

In 1998, Dr. JoAnn M. Paul and Dr. Marlin H. Mickle of the University of Pittsburgh presented an architecture that does not require a massive amount of parallel processors while, simultaneously, reducing the latency to perform LU factorization of a matrix size $n*n$ down to $4n-4$ arithmetic operation cycles. Moreover, in sustained operation, a new LU factorization is computed every single cycle for the minimal block pipelining period of 1 [1]. To date, this architecture is the optimal solution to LU factorization in terms of latency. The only question remaining is if it is at all feasible to create the pipeline to optimally solve LU factorization for non-trivial matrix size using custom hardware such as an FPGA.

This section presents the architecture devised by Dr. Paul and Dr. Mickle. Next, the hardware used to implement the pipeline is presented. Finally, an overview of the design flow, from resource estimation to final implementation is discussed.

3.1 THREE DIMENSIONAL ARCHITECTURE OVERVIEW

In Figure 2, the pseudocode used to implement LU factorization, also known as Crout Elimination, was presented. As stated, the loop appears trivial at first glance, but for non-trivial matrix sizes, it is clear that Crout Elimination required a substantial number of arithmetic operations. The following figure shows the Crout loop “unrolled” for a 4x4 matrix revealing all of the computations required to perform the function.



```
# DIVIDE ON ITERATION #1
# a(1,2) = a(1,2) / a(1,1)
# a(1,3) = a(1,3) / a(1,1)
# a(1,4) = a(1,4) / a(1,1)

# Multiply-Subtract operations on iteration #1
# a(2,2) = a(2,2) - a(2,1) * a(1,2)
# a(3,2) = a(3,2) - a(3,1) * a(1,2)
# a(4,2) = a(4,2) - a(4,1) * a(1,2)
# a(2,3) = a(2,3) - a(2,1) * a(1,3)
# a(3,3) = a(3,3) - a(3,1) * a(1,3)
# a(4,3) = a(4,3) - a(4,1) * a(1,3)
# a(2,4) = a(2,4) - a(2,1) * a(1,4)
# a(3,4) = a(3,4) - a(3,1) * a(1,4)
# a(4,4) = a(4,4) - a(4,1) * a(1,4)

# DIVIDE ON ITERATION #2
# a(2,3) = a(2,3) / a(2,2)
# a(2,4) = a(2,4) / a(2,2)

# Multiply-Subtract operations on iteration #2
# a(3,3) = a(3,3) - a(3,2) * a(2,3)
# a(4,3) = a(4,3) - a(4,2) * a(2,3)
# a(3,4) = a(3,4) - a(3,2) * a(2,4)
# a(4,4) = a(4,4) - a(4,2) * a(2,4)

# DIVIDE ON ITERATION #3
# a(3,4) = a(3,4) / a(3,3)

# Multiply-Subtract operations on iteration #3
# a(4,4) = a(4,4) - a(4,3) * a(3,4)
```

Figure 11: Crout loop unrolled for the 4x4 case

As seen in the loop, a significant number of operations are required for even a smaller case like $n = 4$. Crout Elimination requires $n^3/3$ arithmetic operation pairs. When n^3 is not an integer multiple of three, $n^3/3$ is rounded to the nearest integer [1].

With the loop unrolled, it is apparent that parts of the Crout algorithm can be made parallel as enough data is present to perform partial computations in each stage. Overall, the first operations that must be performed are the division functions in the first “active” row of each stage where the “active” row number is equal to the current stage number. Once the quotients are computed, the multiply operations can be performed. Thereafter, the subtraction function can be performed. After all three arithmetic functions are performed, it is safe to begin the computations for the next iteration of the Crout loop.

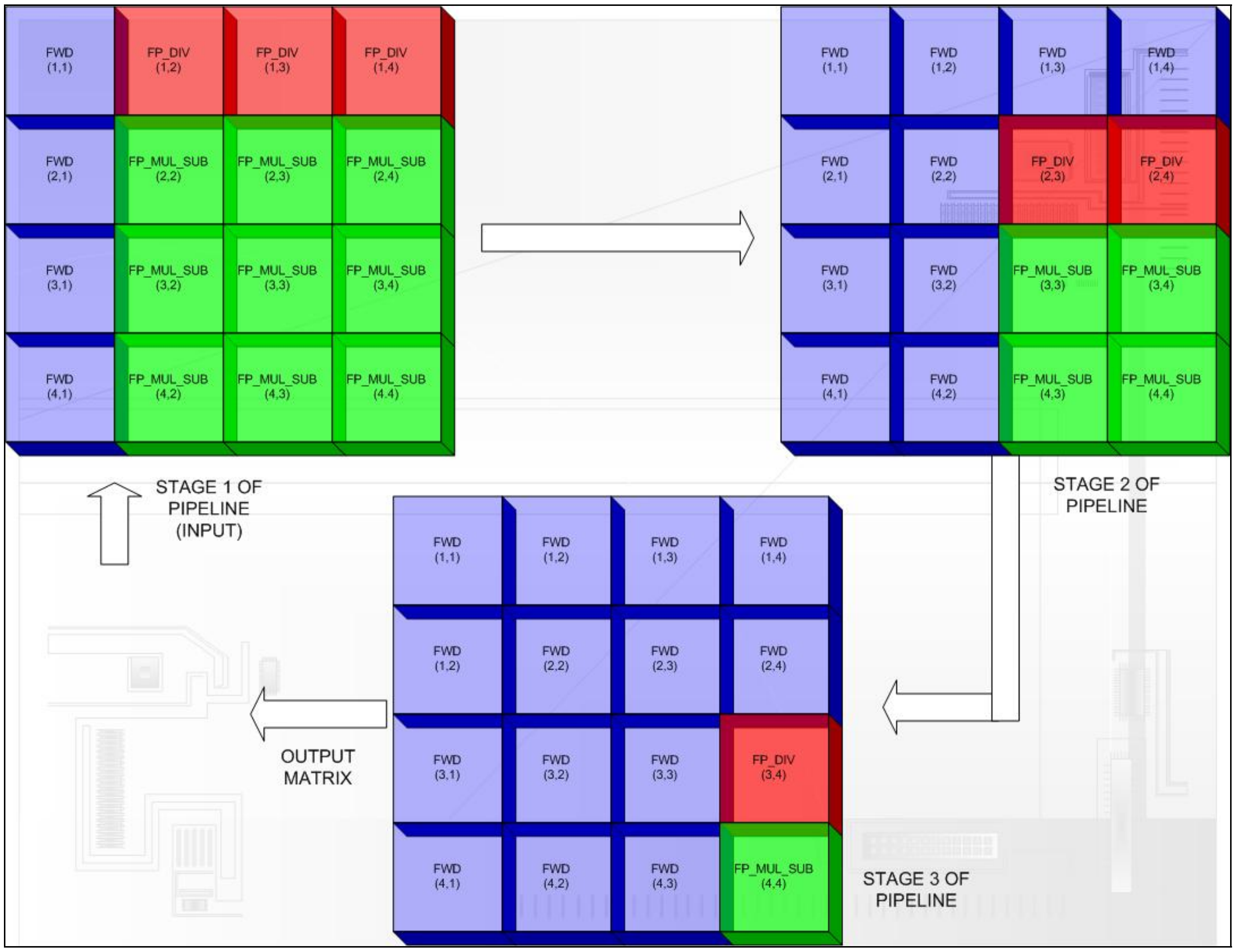


Figure 12: Crout 3D pipeline

Figure 12 helps graphically depict the structure of the 3D pipeline for a 4x4 case. The Paul-Mickle 3D architecture groups these Crout loop processing iterations into matrix pipeline stages. The elements of each stage of the matrix correspond to one of three functions. First, the function labeled "FWD" simply forwards data to the next matrix stage without any arithmetic operation being performed. Next, the function labeled "FP_DIV" divides the current contents of the matrix by the element (*stage,stage*) where *stage* corresponds to the current stage of the matrix pipeline. All quotients are computed in parallel.

After the quotients are available, the function "FP_MUL_SUB" performs an arithmetic pair of operations. First, the quotient is multiplied by the corresponding row element under the element (*stage,stage*). For clarity, suppose one was examining element (3,2) during matrix stage 1 of the pipeline. The quotient computed in element (1,2) would be multiplied by the corresponding row element under (*stage,stage*) which, in this example, is element (3,1). After the product is computed, the product is subtracted from the current value of the element being processed. At this point, all computed values are forwarded to either the next stage of the matrix pipeline or are output from the pipeline in the final stage since the solution is now known.

The pipeline falls into the "3D category" due to matrix stage stacking. Each matrix pipeline stage is a 2D pipeline. Each element of the matrix stage is a linear pipeline. The "FWD" function is a linear pipeline of no-operations that is equal in latency to the total number of cycles it takes to divide, multiply, and subtract. The "FP_DIV" function is a divide pipeline. The "FP_MUL_SUB" pipeline includes three different pipelines. The first is the multiply pipeline used to multiply the quotient by the corresponding row element under (*stage,stage*). Once the product is computed, the subtraction can occur. However, a pipeline is hidden in that the current contents of the matrix stage element that has the multiply-subtract requirement must be maintained until both division and multiplication are complete. The final pipeline in each matrix stage is the actual subtraction pipeline. Also, an additional, no operation pipeline stages at the output of the divider function must be added to compensate for the latency of the multiply-subtract function. So long as the matrix stage is created as described, a full matrix can be input on each cycle making the 2D pipeline fully systolic. The third dimension is the stacking of each matrix stage. The output of a stage's current element connects directly to the input of the corresponding element of the next stage.

3.1.1 3D Pipeline Example

This section details the "flow" of a matrix element through the pipeline in an effort to clarify the pipeline operation. To simplify the description, a 3x3 matrix example is used to demonstrate all arithmetic operations required on a particular element. In particular, the flow of data for element (3,3) of the 3x3 case is examined. Once this flow is understood for the 3x3 case, any user of this logic should be capable of scaling the logic to accommodate any size of matrix.

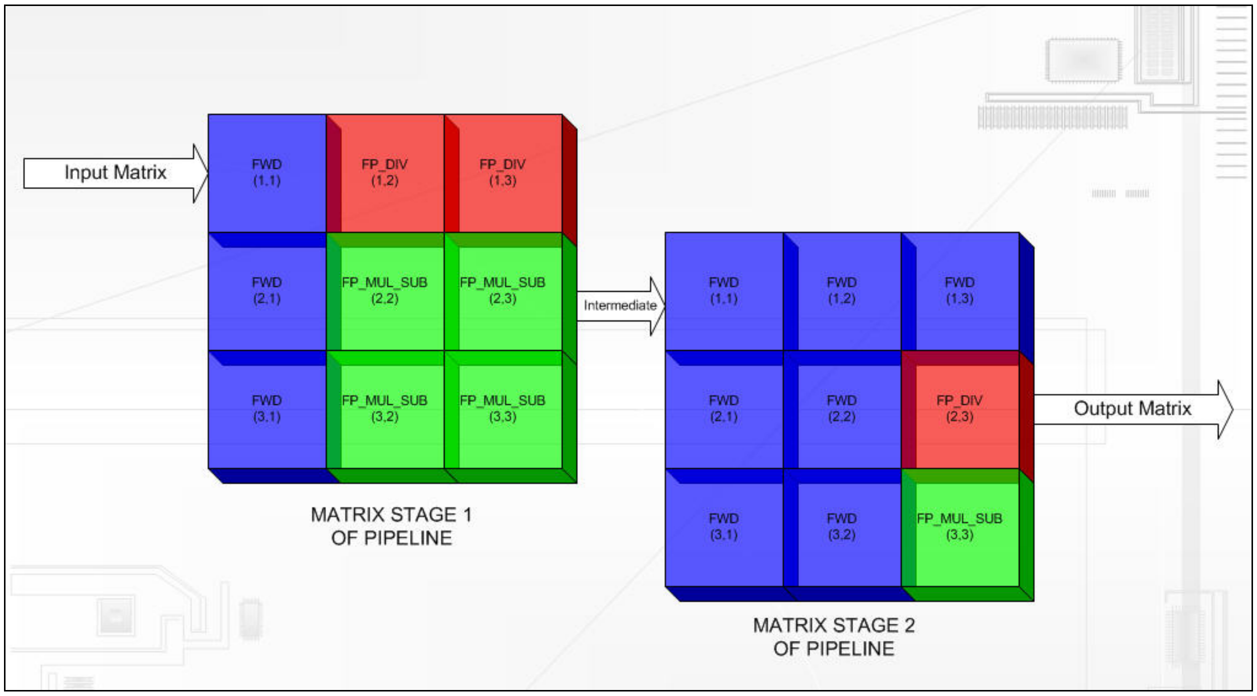


Figure 13: Overview of the 3x3 example.

A 3x3 Paul-Mickle pipeline requires 2 matrix pipeline stages. Figure 13 details the arithmetic operations that occur in each element of the particular stages. The structure of the pipeline is identical to stages 2 and 3 of the 4x4 example presented in Section 3.1. A complete matrix arrives at the node "Input Matrix" and exits the pipeline at the node "Ouput Matrix". Since each computing element (i.e. FWD, FP_DIV, or FP_MUL_SUB) is pipelined, a new matrix can be presented at the node "Input Matrix" each clock cycle. After the initial latency of the computing elements passes, a solved matrix arrives at the node "Ouptut Matrix" on each clock cycle. The intermediate node is the point where computations are complete for the first pipeline stage and data enters the second stage. Finally, the output of each element is connected to the input of the corresponding element.

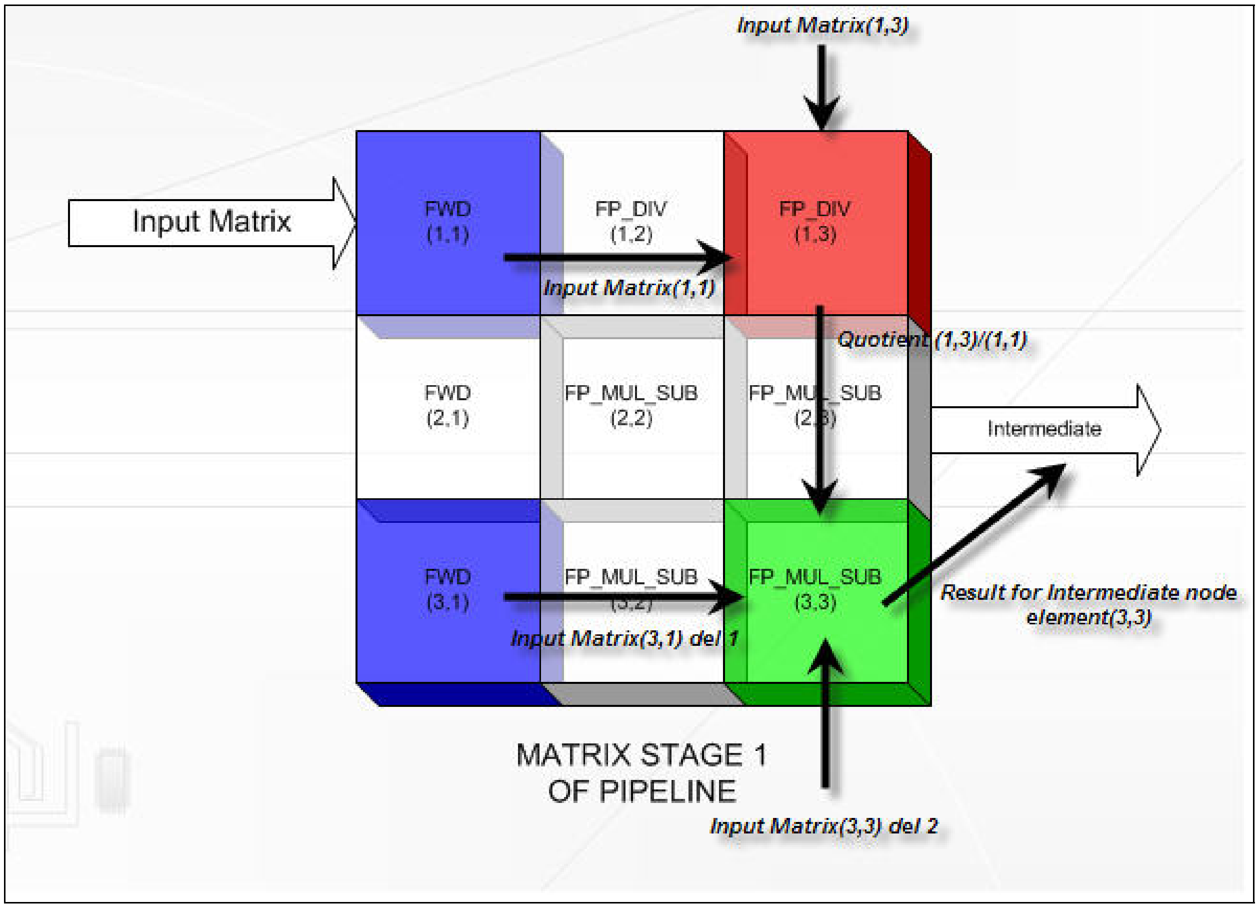


Figure 14: Dataflow for matrix stage 1

Referencing Figure 14, data arriving at the node "Input Matrix" is immediately used as input for the FP_DIV processing element. In the case of a 3x3 matrix, the FP_DIV logic at location (1,3) uses both Input Matrix$_{(1,3)}$ and Input Matrix$_{(1,1)}$ to compute the quotient (1,3) / (1,1). After the quotient is present, the product of quotient$_{(1,3)}$ * Input Matrix$_{(3,1)}$. Since each processing element of the matrix is considered to be systolic, a delay is required so that one meets the requirement of introducing a new matrix every clock cycle. The delay required for the multiplication at element (3,3) is denoted in Figure 14 as "del 1". This delay is equivalent to any delay introduced by the division function.

Once the product is calculated, the subtraction of product$_{(3,3)}$ from Input Matrix$_{(3,3)}$ can be performed. Both multiplication and subtraction are performed in the processing element FP_MUL_SUB. Again, another delay must be introduced on the data at node "Input Matrix" to keep the pipeline systolic. This delay is denoted in Figure 14 as "del 2". This delay is equal to the number of cycles required by both the division and multiplication functions. After subtraction is complete, all results are available at the node labeled "Intermediate" for the second stage of the pipeline.

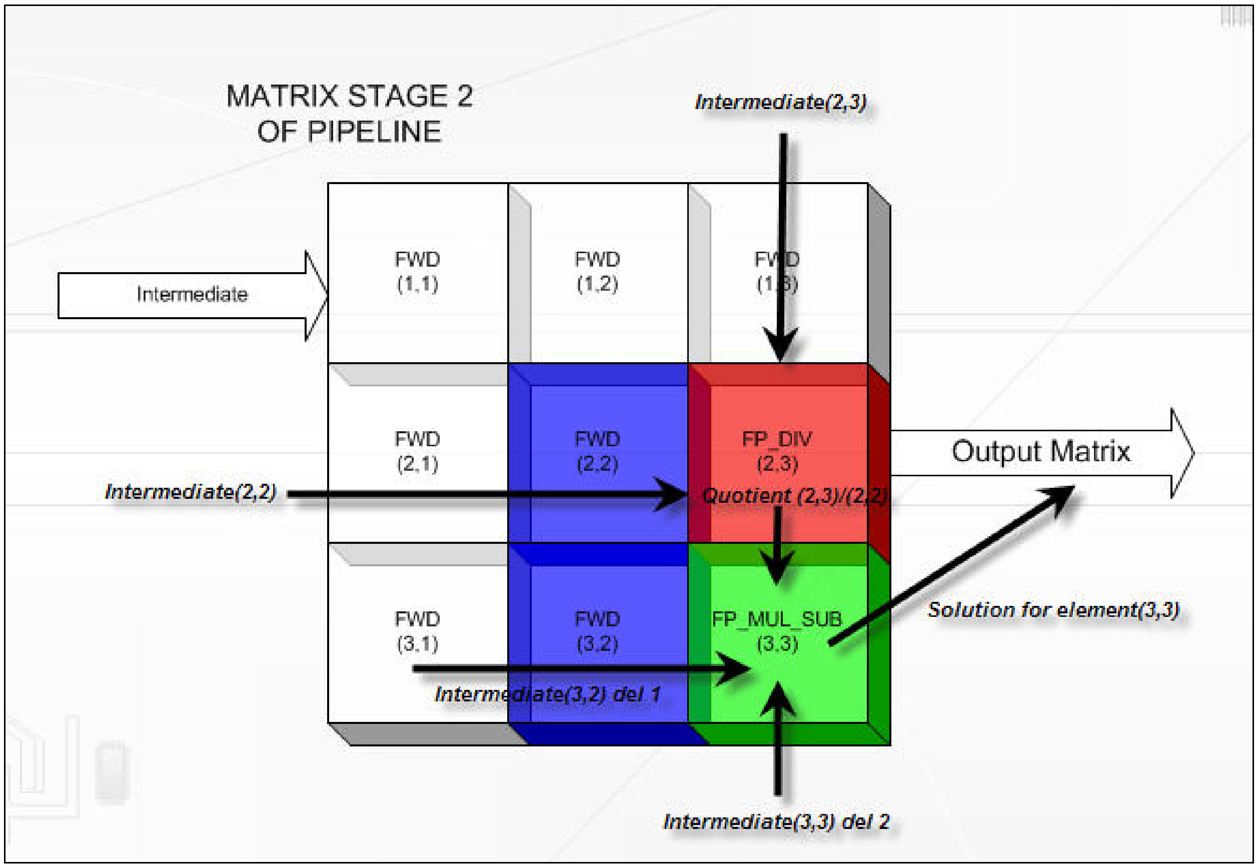


Figure 15: Second matrix stage of the 3x3 example.

Figure 15 details the flow of data in the second stage of the pipeline. Data arrives at the intermediate node and the quotient is computed at element$_{(2,3)}$ from Intermediate$_{(2,3)}$ and Intermediate$_{(2,2)}$. After the quotient at element$_{(2,3)}$ is ready, the subsequent multiplication of quotient$_{(2,3)}$ and Intermediate$_{(3,2)}$ at element$_{(3,3)}$ can occur after the delay denoted by "del 1". Finally, subtraction of the product calculated at element$_{(3,3)}$ from Intermediate$_{(3,3)}$ can be performed after the delay "del 2". After these operations are performed, the solution is present at the "Output Matrix" node.

3.2 HARDWARE SELECTION

Now that the architecture is understood, work can begin on the implementation of the pipeline. Based on the description of [1], it is quite apparent that numerous divide, multiply, and subtraction processing elements are required. In an effort to demonstrate feasibility of "real world" implementation, the processing elements are chosen to support IEEE single precision floating point representation. Since floating point computations typically require a significant amount of logic to implement, a rather large FPGA must be used. Moreover, a non-trivial matrix size must be selected.

To satisfy the "large FPGA" requirement, the Xilinx Virtex 5 SXT 50 is selected. The device provides 32,640, 6 input look up tables (LUTs) as well as 32,640 flip flops. Moreover, an additional 288 48-bit dedicated multiply-accumulate functions known as DSP48 blocks are available for general use [13]. To avoid the intricacies involved with creating a custom printed circuit board, the Xilinx ML506 development board is chosen to implement the design. This board has the Virtex 5 device required to implement the pipeline as well as peripheral circuitry useful to test the pipeline once developed.

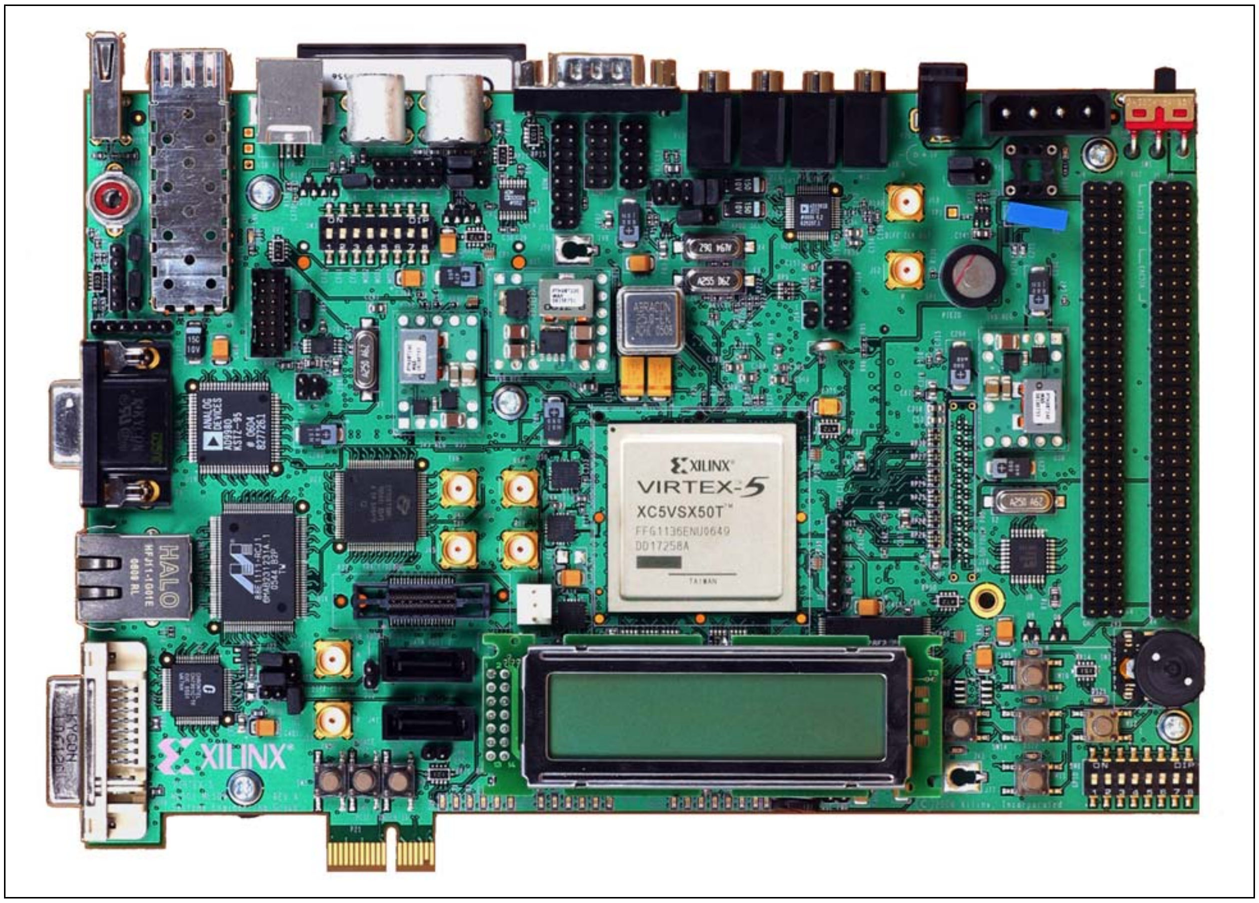


Figure 16: The Xilinx ML506 development kit

In an effort to demonstrate a non-trivial matrix pipeline is reasonable for implementation, a 5x5 matrix pipeline is chosen. While Crout Elimination is complex for any size of matrix, a 5x5 matrix shall require a significant amount of floating point processing to demonstrate that the 3D pipeline can be realized for a non-trivial size. Overall, the Virtex 5 SXT 50 should be able to hold all of the logic and test circuitry required for the 5x5 solution using IEEE754 single precision floating point numbers.

3.3 DESIGN FLOW OVERVIEW

Although the design is targeting FPGA technology, a design methodology similar to that of ASICs is used. First, primitives required for the design are required to be created. Overall, this entails determining solutions for the floating point functions required in the pipeline. For demonstration purposes, it is desirable to use floating point functions that consume as little FPGA resources as possible.

Once these floating point functions are chosen, they are assembled to form the 3D Crout Elimination pipeline. VHDL is used to assemble the pipeline as it is one of the major languages used for logic synthesis. A VHDL simulator as well as a custom written VHDL testbench is used to verify the functionality of the pipeline. While an FPGA is reconfigurable unlike its ASIC brethren, it is rather inefficient as well as difficult to correct errors by synthesizing and uploading results to the FPGA as opposed to simulating the functionality of the device. FPGA build times can take hours for designs as complex as a 5x5 Crout Elimination pipeline.

After the Crout Elimination pipeline functionality is verified, work begins on a way to test the circuit in-system. Overall, an interface is required to both drive data into the pipeline as well as capture results. The simplest way to perform this task on a wide range of test vectors is to have a CPU drive these test vectors into the pipeline. The CPU chosen to perform this task is the Xilinx Microblaze processor. Also, since the input and output bit widths of the Crout pipeline are rather wide (25 matrix elements at 32 bits wide is equal to 800 total bits), some form of translation is required to convert the 32 bit accesses of the Microblaze to the 800-bit wide vectors the pipeline requires. This translation can be achieved using on-chip memory available on the Xilinx device.

Once the pipeline and associated test circuitry are created, the design is synthesized and uploaded to the target device using the Xilinx ISE tools. Ideally, a simple test program is written to exercise the hardware and observe the results. Should these results match expected results, the development of the design is complete and the 3D Crout Elimination pipeline will be realized!

4.0 DESIGN, IMPLEMENTATION AND VERIFICATION OF THE PIPELINE

This chapter details the design, implementation, and verification of the 3D Paul-Mickle LU Factorization Pipeline. First, the development of IEEE 754 floating point primitives is discussed in an effort to determine logic utilization requirements. Next, the design of the pipeline is discussed along with simulation results. Third, the creation of an embedded system using a Xilinx FPGA to physically test the pipeline in hardware is presented. Overall, this chapter is intended to present the design flow to implement the architecture proposed in chapter three.

4.1 IEEE 754 FLOATING POINT PRIMITIVES

The core mathematical primitives required to perform Crout Elimination are simply divide, multiply, and subtract. Unfortunately, IEEE754 arithmetic functions are typically expensive in terms of hardware utilization or CPU cycles given the algorithms required to perform the said functions. However, floating point functions are virtually required as they provide precision not found using integer or fixed point numerical representation. As a result, using the IEEE754 format is almost mandatory for real-world applications. Many vendors provide intellectual property that performs floating point arithmetic. For this thesis, two available solutions are examined.

The first solution explored is a synthesizable IEEE HDL floating point library known as 1076.3. The goal of IEEE1076.3 is to provide a floating point synthesis package for VHDL and Verilog based on IEEE 754 [14]. Currently, this package is freely available for download and is actively maintained as of 2008. Unfortunately, after some experimentation, this package is not ready for the tool suite used in this thesis. While the maintainer of the package has had success using the package with the Synplify Synplicity synthesis tool, Xilinx ISE synthesis tools, as of version 9.2.04, do not support some of the VHDL constructs required by the package [15]. Fortunately, some features of the new package are used for verification purposes and are discussed later.

The solution used in the implementation of the pipeline is provided by Xilinx via the Coregen FPGA IP generator [16]. Assuming a Xilinx user has a license to use ISE, Xilinx provides a netlist generator that creates a fully working floating point core. Moreover, the floating point core can be customized in various ways to provide users flexibility in examining performance vs. area tradeoffs. Since this solution is known to work, it was chosen for this implementation. With that in mind, creation of the cores and resource utilization must be considered to see if these cores do not exceed the available resources in the target device.

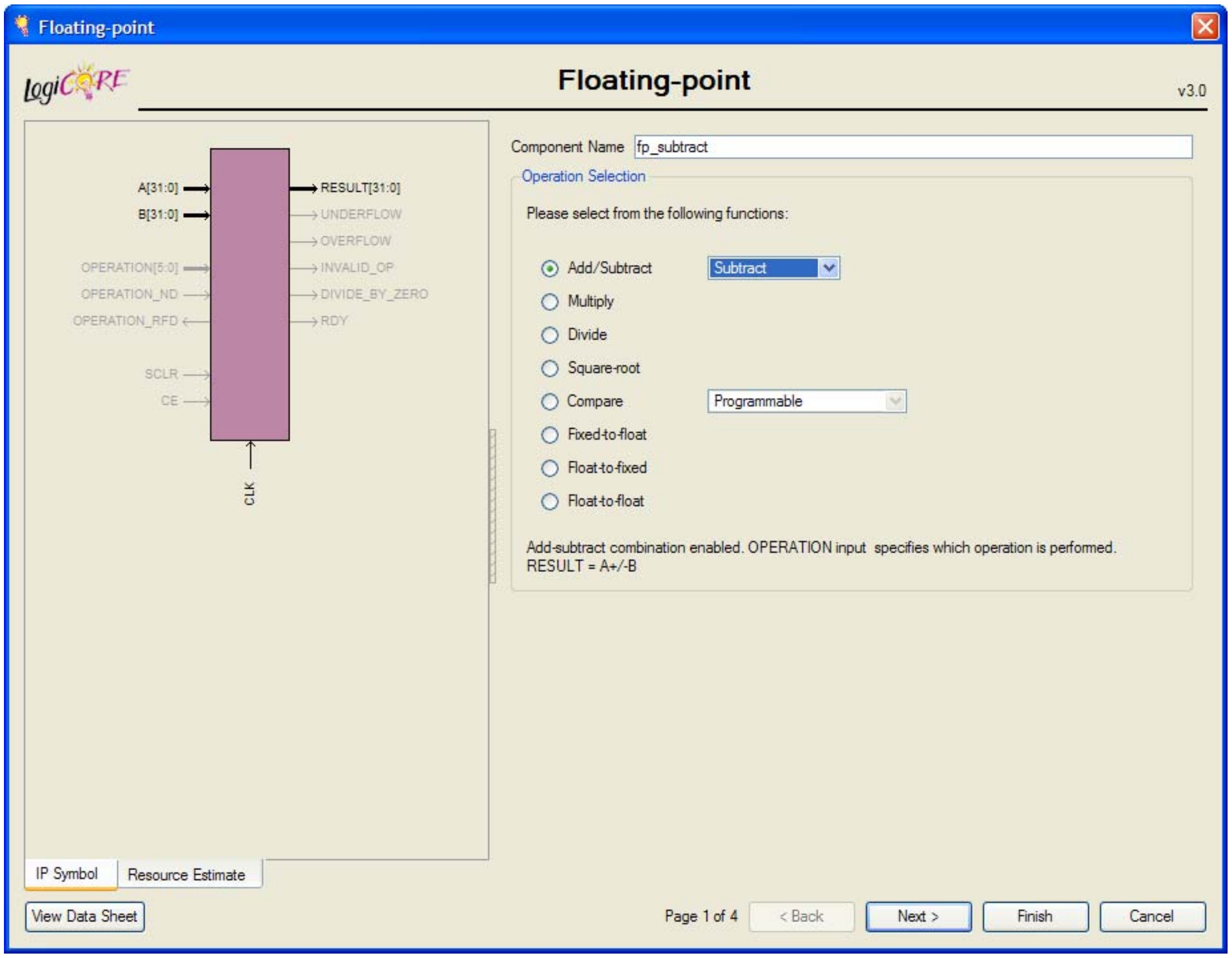
4.1.1 Floating Point Logic Creation

Since the target device is a Xilinx Virtex 5 SXT 50, the following key resources are available for use. First, the device has 32,640, 6-input Look Up Tables (LUTs) as well as 32,640 flip flops on the FPGA fabric. Next, the device has 288 DSP slices available that contain logic for creating both adders and multipliers without using FPGA LUT and flip flop resources. Finally, the device contains 144 36 kilobit “blockRAMs” for use as general purpose memory.

While the embedded memory is not critical to the Crout pipeline itself, it will serve a useful purpose as seen later in this thesis.

With these area constraints known, the parameters used to generate the floating point cores for the 5x5 pipeline can be determined. Again, the floating point cores are created using the Xilinx tool Coregen. Coregen allows one to custom tailor the floating point core to fit their area requirements two ways: pipeline latency and cycles per operation when using IEEE754 floating point numbers. The following overview details the approach used to generate the cores.

First, the user selects the function to create. This is done via a simple menu after a user chooses to generate a floating point core in the Coregen IP menu.



Component Name: fp_subtract

Operation Selection

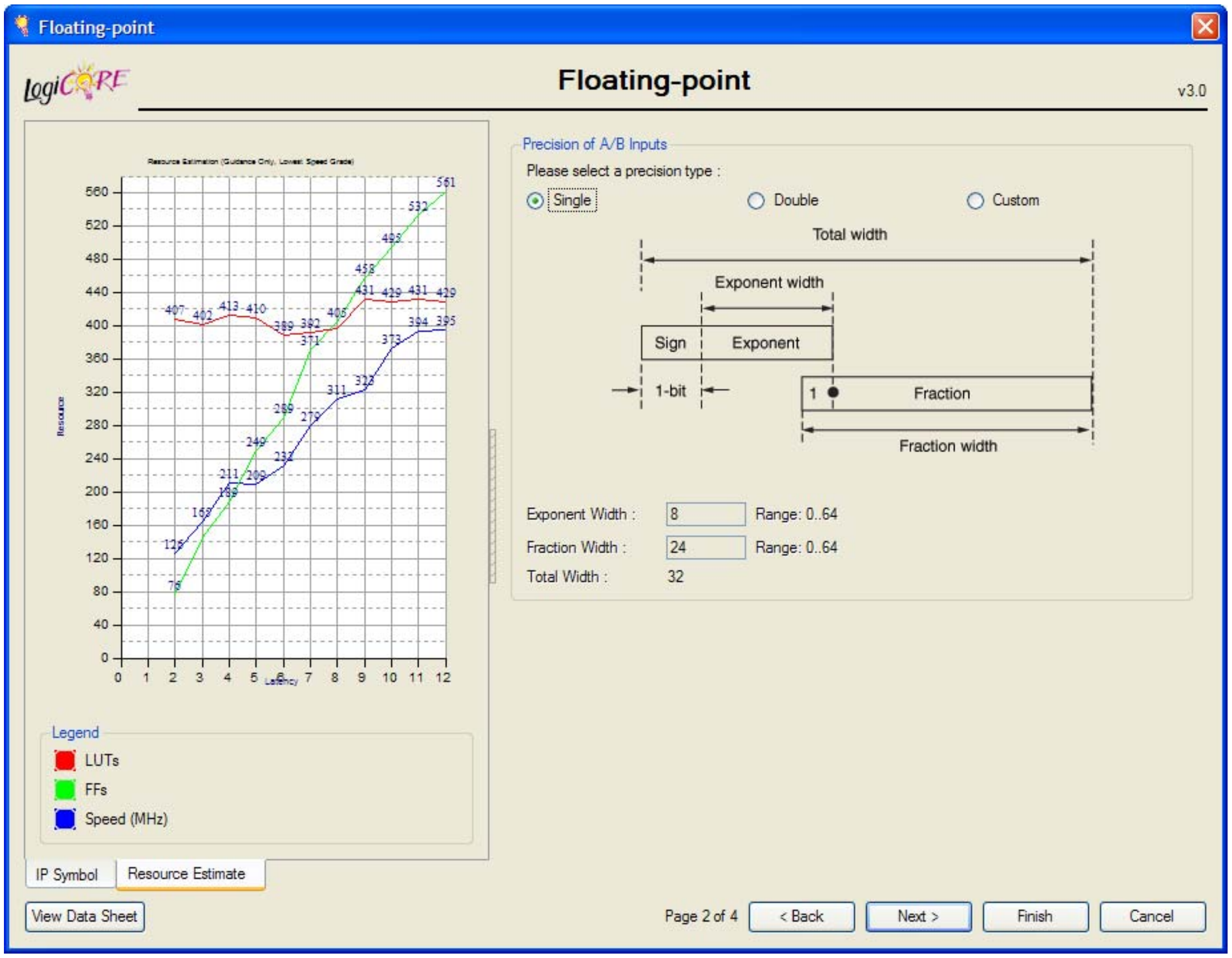
Please select from the following functions:

[✓ Add/Subtract] Subtract
Multiply
Divide
Square-root
Compare — Programmable
Fixed-to-float
Float-to-fixed
Float-to-float

Add-subtract combination enabled. OPERATION input specifies which operation is performed.
RESULT = A+/-B

Figure 17: Xilinx Coregen Floating Point Selection Menu.

The menu presents various floating point functions the user can choose. In this example, a floating point subtraction function is created. Once the function is chosen, the user then customizes the precision of the core.



Precision of A/B Inputs

Please select a precision type : [✓ Single] Double Custom

Exponent Width :	8	Range: 0..64
Fraction Width :	24	Range: 0..64
Total Width :	32	

Legend

🟥 LUTs

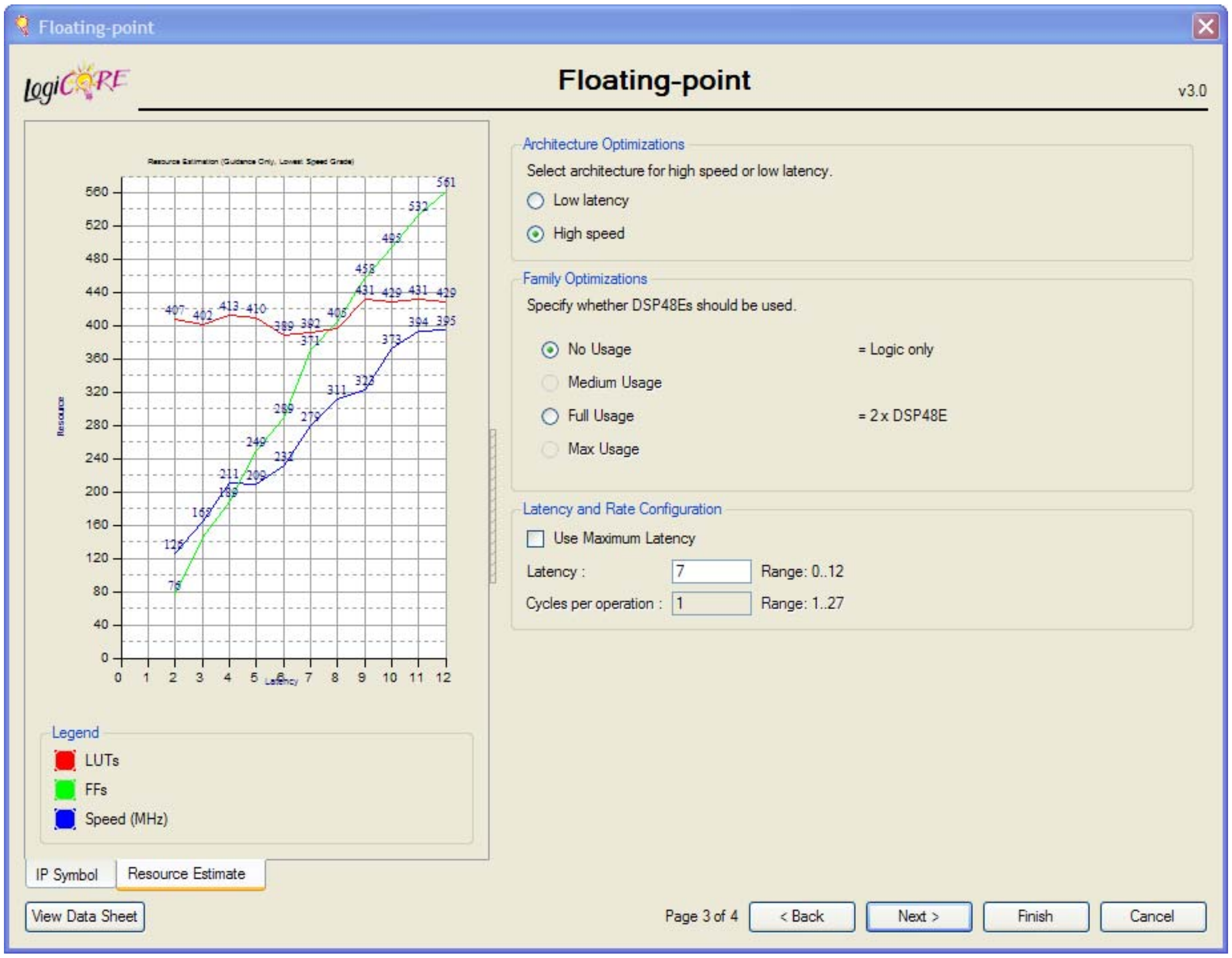
🟩 FFs

🟦 Speed (MHz)

Figure 18: Floating point precision menu

For the Crout pipeline, "single precision" floating point numbers are chosen as indicated by the "precision type" checkboxes. As specified in IEEE754, there is one sign bit, 8 exponent bits, and 23 mantissa bits. Be aware that if more logic were available in the target device, a user could choose double precision or custom field widths for their applications. If a custom format is required, the user would input the field lengths in the "Exponent Width" and "Fraction Width" boxes.

The final option provided by Coregen allows the user to tweak the area of the core by trading off performance.



Floating-point

Architecture Optimizations

Select architecture for high speed or low latency.

Low latency [✓ High speed]

Family Optimizations

Specify whether DSP48Es should be used.

[✓ No Usage] = Logic only

Medium Usage

Full Usage = 2 x DSP48E

Max Usage

Latency and Rate Configuration

☐ Use Maximum Latency

Latency : 7 Range: 0..12

Cycles per operation : 1 Range: 1..27

Figure 19: Floating point performance vs. area tradeoffs

As seen in Figure 19, LUT utilization for a subtraction function remains relatively constant while flip flop utilization increases as the required maximum frequency of the core increases since additional pipeline stages are required. Also, the user may choose to use Xilinx DSP48 elements to reduce the logic required for the addition and multiplication functions. DSP48 elements lower the LUT and flip flop count requirements and boost the overall performance of the core. However, the tradeoff is the limited number of DSP elements in the FPGA: there are only 288 of these present in the targeted device. DSP elements also restrict the portability of the design to other FPGA platforms as many FPGA platforms do not have this functionality.

After experimenting with the various tradeoffs given the resource limitations, the parameters chosen for the design indicate that a 5x5 solution is feasible in the target device under certain constraints. First, the IEEE754 subtraction and multiplication functions will consume an acceptable amount of resources while still maintaining full systolic functionality. A pipeline latency of 7 is chosen for the subtraction function with zero DSP slice utilization since this value balances out performance versus logic utilization. Next, the multiply function is chosen to be an 8 stage pipeline with full systolic operation for the same reasons as the subtraction function: performance versus logic utilization. However, multipliers tend to consume significant LUTs, so one DSP48 element is used per multiplier in an effort to fit the design in the target device.

The divide function needs to be created in a non-systolic fashion as logic utilization increases if the function is "fully pipelined". Documentation on the divide function shows that the number of flip flops required for a systolic divide function is over 5 times greater than one that requires wait states [17]. As a result, the wait-state version of the core is used along with its 26 cycle latency. This does not disqualify the 3D pipeline from implementation in an FPGA since the goal is to prove that a new matrix can begin processing as soon as a current matrix is done being processed in the first stage of the pipeline. However, more logic would be required if the user desired a solution that clocks a new matrix into the pipeline on each edge of the system clock.

Since the design is using arithmetic functions that are not “fully systolic”, shift registers are not required to delay the input and subsequent intermediate nodes. This saves a logic as well since the input and intermediate nodes will only be registered at each matrix stage. However, a “complete” systolic implementation will require these shift registers thus boosting the total logic count for the design.

4.1.2 Floating Point Logic Utilization

Since the demonstration pipeline is four “matrix” stages, the total logic required by all of the floating point elements based on the architecture discussed in Chapter 3 must be determined. The following table details expected logic utilization for all of the floating point cores required by the pipeline. The number of cores required is derived from the unrolled loop of a 5x5 matrix as discussed in Section 3.1 and detailed in Figure 11.

Table 1 : Resource Estimation

	Primitive	LUTs Required	Flip Flops Required	DSPs Required
	FP Divide	227	233	0
	FP Multiply	293	242	1
	FP Subtract	392	371	0
Stage 1	Number of Primitives	Total LUTs	Total Flip Flops	Total DSPs
Divide	4	908	932	0
Multiply	16	4688	3872	16
Subtract	16	6272	5936	0
Stage 2				
Divide	3	681	699	0
Multiply	9	2637	2178	9
Subtract	9	3528	3339	0
Stage 3				
Divide	2	454	466	0
Multiply	4	1172	968	4
Subtract	4	1568	1484	0
Stage 4				
Divide	1	227	233	0
Multiply	1	293	242	1
Subtract	1	392	371	0
Totals	70	**22820**	**20720**	**30**

It is determined that the parameters chosen indicate that the pipeline fits in the target device. More than 10,000 logic elements are remaining. These will be reserved for the simple, processor based system functions required to drive the pipeline in hardware once the pipeline is synthesized.

4.2 PIPELINE AND TEST CIRCUITRY DEVELOPMENT

Now that all issues pertaining to the creation of floating point logic are resolved, focus is shifted to the creation of the pipeline itself. This section details the design methodologies used in the creation of the three dimensional LU factorization pipeline as well as support circuitry needed to test the pipeline. Overall, a hierarchical design approach is used so that a relatively clear method of using conditional instantiation techniques available in HDLs is facilitated.

4.2.1 Multiply-Subtract Component

The first component in the design hierarchy required is a multiply followed by subtraction function. Recall a “matrix stage” in the pipeline:

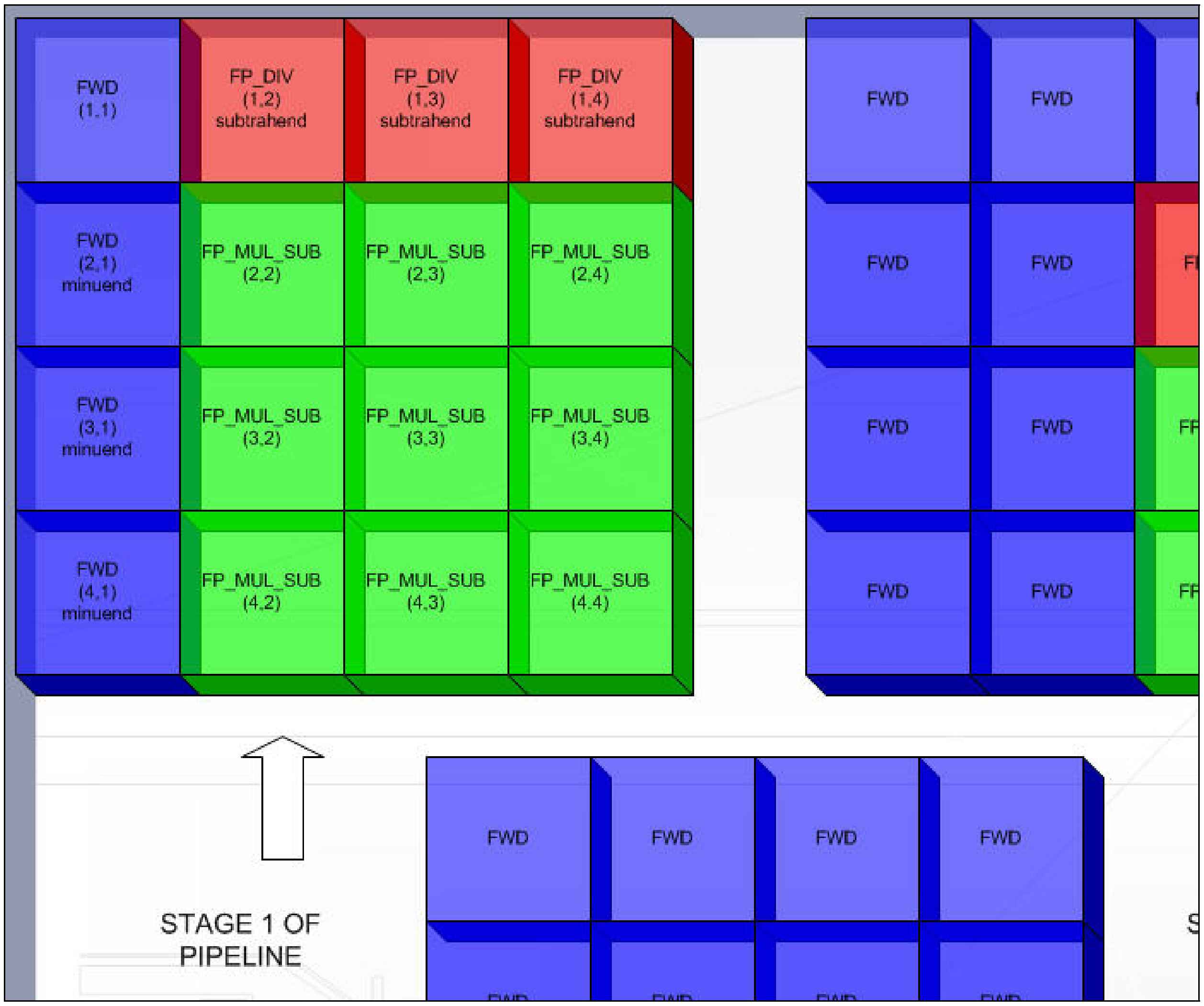


Figure 20: Single stage of 3D pipeline

As seen in the figure above from the 4x4 example detailed in Chapter Three, the multiply and subtract functions occur consecutively after the quotient is calculated from the floating point division functions. To express this clearly, suppose one is looking at the calculations for element A(3,3) and A represents stage one of the pipeline.

First, the following divide function must occur on element A(1,3):

$$A(1,2) = A(1,2) / A(1,1)$$

Once division is complete, we are free to perform:

$$A(3,3) = A(3,3) - A(3,1)*A(1,3)$$

So, in maintaining a proper design hierarchy, a circuit is developed to accept the results of both the current value of element being processed as well as both the result of the division function and the column of values used as the minuend of the subtraction function. Once these functions are performed, the results are forwarded to the next matrix stage of the pipeline. This multiply-then-subtract function begins once the divide function for the current stage is complete.

Creating this logic as a single component allows easy instantiation of multiple multiply-then-subtract functions via the use of VHDL generics. In the next section, this component is combined with the floating point divide function as well as conditional instantiation to create the basis of the pipeline.

4.2.2 Crout Matrix Stage Component

With the "multiply then subtract" function complete, conditional instantiation is used to determine connectivity in each stage of the matrix pipeline. The following pseudocode details this instantiation. It is a basic nested instantiation loop where the term MATRIX_M indicates the size of the matrix while the term STAGE represents the pipeline stage we wish to generate. Also, the term "mtx_in" represents the input from the previous stage of the pipeline while the term "rslt_m_i" indicates an output of the stage.

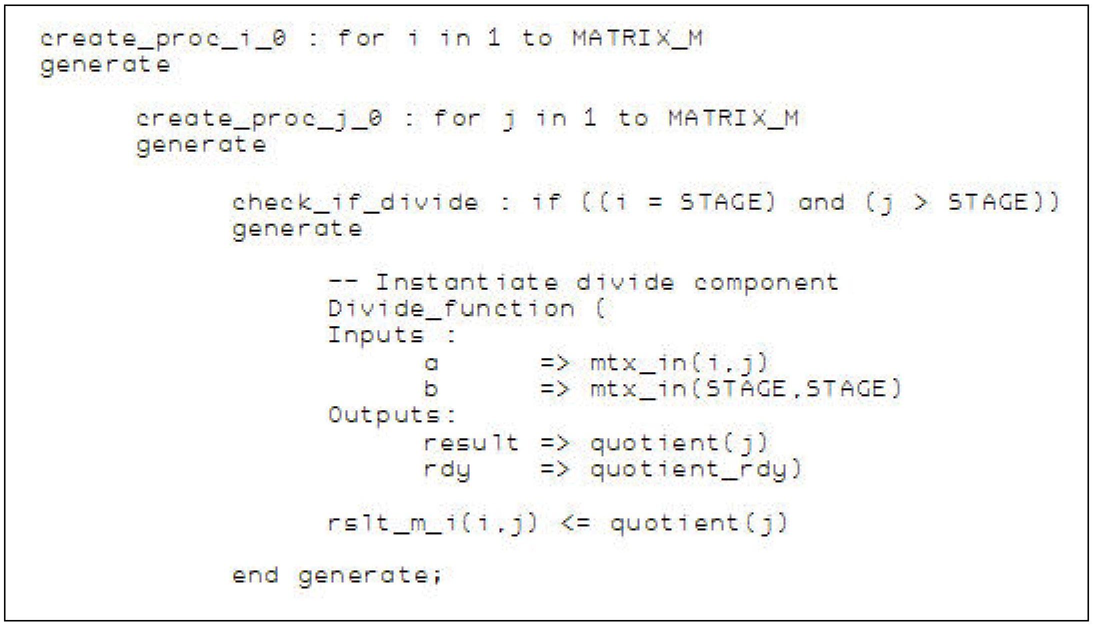
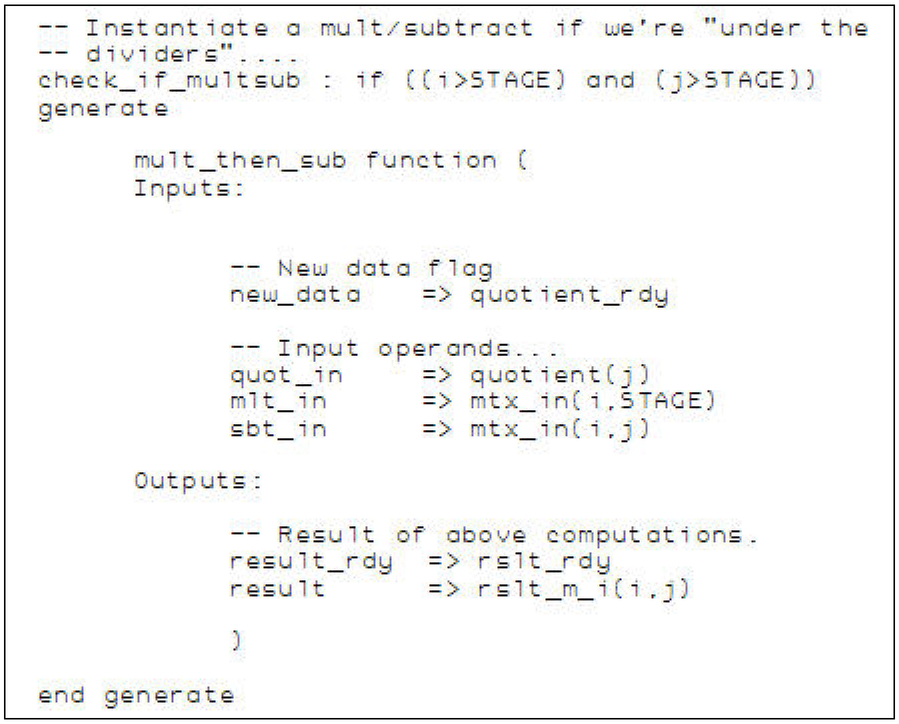

```
create_proc_i_0 : for i in 1 to MATRIX_M
generate

      create_proc_j_0 : for j in 1 to MATRIX_M
      generate

            check_if_divide : if ((i = STAGE) and (j > STAGE))
            generate

                  -- Instantiate divide component
                  Divide_function (
                  Inputs :
                        a      => mtx_in(i,j)
                        b      => mtx_in(STAGE,STAGE)
                  Outputs:
                        result => quotient(j)
                        rdy    => quotient_rdy)

                  rslt_m_i(i,j) <= quotient(j)

            end generate;
```

Figure 21: Conditional component instantiation for Crout pipeline (divide).

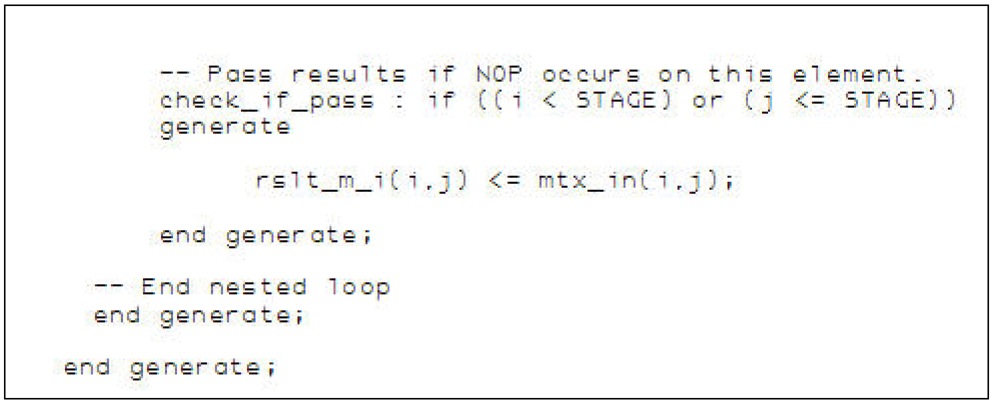
In the first section of the instantiation of components that create a Crout pipeline stage, dividers are placed in the first row where dividers are required to be instantiated given the current stage. The only exception is the first element of the row since this element is used as the divisor of the current stage. The flag "quotient_rdy" indicates when division is complete and that the upcoming multiply-subtract logic can begin processing with the quotient. Finally, the quotient itself can be forwarded to the next stage since no other operations are required on this particular element of the matrix. To be clear, no values of the current stage will be forwarded to the next stage until the whole divide/multiply/subtract function is complete in the actual implementation!



```
-- Instantiate a mult/subtract if we're "under the
-- dividers"....
check_if_multsub : if ((i>STAGE) and (j>STAGE))
generate

      mult_then_sub function (
      Inputs:


            -- New data flag
            new_data    => quotient_rdy

            -- Input operands...
            quot_in     => quotient(j)
            mlt_in      => mtx_in(i,STAGE)
            sbt_in      => mtx_in(i,j)

      Outputs:

            -- Result of above computations.
            result_rdy  => rslt_rdy
            result      => rslt_m_i(i,j)

            )

end generate
```

Figure 22: Conditional component instantiation for Crout pipeline (multiply-subtract)

The above conditional instantiation determines placement of the “multiply-then-subtract” components created in the previous section. Again, this component starts processing new data once the floating point division is complete. Once the “rslt_rdy” flag is asserted, the results on the term “rslt_m_i” are ready for forwarding to the next stage. In the actual hardware implementation, the assertion of “rslt_rdy” indicates that the stage is ready to accept a new matrix.



```
        -- Pass results if NOP occurs on this element.
        check_if_pass : if ((i < STAGE) or (j <= STAGE))
        generate

              rslt_m_i(i,j) <= mtx_in(i,j);

        end generate;

    -- End nested loop
    end generate;

  end generate;
```

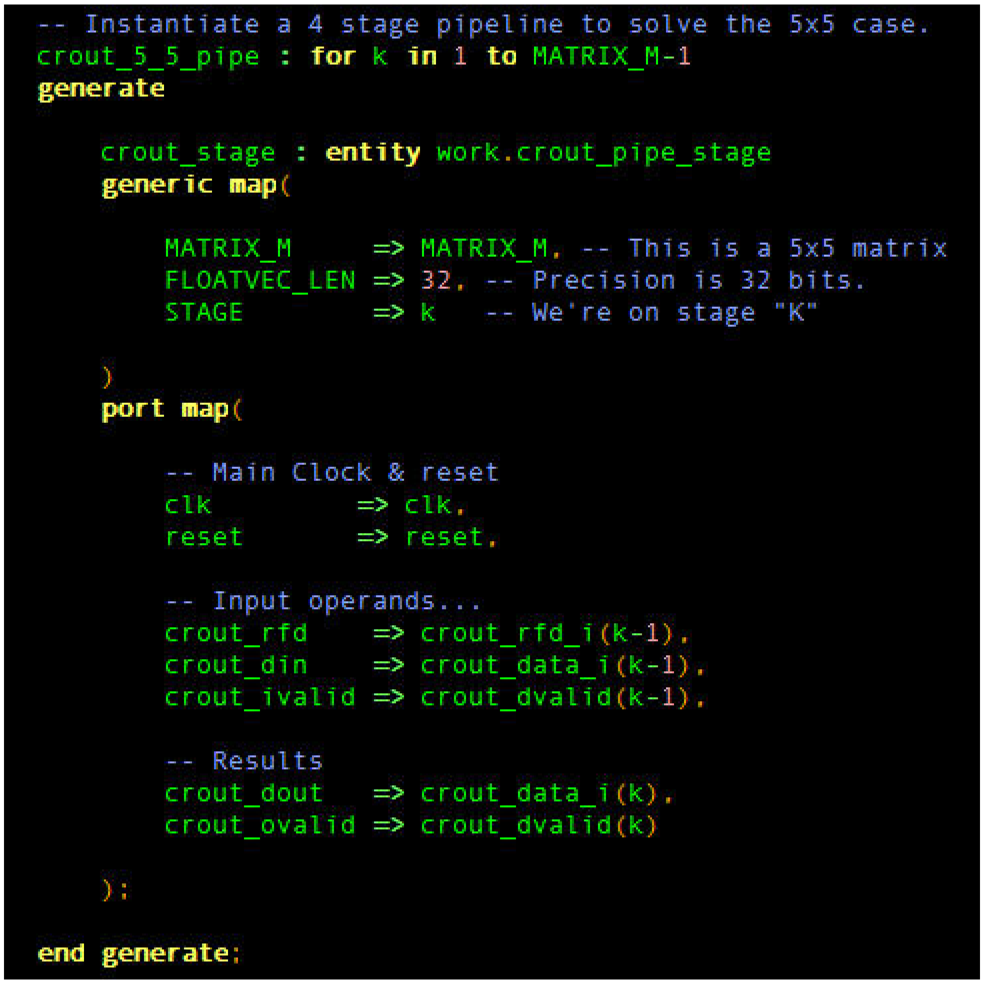
Figure 23: Conditional component instantiation for Crout pipeline (pass current element).

With the divide, multiply, and subtraction components instantiated, the final "component" is instantiated. Depending on the stage being generated, some elements require no mathematical functions performed. Therefore, the current input to the stage is simply forwarded to the output of the stage. Nothing is changed.

In general, this pseudocode creates a single stage of the 3D Crout pipeline. While the actual VHDL solution resembles this pseudocode, it is important to note that some additional conditions must be added to prevent unwanted latches during synthesis. These conditions are included in the source code provided in the Appendix and design files. With the pipeline stage created, one can proceed to the next level of hierarchy: the creation of the 5x5 pipeline with additional circuitry to drive data through the pipeline.

4.2.3 Complete Crout Pipeline and Associated Test Circuitry

Now that a single, generic stage of the Crout pipeline is complete, the actual 5x5 processing pipeline is created as well as additional support circuitry required to test the pipeline on an FPGA. First, at the next level of hierarchy up from the creation of the Crout single pipeline stage, the actual pipeline requires instantiation. Since the Crout pipeline stage was made using generics, this is a relatively painless process. One simply uses a generate loop to create the four stages required to solve a 5x5 matrix. The following is the actual VHDL instantiation used in the design.



```
-- Instantiate a 4 stage pipeline to solve the 5x5 case.
crout_5_5_pipe : for k in 1 to MATRIX_M-1
generate

    crout_stage : entity work.crout_pipe_stage
    generic map(

        MATRIX_M     => MATRIX_M, -- This is a 5x5 matrix
        FLOATVEC_LEN => 32, -- Precision is 32 bits.
        STAGE        => k   -- We're on stage "K"

    )
    port map(

        -- Main Clock & reset
        clk         => clk,
        reset       => reset,

        -- Input operands...
        crout_rfd    => crout_rfd_i(k-1),
        crout_din    => crout_data_i(k-1),
        crout_ivalid => crout_dvalid(k-1),

        -- Results
        crout_dout   => crout_data_i(k),
        crout_ovalid => crout_dvalid(k)

    );

end generate;
```

Figure 24: 5x5 Crout Pipeline Instantiation

The Crout pipeline is configured by the MATRIX_M, STAGE, and FLOATVEC_LEN vectors. MATRIX_M is the size of the matrix (i.e. 5x5), STAGE is used by the generate loop to allow the lower level "crout_pipe_stage" logic know the stage it is creating, and FLOATVEC_LEN is a generic provided for future use should a user want to expand the width of the elements of the matrix being processed. The generate loop insures that the outputs of stage 1 connect to the inputs of stage 2, the outputs of stage 2 connect to the inputs of stage 3, etc.

Overall, the control signals are rather straightforward. The signal “crout_rfd” indicates that a new matrix can be pushed into the pipeline. This signal is only used by the logic responsible for pushing a new matrix into the pipeline only when the first stage is not busy. The “crout_dvalid” signal indicates that valid data is being pushed into the pipeline. At the output, a simple signal called “crout_dvalid” qualifies the matrix data on the output “crout_data_i”.

Driving the pipeline requires some additional circuitry as seen in the following block diagram. Since streaming data is not available, circuitry must be added to mimic matrix data streaming into the Crout pipeline.

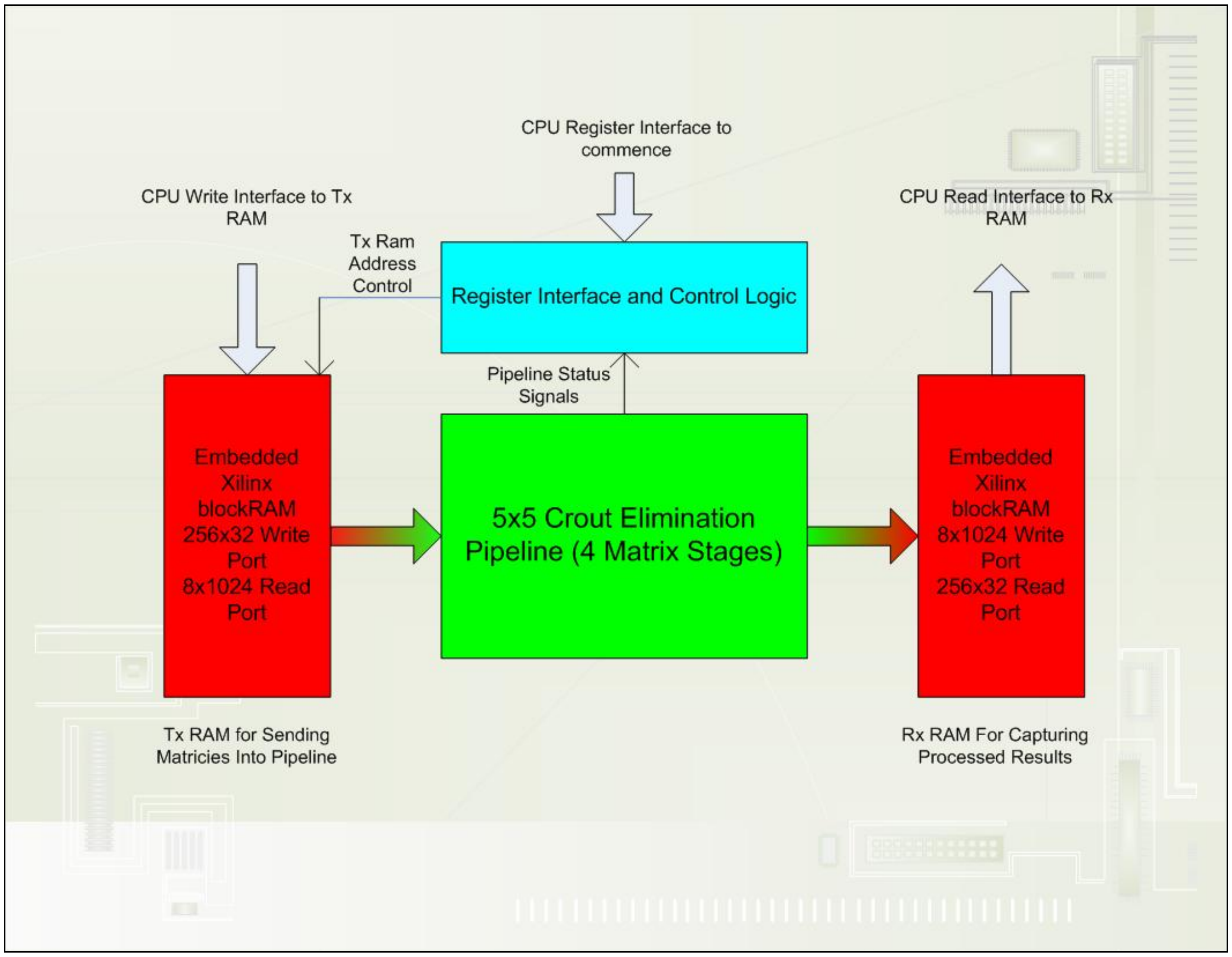

CPU Register Interface to commence
CPU Write Interface to Tx RAM
CPU Read Interface to Rx RAM
Tx Ram Address Control
Register Interface and Control Logic
Pipeline Status Signals
Embedded Xilinx blockRAM 256x32 Write Port 8x1024 Read Port
5x5 Crout Elimination Pipeline (4 Matrix Stages)
Embedded Xilinx blockRAM 8x1024 Write Port 256x32 Read Port
Tx RAM for Sending Matricies Into Pipeline
Rx RAM For Capturing Processed Results

Figure 25: Block Diagram of Crout Pipeline Test Circuit

Overall, two memories (generated by Xilinx Coregen) are required to both send and receive matrices via a standard CPU interface (i.e. address, 32 bit data, write enable, etc.). A feature of Xilinx embedded memories is the ability to configure varying widths on the read and write ports of the blockRAM. For matrix loading purposes, the user loads all matrix data into the write side of the 256x32 transmit RAM in row major format. The size of the SRAM constrains the user to loading up to 8 matrices at a time. Be aware that this is not due to a limitation of the Crout pipeline: it can obviously service more than 8 matrices at a time provided the user waits for the first stage of the pipeline to be free before pushing another matrix into the pipeline. The read port of the transmit RAM is 1024 bits wide with matrix element (1,1) corresponding to bits (31:0) of this vector. Element (1,2) would be bits (63:32) and so on. Since a 5x5 matrix only requires 25 elements (800 bits), all remaining bits are unused. The user must account for these unused bits when loading the transmit RAM (i.e. matrices must be loaded on 32 DWORD boundaries).

The "Register Interface and Control Logic" provides a means for a user to start the transfer of matrix data from the transmit RAM to the Crout pipeline. It also monitors the "ready for data" signal so that the "first matrix stage is free" rule is obeyed. Finally, it subsequently increments the read address pointer of the transmit RAM. This register is written with a "number of matrices to send" value after loading the transmit RAM. This register write subsequently starts pushing matrix data into the pipeline.

Processed data is captured by the receive RAM. The write port of this RAM is 8x1024 bits wide. The format of this data is the same as the transmit RAM. The user can read data out via the read port of the receive RAM using the CPU interface. The CPU interface is identical to the transmit RAM (256x32). The unused bits of the matrix vector remain unused, but the user must account for these unused locations in receive RAM. Matrices are stored on 32 DWORD boundaries.

4.3 PIPELINE AND TEST CIRCUITRY SIMULATION

Now that the concept is developed, it is best to simulate the design to insure that the circuit is behaving as expected functionally! For this purpose, a test environment is created in

VHDL and the logic is simulated using Modelsim. While additional logic, namely a simple microcontroller and external memory controller for test code storage, are added later, it is rather useless to simulate the entire chip because our component of interest is the Crout pipeline. Simulation of a full CPU for this application is rather unnecessary. As a result, the testbench shall mimic the actions of the CPU.

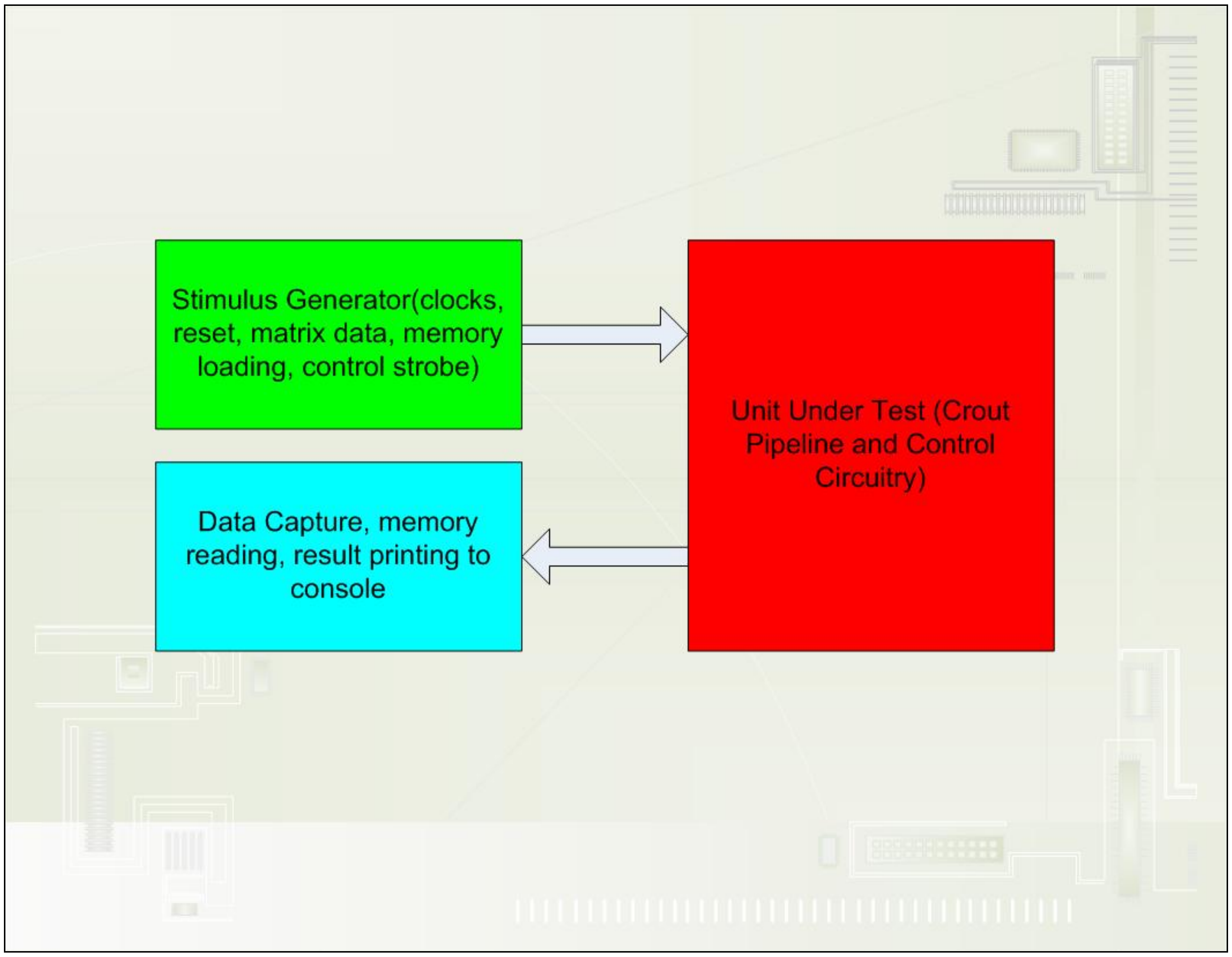

Stimulus Generator(clocks, reset, matrix data, memory loading, control strobe)
Data Capture, memory reading, result printing to console
Unit Under Test (Crout Pipeline and Control Circuitry)

Figure 26: Testbench block diagram

Overall, the stimulus generator basically mimics the functions of the soon-to-be-added CPU as well as the core FPGA functions (clock management, etc). The testbench code instantiates the Crout logic discussed in previous sections as well as components to simulate clock generation, power-on reset, and data driving. The user of the testbench simply loads VHDL "real" values into the transmit RAM. Since VHDL inherently lacks the concept of an IEEE754 floating point data-type, this task may seem rather difficult at first. Fortunately, the proposed IEEE floating point library for synthesis has procedures to provide "real to IEEE754" and vice-versa to substantially ease the amount of work required to verify the Crout pipeline. Basically, these functions can convert a real number to the equivalent IEEE754 logic vector.

Once the user loads the matrix data via the testbench procedures that mimic CPU accesses, the user is required to strobe the control logic with the number of matrices to process. The simulation displays the expected solutions before proceeding. The testbench then polls the "busy flag" to wait for processing to complete. Once complete, the testbench displays the results. Moreover, assertions are added in the Crout code itself to show intermediate results as data is passing from stage to stage in the pipeline. Obviously, these assertions are not synthesized, but are useful for feedback during development.

The following figures detail the results of a simulation run of the Crout pipeline. They are intended to show the critical operations that occur during the operation of the pipeline and are by no means intended to show the reader each and every value being sent through the pipeline. They are intended to show, via simulation, that the pipeline is behaving as expected.

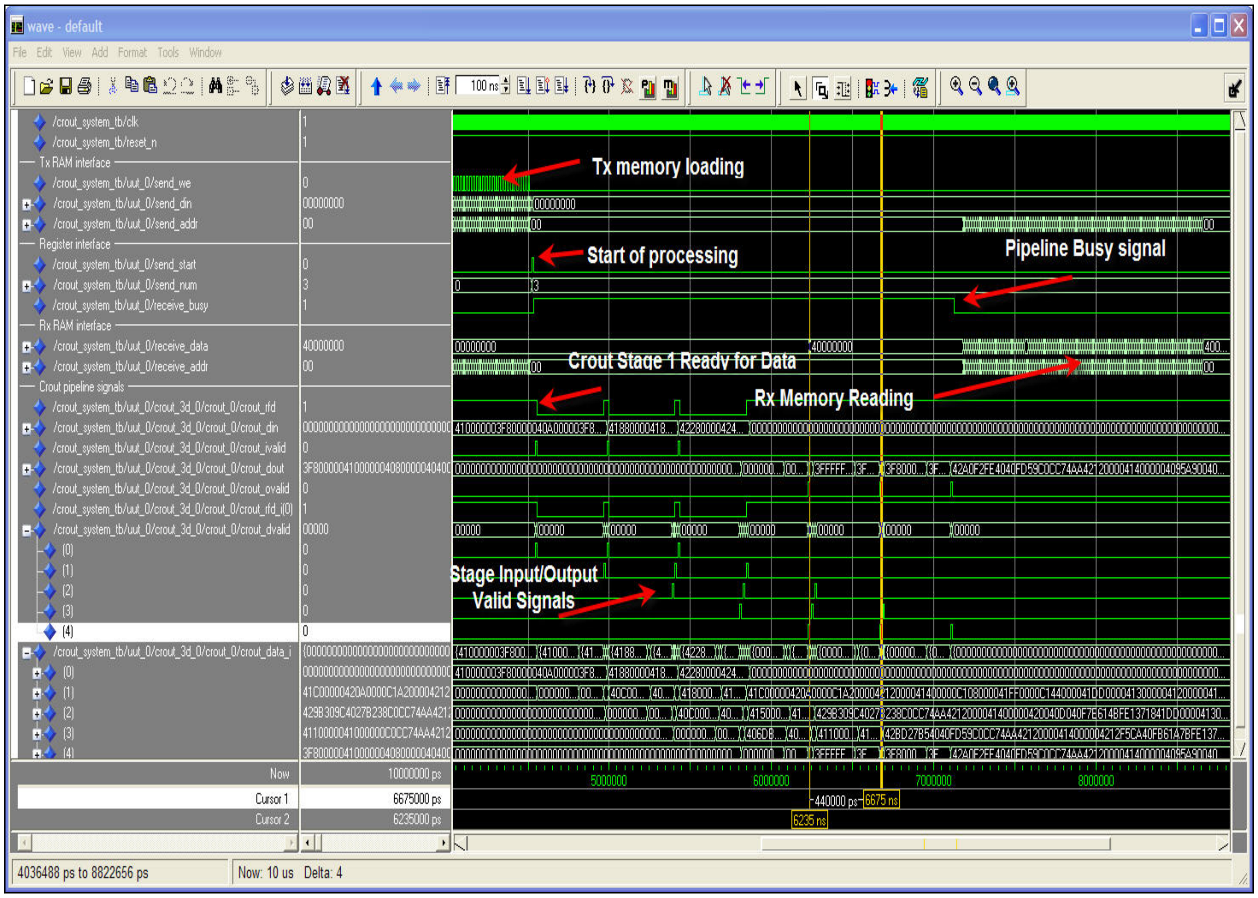
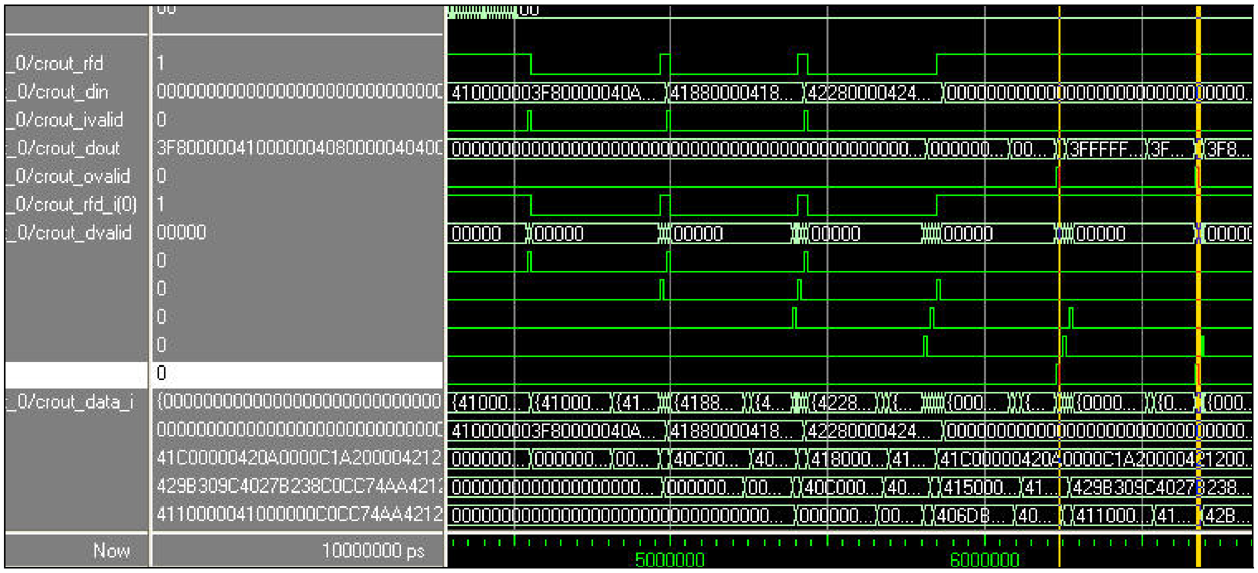

wave - default
Tx memory loading
Start of processing
Pipeline Busy signal
Crout Stage 1 Ready for Data
Rx Memory Reading
Stage Input/Output Valid Signals
/crout_system_tb/clk
/crout_system_tb/reset_n
Tx RAM interface
/crout_system_tb/uut_0/send_we
/crout_system_tb/uut_0/send_din
/crout_system_tb/uut_0/send_addr
Register interface
/crout_system_tb/uut_0/send_start
/crout_system_tb/uut_0/send_num
/crout_system_tb/uut_0/receive_busy
Rx RAM interface
/crout_system_tb/uut_0/receive_data
/crout_system_tb/uut_0/receive_addr
Crout pipeline signals
/crout_system_tb/uut_0/crout_3d_0/crout_rfd
/crout_system_tb/uut_0/crout_3d_0/crout_din
/crout_system_tb/uut_0/crout_3d_0/crout_ivalid
/crout_system_tb/uut_0/crout_3d_0/crout_dout
/crout_system_tb/uut_0/crout_3d_0/crout_ovalid
/crout_system_tb/uut_0/crout_3d_0/crout_rfd_i(0)
/crout_system_tb/uut_0/crout_3d_0/crout_dvalid
/crout_system_tb/uut_0/crout_3d_0/crout_data_i
Now 10000000 ps
Cursor 1 6675000 ps
Cursor 2 6235000 ps
440000 ps
6675 ns
6235 ns
4036488 ps to 8822656 ps
Now: 10 us Delta: 4

Figure 27: Pipeline simulation results

The waveforms above show that the user loads matrix data into the pipeline via the Tx memory interface. Afterwards, the user strobes the “start processing” register with a value of three. The pipeline busy signal goes active immediately following this strobe. Next, one can see the stages receiving matrix data via the “stage input/output signals”. The time between each of these cycles is 44 clock cycles. Recall that 26 cycles are expected for the division, 7 for the subtraction, and 8 for the multiplication. This constitutes 41 cycles. The additional 3 cycles are primarily due to the control logic overhead (thus the wider pulse width of “ready_for_data”). However, these timing diagrams show that a partial matrix solution is output from a stage every

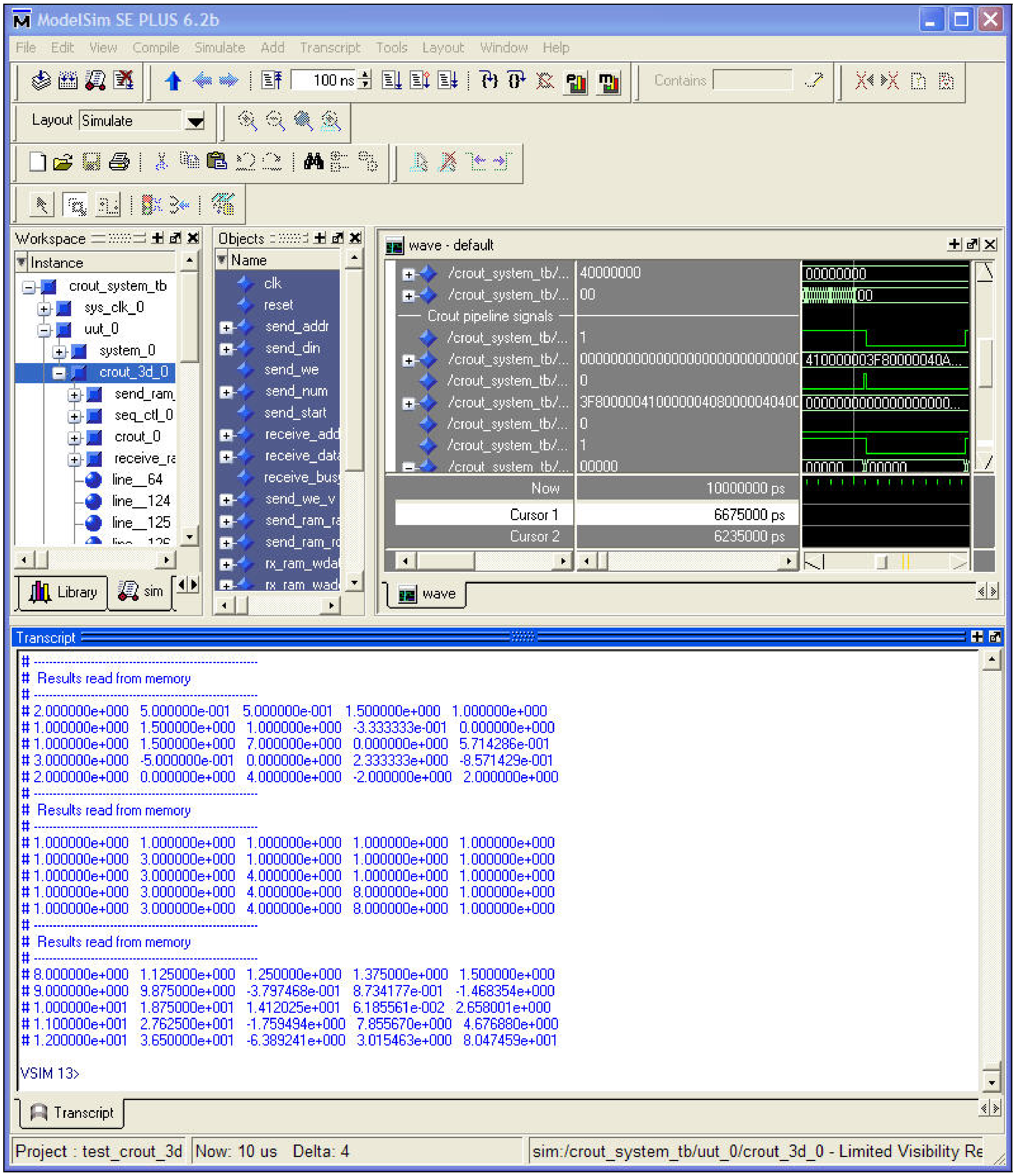
44 cycles. Therefore, after the first matrix arrives at the output of the pipeline, a new matrix solution is present every 44 clock cycles. This is the intended behavior. The next figure shows this sequencing up close due to the critical nature of this operation.



```
                     00
_0/crout_rfd         1
_0/crout_din         0000000000000000000000000000000
_0/crout_ivalid      0
_0/crout_dout        3F8000004100000040800000404OC
_0/crout_ovalid      0
_0/crout_rfd_i(0)    1
_0/crout_dvalid      00000
                     0
                     0
                     0
                     0
                     0
_0/crout_data_i      {000000000000000000000000000000
                     0000000000000000000000000000000
                     41C00000420A0000C1A200004212
                     429B309C4027B238C0CC74AA4212
                     4110000041000000C0CC74AA4212
Now                  10000000 ps
```

Figure 28: Close up of stage to stage sequencing

Next, so that the user can actually make sense of data as it is read from the Rx RAMs, the IEEE proposed floating point library for VHDL is used to convert the 32 bit floating point logic vectors to user friendly VHDL “real” values. In simulation, the console displays the results of each matrix processed. The following screenshot shows the results of the Crout pipeline as displayed on the simulation console. The real numbers displayed are the output of the IEEE754_to_real functions that are based on the IEEE floating point for synthesis library.



```
/crout_system_tb/...   40000000
/crout_system_tb/...   00
Crout pipeline signals
/crout_system_tb/...   1
/crout_system_tb/...   0000000000000000000000000000
/crout_system_tb/...   0
/crout_system_tb/...   3F80000041000000408000004040
/crout_system_tb/...   0
/crout_system_tb/...   1
/crout_system_tb/...   00000
Now        10000000 ps
Cursor 1    6675000 ps
Cursor 2    6235000 ps
```

```
# -----------------------------------------------------------
#  Results read from memory
# -----------------------------------------------------------
# 2.000000e+000  5.000000e-001  5.000000e-001  1.500000e+000  1.000000e+000
# 1.000000e+000  1.500000e+000  1.000000e+000  -3.333333e-001  0.000000e+000
# 1.000000e+000  1.500000e+000  7.000000e+000  0.000000e+000  5.714286e-001
# 3.000000e+000  -5.000000e-001  0.000000e+000  2.333333e+000  -8.571429e-001
# 2.000000e+000  0.000000e+000  4.000000e+000  -2.000000e+000  2.000000e+000
# -----------------------------------------------------------
#  Results read from memory
# -----------------------------------------------------------
# 1.000000e+000  1.000000e+000  1.000000e+000  1.000000e+000  1.000000e+000
# 1.000000e+000  3.000000e+000  1.000000e+000  1.000000e+000  1.000000e+000
# 1.000000e+000  3.000000e+000  4.000000e+000  1.000000e+000  1.000000e+000
# 1.000000e+000  3.000000e+000  4.000000e+000  8.000000e+000  1.000000e+000
# 1.000000e+000  3.000000e+000  4.000000e+000  8.000000e+000  1.000000e+000
# -----------------------------------------------------------
#  Results read from memory
# -----------------------------------------------------------
# 8.000000e+000  1.125000e+000  1.250000e+000  1.375000e+000  1.500000e+000
# 9.000000e+000  9.875000e+000  -3.797468e-001  8.734177e-001  -1.468354e+000
# 1.000000e+001  1.875000e+001  1.412025e+001  6.185561e-002  2.658001e+000
# 1.100000e+001  2.762500e+001  -1.759494e+000  7.855670e+000  4.676880e+000
# 1.200000e+001  3.650000e+001  -6.389241e+000  3.015463e+000  8.047459e+001

VSIM 13>
```

```
Project : test_crout_3d   Now: 10 us   Delta: 4   sim:/crout_system_tb/uut_0/crout_3d_0 - Limited Visibility Re
```

Figure 29: Simulation console as results are read from memory

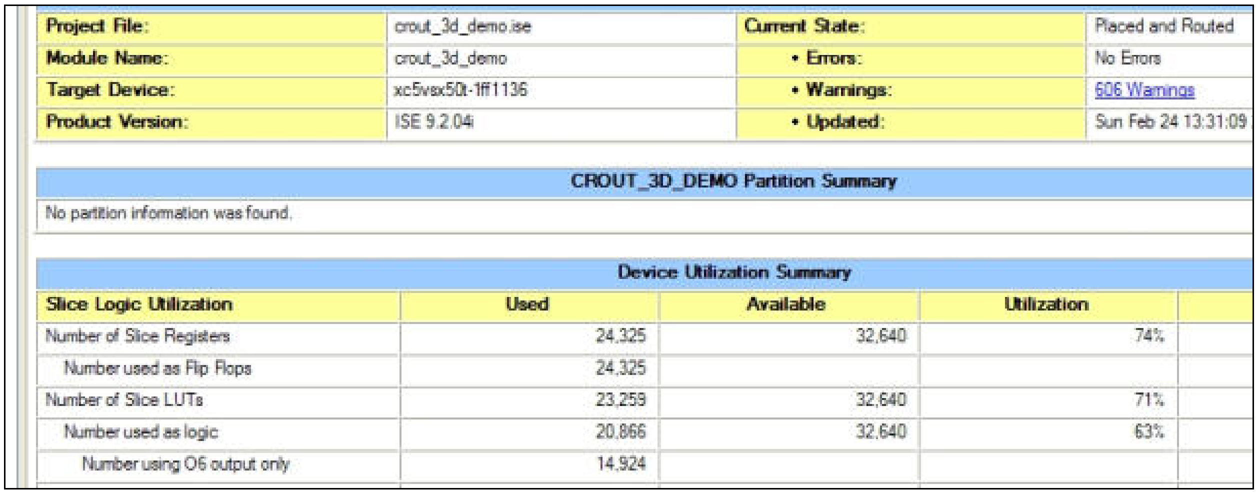
Observing these results, it appears safe to move to the synthesis stage to actually get the circuit up and running in hardware. Just before a full system is synthesized, it is somewhat obvious that a quick test-run of the pipeline in synthesis would be beneficial to see how well actual results match expected results. Synthesizing the system at this point of the design allows users to determine logic utilization of the Crout pipeline before all of the CPU logic overhead is added to the design.

4.4 SYSTEM SYNTHESIS

This section details the synthesis of the system. First, a test run is performed to determine if the Crout pipeline will fit in the target device. To facilitate hardware testing of the Crout pipeline, some logic must be available to add a simple Xilinx Microblaze CPU to load floating point matrices into the Crout pipeline. After the test run, a Microblaze base system is created for testing the pipeline. Next, final area results are presented before demonstrating a fully working system.

4.4.1 Pipeline "Test Synthesis"

Earlier in the design process, area estimates were calculated to determine the feasibility of fitting the 5x5 Crout pipeline into a Xilinx Virtex 5 SXT 50 device. In order to check the accuracy of these estimates, it would be best to synthesize the design with just the pipeline, the minimal control logic, and the memories. No constraints are used with this design, so only "raw" results are presented. The tools will not work to meet any timing goals.



Project File:	crout_3d_demo.ise	Current State:	Placed and Routed
Module Name:	crout_3d_demo	• Errors:	No Errors
Target Device:	xc5vsx50t-1ff1136	• Warnings:	606 Warnings
Product Version:	ISE 9.2.04i	• Updated:	Sun Feb 24 13:31:09

CROUT_3D_DEMO Partition Summary
No partition information was found.

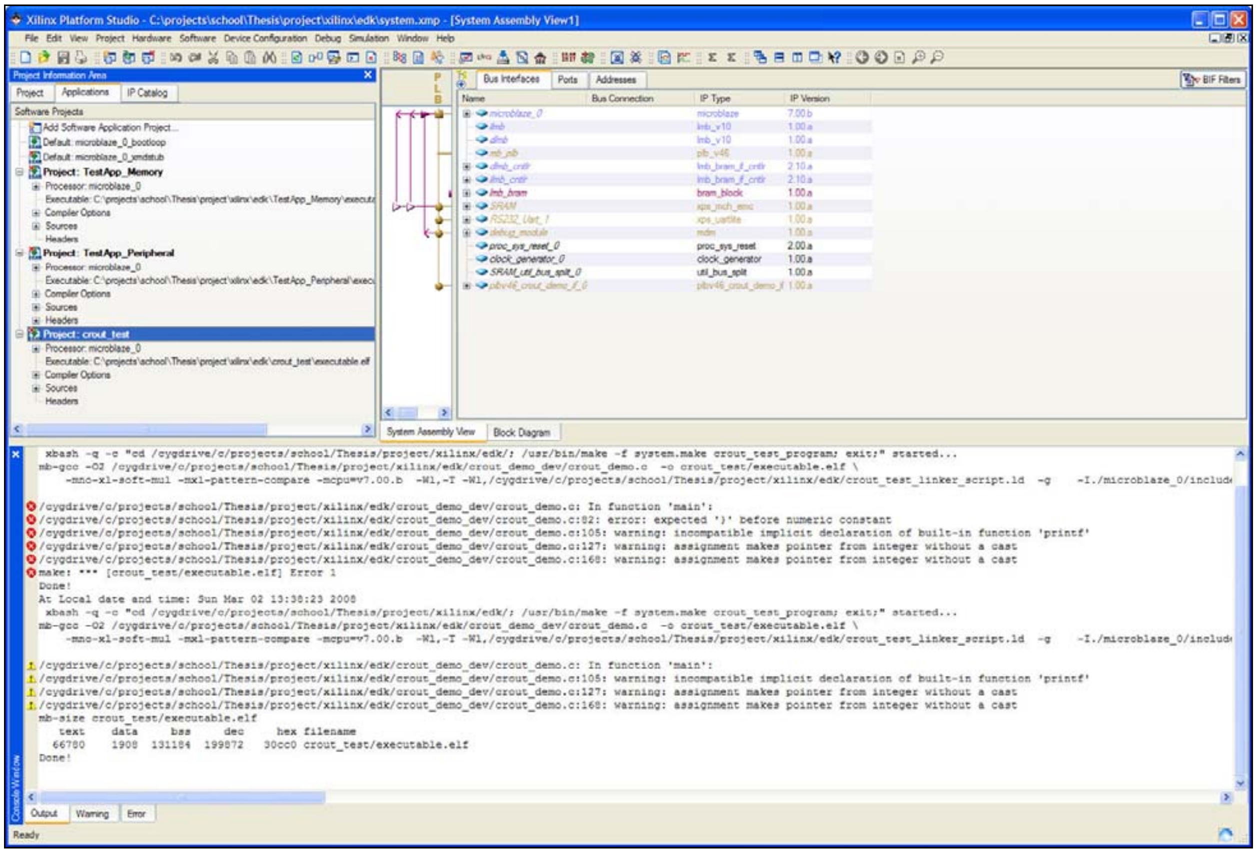
Device Utilization Summary			
Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	24,325	32,640	74%
Number used as Flip Flops	24,325		
Number of Slice LUTs	23,259	32,640	71%
Number used as logic	20,866	32,640	63%
Number using O6 output only	14,924		

Figure 30: Pre "full system" synthesis results

After synthesis and place and route of the “reduced” system, Xilinx reports that 24,325 flip flops are currently used along with 23,259 LUTs. This is somewhat higher than the 22,820 LUTs and 20,720 flip flops expected during the logic estimate stage. This additional logic might be attributed to the added control logic and rather large memory structures, but logic estimates typically vary due to numerous factors such as FPGA architecture, clock speed requirements, and synthesis algorithm seeds. Also, many flip flops are used to store intermediate results at each matrix stage of the pipeline. Overall, the numbers indicate that adding a CPU to test the system in hardware is not an issue as the CPU, a Xilinx Microblaze, and the peripherals required consume far fewer than the 10,000 logic elements available in the device.

4.4.2 Xilinx EDK and Microblaze

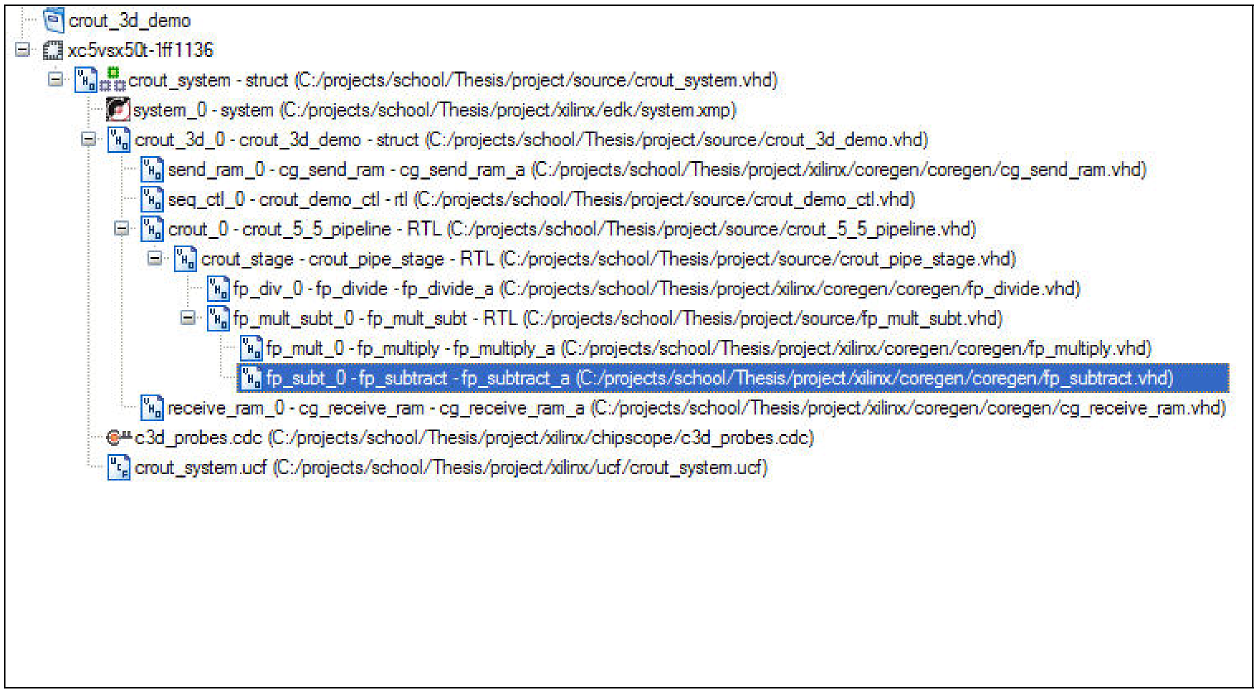
To facilitate testing of the Crout pipeline, a CPU is added to the design. This allows a user to write basic C programs to load the floating point data into the transmit RAMs as well as read received data from the receive RAMs once processing is finished. Moreover, a simple UART is added so that results are displayed on any terminal program. Finally, the compiled executable is stored in an external SRAM because the amount of remaining on-chip memory does not suffice. Most on-chip SRAM is being used by the transmit and receive RAMs. The Microblaze CPU, the UART, and external memory interface are provided by Xilinx via EDK.



```
 xbash -q -c "cd /cygdrive/c/projects/school/Thesis/project/xilinx/edk/; /usr/bin/make -f system.make crout_test_program; exit;" started...
mb-gcc -O2 /cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c  -o crout_test/executable.elf \
    -mno-xl-soft-mul -mxl-pattern-compare -mcpu=v7.00.b  -Wl,-T -Wl,/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_test_linker_script.ld  -g    -I./microblaze_0/includ

/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c: In function 'main':
/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c:82: error: expected ')' before numeric constant
/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c:105: warning: incompatible implicit declaration of built-in function 'printf'
/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c:127: warning: assignment makes pointer from integer without a cast
/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c:168: warning: assignment makes pointer from integer without a cast
make: *** [crout_test/executable.elf] Error 1
Done!
At Local date and time: Sun Mar 02 13:38:23 2008
 xbash -q -c "cd /cygdrive/c/projects/school/Thesis/project/xilinx/edk/; /usr/bin/make -f system.make crout_test_program; exit;" started...
mb-gcc -O2 /cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c  -o crout_test/executable.elf \
    -mno-xl-soft-mul -mxl-pattern-compare -mcpu=v7.00.b  -Wl,-T -Wl,/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_test_linker_script.ld  -g    -I./microblaze_0/includ

/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c: In function 'main':
/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c:105: warning: incompatible implicit declaration of built-in function 'printf'
/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c:127: warning: assignment makes pointer from integer without a cast
/cygdrive/c/projects/school/Thesis/project/xilinx/edk/crout_demo_dev/crout_demo.c:168: warning: assignment makes pointer from integer without a cast
mb-size crout_test/executable.elf
   text    data     bss     dec     hex filename
  66780    1908  131184  199872   30cc0 crout_test/executable.elf
Done!
```

Figure 31: EDK session showing a simple processor system

EDK handles the creation of a CPU system as well as basic peripherals such as UARTs and memory controllers. A custom peripheral is created to interface to the Crout pipeline test structure. The control registers are mapped to one area of the Microblaze's PLB bus while the Tx and Rx memories are mapped to other regions. EDK also handles the compilation of code written for the Microblaze. Some errors in the test code were caught at first, but were subsequently fixed and compilation is complete as shown in Figure 31. The total memory footprint of the code is almost 200KB without any optimizations set. This is far less than the 4MB available in external SRAM.



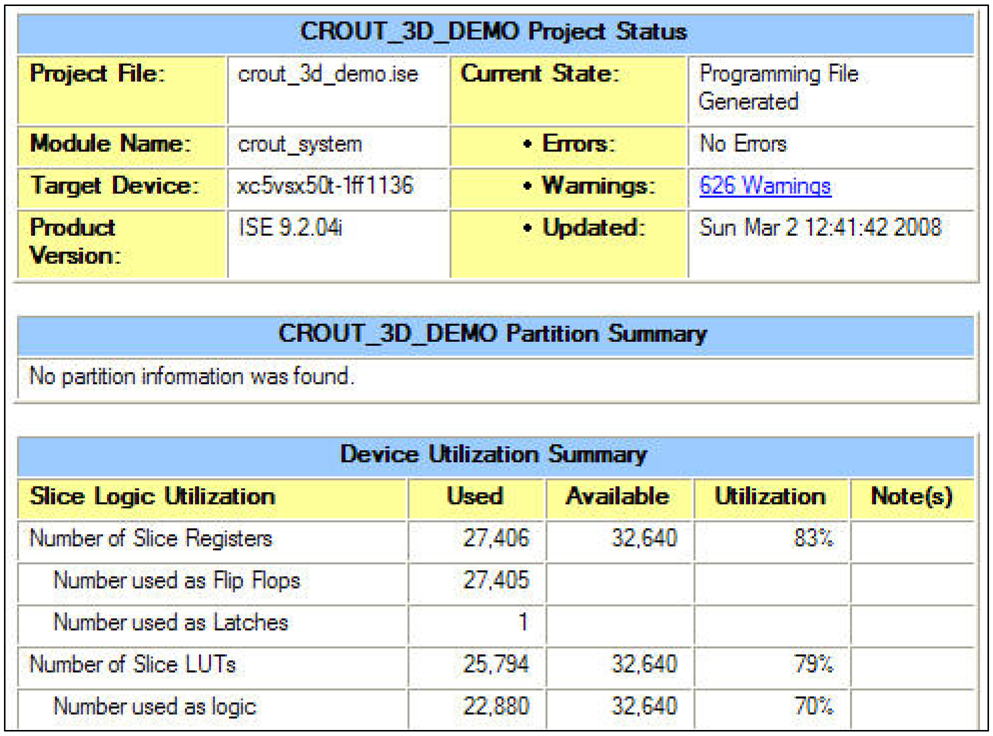
```
crout_3d_demo
xc5vsx50t-1ff1136
  crout_system - struct (C:/projects/school/Thesis/project/source/crout_system.vhd)
    system_0 - system (C:/projects/school/Thesis/project/xilinx/edk/system.xmp)
    crout_3d_0 - crout_3d_demo - struct (C:/projects/school/Thesis/project/source/crout_3d_demo.vhd)
      send_ram_0 - cg_send_ram - cg_send_ram_a (C:/projects/school/Thesis/project/xilinx/coregen/coregen/cg_send_ram.vhd)
      seq_ctl_0 - crout_demo_ctl - rtl (C:/projects/school/Thesis/project/source/crout_demo_ctl.vhd)
      crout_0 - crout_5_5_pipeline - RTL (C:/projects/school/Thesis/project/source/crout_5_5_pipeline.vhd)
        crout_stage - crout_pipe_stage - RTL (C:/projects/school/Thesis/project/source/crout_pipe_stage.vhd)
          fp_div_0 - fp_divide - fp_divide_a (C:/projects/school/Thesis/project/xilinx/coregen/coregen/fp_divide.vhd)
          fp_mult_subt_0 - fp_mult_subt - RTL (C:/projects/school/Thesis/project/source/fp_mult_subt.vhd)
            fp_mult_0 - fp_multiply - fp_multiply_a (C:/projects/school/Thesis/project/xilinx/coregen/coregen/fp_multiply.vhd)
            fp_subt_0 - fp_subtract - fp_subtract_a (C:/projects/school/Thesis/project/xilinx/coregen/coregen/fp_subtract.vhd)
      receive_ram_0 - cg_receive_ram - cg_receive_ram_a (C:/projects/school/Thesis/project/xilinx/coregen/coregen/cg_receive_ram.vhd)
    c3d_probes.cdc (C:/projects/school/Thesis/project/xilinx/chipscope/c3d_probes.cdc)
    crout_system.ucf (C:/projects/school/Thesis/project/xilinx/ucf/crout_system.ucf)
```

Figure 32: Complete synthesis hierarchy with Microblaze System

Once the Microblaze system is created in EDK, it is added to the synthesis hierarchy. This means that the total demonstration system is complete, and that the working design can be uploaded to the target FPGA for testing. Moreover, constraints are added via the ‘crout_system.ucf” file so that pin placement matches the peripherals on the test board. The constraints also direct the synthesis tools to meet the modest 50MHz timing requirements of the system. Moreover, a logic analyzer is embedded in the design to probe nodes in the FPGA to demonstrate the pipeline is functioning exactly as seen in simulation. This is achieved by using the Xilinx Chipscope tool and the resulting ‘c3d_probes.cdc’ file.

4.4.3 Full System Synthesis

Overall, the full system synthesis results added some significant logic to the complete device. Roughly 7,000 flip flops and 2,000 LUTs were added. While this is a significant amount of logic, 17% of the flip flops available and 20% of the LUTs remain available in the target device. By no means could another Crout stage be added; however, additional functions a user may desire may fit in the remaining area (i.e. higher speed communication to a host, etc.). This basically indicates that a 3D Crout pipeline and associated support circuitry of a non-trivial dimension is feasible in modern devices. The final part of the design now comes down to verifying the test structure created actually works in hardware!



CROUT_3D_DEMO Project Status			
Project File:	crout_3d_demo.ise	**Current State:**	Programming File Generated
Module Name:	crout_system	• **Errors:**	No Errors
Target Device:	xc5vsx50t-1ff1136	• **Warnings:**	626 Warnings
Product Version:	ISE 9.2.04i	• **Updated:**	Sun Mar 2 12:41:42 2008

CROUT_3D_DEMO Partition Summary
No partition information was found.

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	27,406	32,640	83%	
Number used as Flip Flops	27,405			
Number used as Latches	1			
Number of Slice LUTs	25,794	32,640	79%	
Number used as logic	22,880	32,640	70%	

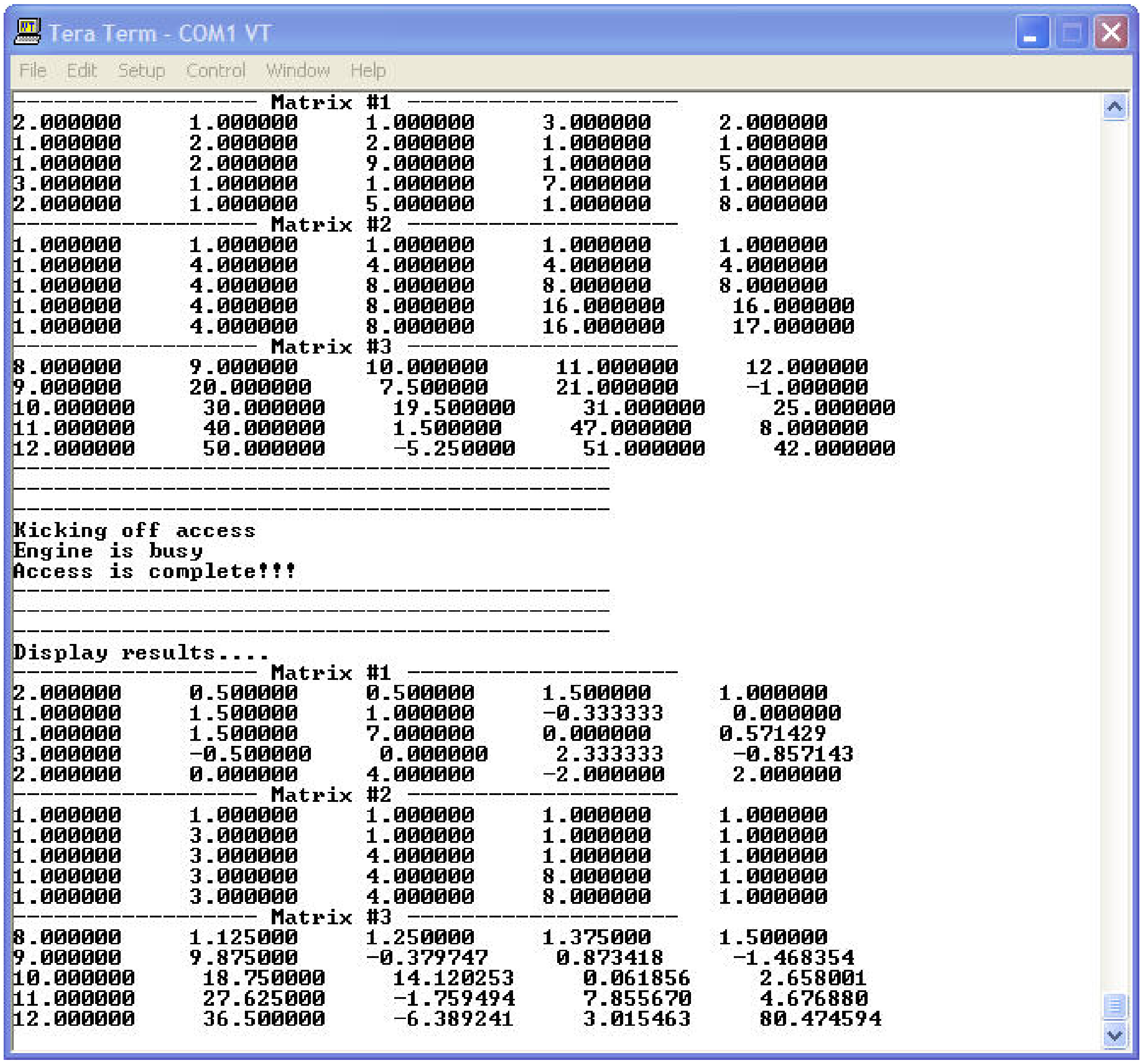
Figure 33: Final system synthesis results

4.5 HARDWARE VERIFICATION

Hardware verification is primarily divided into two parts. First, a simple C application is written for the Microblaze to verify the logic is properly processing the results expected. Next, the embedded logic analyzer is used to determine that the hardware is functioning as seen in simulation. Provided these two goals are met, the challenge of implementing a 3D, 5x5 Crout Elimination pipeline will be deemed complete!

4.5.1 Software testing

The software written to test the pipeline virtually mirrors the code written for the testbench. Three matrices are loaded into the transmit RAM. An access to the control logic is made indicating that three matrices requiring processing are loaded into transmit RAM. This access starts pushing matrix data into the pipeline. The code then polls the busy flag awaiting completion of the processing. Finally, the results are displayed to the console in decimal format because reading floating point values in hexadecimal notation is a rather difficult task for most normal humans. The following screen shot shows the output of the FPGA that is dumped to a console program running on a PC.



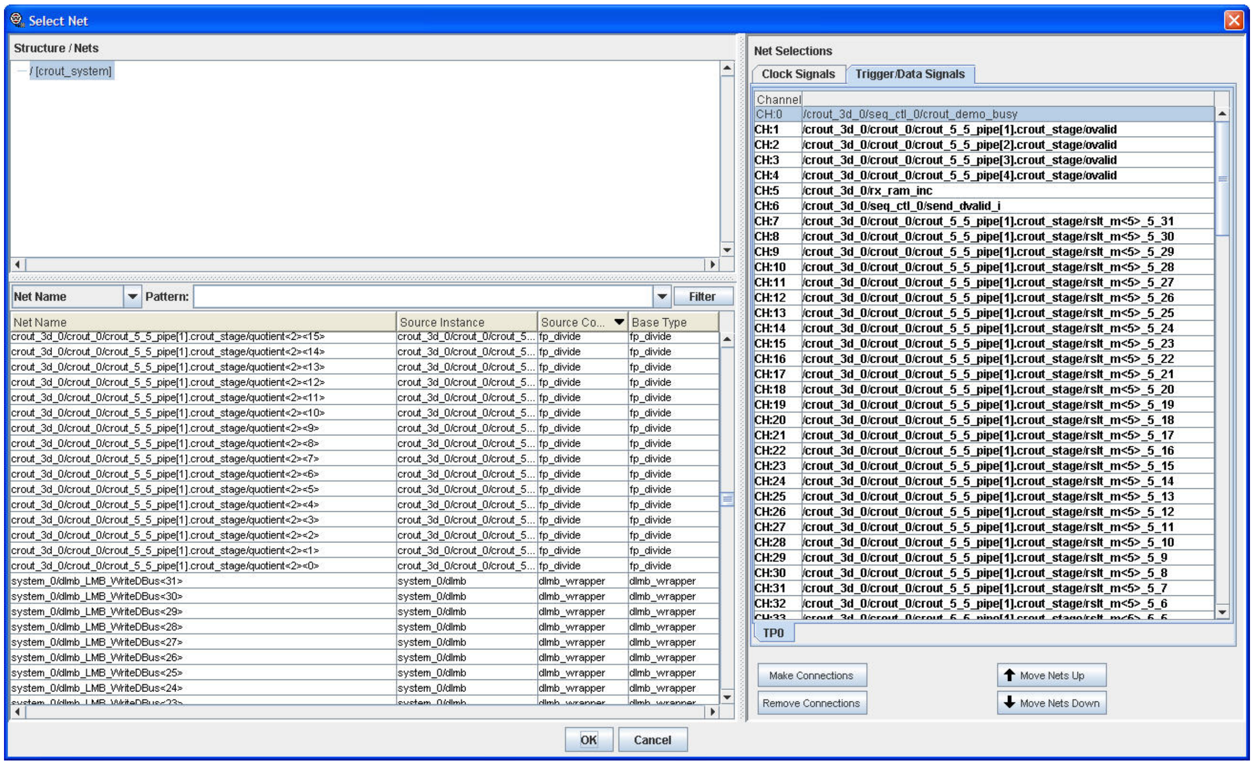
```
------------------- Matrix #1 ---------------------
2.000000     1.000000     1.000000     3.000000     2.000000
1.000000     2.000000     2.000000     1.000000     1.000000
1.000000     2.000000     9.000000     1.000000     5.000000
3.000000     1.000000     1.000000     7.000000     1.000000
2.000000     1.000000     5.000000     1.000000     8.000000
------------------- Matrix #2 ---------------------
1.000000     1.000000     1.000000     1.000000     1.000000
1.000000     4.000000     4.000000     4.000000     4.000000
1.000000     4.000000     8.000000     8.000000     8.000000
1.000000     4.000000     8.000000     16.000000     16.000000
1.000000     4.000000     8.000000     16.000000     17.000000
------------------- Matrix #3 ---------------------
8.000000     9.000000     10.000000     11.000000     12.000000
9.000000     20.000000     7.500000     21.000000     -1.000000
10.000000     30.000000     19.500000     31.000000     25.000000
11.000000     40.000000     1.500000     47.000000     8.000000
12.000000     50.000000     -5.250000     51.000000     42.000000
---------------------------------------------
---------------------------------------------
---------------------------------------------
Kicking off access
Engine is busy
Access is complete!!!
---------------------------------------------
---------------------------------------------
---------------------------------------------
Display results....
------------------- Matrix #1 ---------------------
2.000000     0.500000     0.500000     1.500000     1.000000
1.000000     1.500000     1.000000     -0.333333     0.000000
1.000000     1.500000     7.000000     0.000000     0.571429
3.000000     -0.500000     0.000000     2.333333     -0.857143
2.000000     0.000000     4.000000     -2.000000     2.000000
------------------- Matrix #2 ---------------------
1.000000     1.000000     1.000000     1.000000     1.000000
1.000000     3.000000     1.000000     1.000000     1.000000
1.000000     3.000000     4.000000     1.000000     1.000000
1.000000     3.000000     4.000000     8.000000     1.000000
1.000000     3.000000     4.000000     8.000000     1.000000
------------------- Matrix #3 ---------------------
8.000000     1.125000     1.250000     1.375000     1.500000
9.000000     9.875000     -0.379747     0.873418     -1.468354
10.000000     18.750000     14.120253     0.061856     2.658001
11.000000     27.625000     -1.759494     7.855670     4.676880
12.000000     36.500000     -6.389241     3.015463     80.474594
```

Figure 34: Output to console from system

The top three matrices are the values that are loaded into the transmit RAM. The CPU handles the conversion from decimal notation to IEE754 binary notation. From there, the CPU reports that the access is kicked off and the "Engine is Busy" message indicates that the pipeline is busy. After the busy flag indicates completion of processing, the results are displayed after the reads from receive RAM and subsequent conversion to decimal notation. Fortunately, the results are as expected! Code is written in both behavioral VHDL and Matlab based on the pseudocode detailed in Figure 2 to verify that the input matrices generate the exact same solution. So far, these results demonstrate that the design is at least working as expected in hardware, but more proof is needed to know that the design is working with cycle-for-cycle accuracy!

4.5.2 Internal hardware probing

All major FPGA vendors provide tools to probe internal nodes in FPGAs in an effort to assist in debugging. Xilinx is no different with their Chipscope tools. These tools allow a user to select nodes in the FPGA, configure options to allow various trigger modes not unlike one finds on a logic analyzer, and displays the results in a graphical format. For the Crout pipeline, these tools are valuable to prove that the design is working exactly as expected.

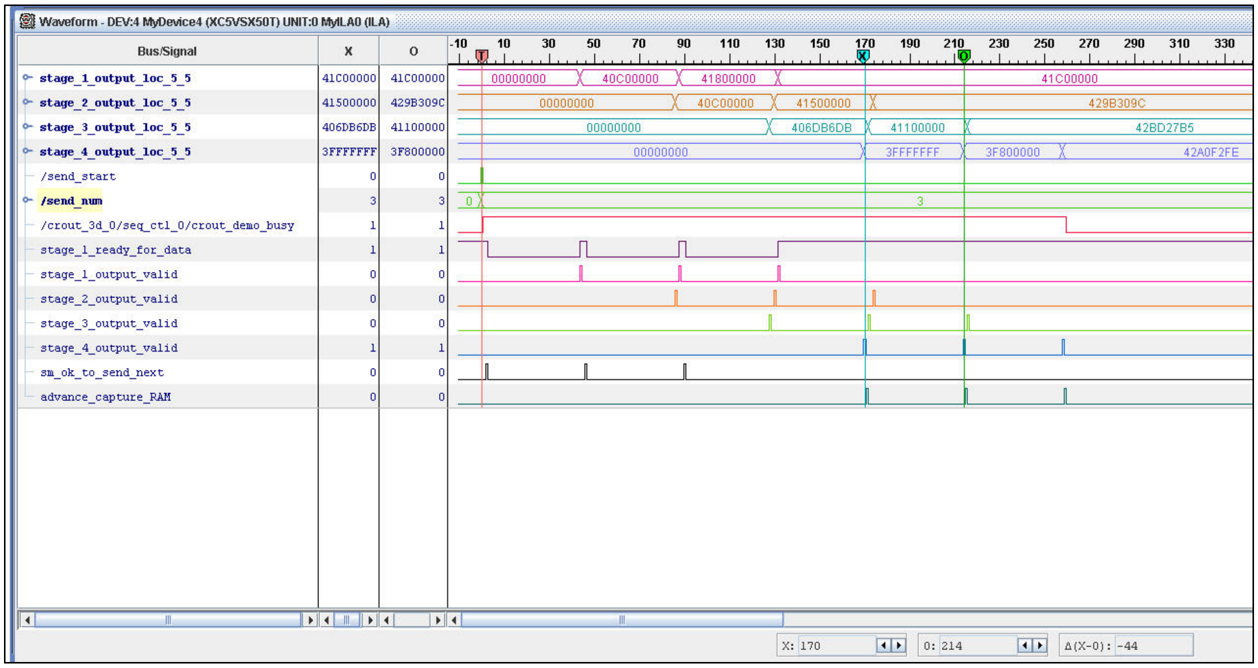


Net Name	Source Instance	Source Co...	Base Type
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><15>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><14>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><13>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><12>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><11>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><10>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><9>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><8>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><7>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><6>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><5>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><4>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><3>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><2>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><1>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/quotient<2><0>	crout_3d_0/crout_0/crout_5...	fp_divide	fp_divide
system_0/dlmb_LMB_WriteDBus<31>	system_0/dlmb	dlmb_wrapper	dlmb_wrapper
system_0/dlmb_LMB_WriteDBus<30>	system_0/dlmb	dlmb_wrapper	dlmb_wrapper
system_0/dlmb_LMB_WriteDBus<29>	system_0/dlmb	dlmb_wrapper	dlmb_wrapper
system_0/dlmb_LMB_WriteDBus<28>	system_0/dlmb	dlmb_wrapper	dlmb_wrapper
system_0/dlmb_LMB_WriteDBus<27>	system_0/dlmb	dlmb_wrapper	dlmb_wrapper
system_0/dlmb_LMB_WriteDBus<26>	system_0/dlmb	dlmb_wrapper	dlmb_wrapper
system_0/dlmb_LMB_WriteDBus<25>	system_0/dlmb	dlmb_wrapper	dlmb_wrapper
system_0/dlmb_LMB_WriteDBus<24>	system_0/dlmb	dlmb_wrapper	dlmb_wrapper

Channel	
CH:0	/crout_3d_0/seq_ctl_0/crout_demo_busy
CH:1	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/ovalid
CH:2	/crout_3d_0/crout_0/crout_5_5_pipe[2].crout_stage/ovalid
CH:3	/crout_3d_0/crout_0/crout_5_5_pipe[3].crout_stage/ovalid
CH:4	/crout_3d_0/crout_0/crout_5_5_pipe[4].crout_stage/ovalid
CH:5	/crout_3d_0/rx_ram_inc
CH:6	/crout_3d_0/seq_ctl_0/send_dvalid_i
CH:7	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_31
CH:8	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_30
CH:9	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_29
CH:10	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_28
CH:11	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_27
CH:12	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_26
CH:13	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_25
CH:14	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_24
CH:15	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_23
CH:16	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_22
CH:17	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_21
CH:18	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_20
CH:19	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_19
CH:20	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_18
CH:21	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_17
CH:22	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_16
CH:23	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_15
CH:24	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_14
CH:25	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_13
CH:26	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_12
CH:27	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_11
CH:28	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_10
CH:29	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_9
CH:30	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_8
CH:31	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_7
CH:32	/crout_3d_0/crout_0/crout_5_5_pipe[1].crout_stage/rslt_m<5>_5_6

Figure 35: Chipscope node probing

The above figure demonstrates a typical session with Chipscope. To help prove the pipeline is behaving as expected, node (5,5) is probed for each stage of the pipeline. Moreover, the critical control signals between stages are probed as well as the signals that control the start of processing. Overall, it is expected that the behavior of these signals will match the simulation results.



Waveform - DEV:4 MyDevice4 (XC5VSX50T) UNIT:0 MyILA0 (ILA)

Bus/Signal	X	O
stage_1_output_loc_5_5	41C00000	41C00000
stage_2_output_loc_5_5	41500000	429B309C
stage_3_output_loc_5_5	406DB6DB	41100000
stage_4_output_loc_5_5	3FFFFFFF	3F800000
/send_start	0	0
/send_num	3	3
/crout_3d_0/seq_ctl_0/crout_demo_busy	1	1
stage_1_ready_for_data	1	1
stage_1_output_valid	0	0
stage_2_output_valid	0	0
stage_3_output_valid	0	0
stage_4_output_valid	1	1
sm_ok_to_send_next	0	0
advance_capture_RAM	0	0

X: 170 O: 214 Δ(X-O): -44

Figure 36: Results of Chipscope Session

As seen above, the results match the simulation results exactly! Each matrix result is output 44 cycles apart. The timing to start the access as well as the assertions of "ready for data" mirror those of the Modelsim simulation. The only restriction is that one node of each stage of the pipeline is probed in terms of matrix elements. Element (5,5) was chosen because it requires arithmetic operations in each "matrix stage" of the pipeline. The output of each matrix indicates that location (5,5) equals the following values:

Matrix 1 (5,5) Output =	0x3FFF_FFFF	=	1.9999998807907104
Matrix 2 (5,5) Output =	0x3F80_0000	=	1.0000000000000000
Matrix 3 (5,5) Output =	0x42A0_F2FE	=	80.47459411621094

The values observed at the (5,5) output are exactly as expected. The discrepancy between the simulation and "decimal" observation of location (5,5) is due to rounding. Aside from that, there are no problems with the output. Based on the design of the pipeline, if location (5,5) is known working, then all elements of the output matrices are assured to be working. This result coupled with the software verification insures the implementation of the 3D 5x5 Crout Elimination pipeline is a success.

4.6 IMPLEMENTATION AND VERIFICATION EPILOGUE

In this chapter, the implementation and verification of the Paul-Mickle 3D Crout Elimination pipeline is proven to be successful given the constraints of the target device. First, the primitives required were generated and an estimate of total logic consumption was calculated. Next, the design hierarchy was created and described. This consisted of a single unit to handle multiply and subtraction as well as the conditional instantiation of computational elements depending on the stage in the pipeline. Once the core pipeline was developed, support circuitry was added so that the logic would easily interface to a basic CPU for testing purposes. Simulations of this test structure were executed to insure proper functional operation. Finally, the complete design was synthesized and uploaded to the target FPGA. Both software and hardware tests were run to demonstrate that the design fully works as expected.

5.0 FUTURE WORK AND CONLUSIONS

A working implementation of the Paul-Mickle pipeline for a matrix of a non-trivial size has been presented. While some deviations from the original, “fully systolic” architecture were made mostly due to the amount of logic available on the target development PCB, this thesis demonstrates that the architecture is extensible using a modern FPGA device. However, more research is required to apply the design to more real world applications. This chapter details some topics that may arise from this thesis and examines some possible solutions.

5.1 FUTURE WORK

While this thesis proves the architecture is extensible, more work is required. This section details possible future work on the Paul-Mickle architecture. While the litany of suggestions is by no means complete, the following topics are some key issues that should be addressed in the future regarding this implementation.

5.1.1 Floating Point Core Limitations and Exception Handling

Real world operation of the Paul-Mickle logic must be robust. Any possible hazards encountered during operation must be handled so that a system can take corrective action and not enter indeterminate states. The presented implementation does not take exceptions into account such as divide by zero nor does it detect operations involving "not a number" (NaN). Also, the floating point cores do not handle denormalized values nor does it support any rounding operations specified by IEEE754 aside from truncation. Therefore, something has to be done to indicate exceptions.

In the case of handling NaN and divide-by-zero cases, flags available from the Xilinx floating point cores could be used to indicate when these exceptions occur [17]. Any time a flag is set in any floating point operation, this information can be sent to the host using the results of the Paul-Mickle pipeline. Ideally, some kind of pipeline that shifts any exception flag setting along with matrix results should exist in this pipeline. This would increase logic utilization, but not significantly. Xilinx datasheets indicate that only a few more LUTs and flip flops would be required per floating point core.

If denormalized values and other rounding operations are required, another floating point core is required. One possible solution is the use of IEEE 1076.3 synthesizable floating point libraries mentioned in Chapter 4. These libraries support denormalized numbers at the expense of higher logic requirements. Moreover, this library supports additional rounding operations that are required by the IEEE floating point specification. Unfortunately, at the time of writing this thesis, these libraries were not working with the Xilinx ISE design environment. In the future, these libraries might be available and are an ideal replacement for the Xilinx core in that the library is theoretically portable across all FPGA platforms should their tools choose to support it.

Finally, it is desirable to expand the single precision floating point values to double precision or beyond provided FPGA resources are available. Since the Crout function is highly iterative in an algorithmic sense, a risk exists where error due to rounding can accumulate through the pipeline stages. Xilinx provides the option to use double precision floating point values in their floating point cores. The tradeoff is both an increase in resource utilization as well as a decrease in maximum frequency. Anything beyond double precision shall require custom floating point cores.

5.1.2 Fully Systolic Operation

The implementation of the Paul-Mickle architecture in this thesis was not systolic at the system clock level due to the amount of logic required by the floating point processing elements. As a design tradeoff, the division function was created requiring a multiple cycle delay. Moreover, logic required by elements that merely delay the current contents of each element (i.e. FWD elements as described in Chapter 3) of a matrix stage were not fully implemented as the shift register structure was not required. The end result is a design that presents a new matrix solution at the end of each “matrix stage” cycle and not at each system clock cycle.

While the implementation described in this thesis is still very useful to demonstrate the feasibility of implementing the Paul-Mickle architecture in an FPGA, the first item that needs to be addressed is the modifications that are necessary to achieve systolic behavior at the system clock level. Systolic behavior can be achieved by creating a division function that is capable of accepting new data at each system clock cycle, and, at the same time, adding the shift register function that delays input elements in each matrix stage to compensate for arithmetic function pipeline latency.

Xilinx provides the option to create a division function with a “core available” latency of one cycle using the Coregen utility detailed in Chapter 4. However, this will add more logic utilization to the design. According to documentation from Xilinx regarding the floating point core used, a synthesized core with true systolic behavior requires five times the logic of a core that has 26 cycles of latency [17].

The designer has two options to implement the FWD element required in each matrix stage in the target FPGA. One method is to use a flip flop to store each bit of an IEEE754 floating point number. This method could grow into considerable flip flop utilization at each forwarding (FWD) element of the design. Suppose, in a systolic approach to the design, one requires 6 cycles for the divide function (i.e. the “del 1” delay as detailed in Chapter 3), 6 cycles for the multiply function (i.e. the “del 2 delay as detailed in Chapter 3), and 6 cycles for the subtraction. This demonstrates a need to delay each FWD element of a matrix stage by 18 cycles.

Overall, 32 flip flops are required to store a single IEEE754 value times a shift register depth of 18. This indicates that each FWD element shall require 576 flip flops! In the case of the 5x5, Paul-Mickle implementation, 60 forwarding functions are required across all matrix stages for a grand total of 34,560 flip flops. Moreover, since intermediate values are required to be forwarded to the next matrix stages, each element requires storage to balance out the latency required by each element. This shall increase flip flop requirements beyond the total of 34,560! In the end, a method to implement a more efficient shift register needs to be explored.

According to Xilinx documentation on the Virtex 5 devices, one half the LUTs available in the FPGA are available for use as an element called an SRL16 [13]. SRL16s use the memory elements of a LUT in a device as a shift register. As a result, the 6 cycles required for compensation of a mathematical function can be reduced from 192 flip flops down to 32 LUTs where each LUT can be used as the 6 cycle delay.

If the SRL16 approach is used to implement the delay chains required by the 5x5 example, only 5,760 LUTs are required. Three delay elements are needed to compensate for each mathematical function in a matrix stage (i.e. the 18 cycle depth). If 32 bit precision is used, each bit of the value is stored across 32 LUTs. The SRL16 provides up to 16 cycles of storage. Each delay only requires 6 cycles, so a delay at the bit level can be packed into a single LUT. Since 60 forwarding elements are used across all matrix stages, one finds that 5,760 LUTs are required. This amount of logic is significantly easier to manage in an FPGA. Given the results in Chapter 4, this delay could be packed into the existing design, but it is rather useless in that the divide function would have to be systolic for the delay function to be useful.

5.1.3 Resource Utilization

As indicated in Section 5.1.2, the Paul-Mickle architecture requires significant logic resources as matrix size increases. Referring to the mathematical representation of Crout elimination in Figure 1, the pseudocode in Figure 2, and the "unrolled loop" in Figure 11, one can see that the number of divide functions required increases linearly and the number of multiply and subtract functions increases exponentially for a square matrix of size M. Moreover, should one desire fully systolic operation as discussed in Section 5.1.2, the number of LUTs required for FWD (delay) elements grows exponentially as a square matrix of size M increases.

The following figures detail the resources required for a Paul-Mickle solution to solving Crout Elimination in an FPGA for a square matrix of size M. They were derived by examining the unrolled loop detailed in Figure 11. These equations assume one is using a Xilinx device with DSP48 slices present. Also, these equations assume one can implement a floating point divide function that provides a solution in 16 cycles. Sixteen cycle latency for floating point division is possible according to the floating point core datasheets [17]. This latency is required to insure a delay element is matched to the maximum delay possible using an SRL16 for delay purposes as discussed in Section 5.1.2.

$$\boxed{f_{divideareaFLOPs} = \sum_{i=1}^{M-1} i \times \left(fpdivide\ coresize_{FLOPs} \right)}$$

Figure 37: Function to compute flip flops required for the divide function

$$\boxed{f_{divideareaLUTs} = \sum_{i=1}^{M-1} i \times \left(fpdivide\ coresize_{LUTs} + 2 \times precision \right)}$$

Figure 38: Function to compute LUTs required for the divide function

The two functions detailed in Figure 37 and Figure 38 show the amount of logic required for the division functions required across all matrix stages. Figure 37 shows that the number of flip flops required grows linearly as do the number of LUTs required. If “full systolic” operation is desired, then one must account for the two delay elements required. Since these are implemented in Xilinx SRL16s, a single delay element consumes a single LUT. The variable “precision” indicates the width of a bit vector corresponding to a floating point number. In the case of this thesis, 32 bit precision was used. This may vary depending on future implementations of this pipeline.

$$\boxed{f_{mulsubareaFLOPs} = \sum_{i=1}^{M-1} i^2 \times \left(fpmult\ coresize_{FLOPs} + fpsubtract\ coresize_{FLOPs} \right)}$$

Figure 39: Function to compute flip flops required for the multiply-then-subtract function

$$\boxed{f_{mulsubareaLUTs} = \sum_{i=1}^{M-1} i^2 \times \left(fpmult\ coresize_{LUTs} + fpsubtract\ coresize_{LUTs} + 4 \times precision \right)}$$

Figure 40: Function to compute LUTs required for the multiply-then-subtract function

$$\boxed{f_{multDSPs} = \sum_{i=1}^{M-1} i^2 \times (DSPslice)}$$

Figure 41: Function to compute DSP48 slices required for the multiply function

The three functions detailed in Figure 39, Figure 40, and Figure 41 show the amount of logic required for the multiply and subtraction functions required across all matrix stages. These figures show that the number of required flip flops, LUTs, and DSP48 elements grows exponentially. If "full systolic" operation is desired, then one must account for the two delay elements required in both the multiply and subtraction function. Since these are implemented in Xilinx SRL16s, a single delay element consumes a single LUT and each arithmetic operation requires two delays for a total of 4 delay elements. Like the division resource utilization functions, the variable "precision" indicates the width of a bit vector corresponding to a floating point number. Also, in this thesis, the floating point multiplication function used DSP logic to facilitate fitting of the pipeline into the target FPGA. This may not be necessary in a future implementation of this pipeline provided sufficient resources are available.

Finally, resources used for the "FWD" elements must be counted. Like the multiplication and subtraction functions, these requirements also grow exponentially in size as the square matrix of size M increases. In the case of a "fully systolic" implementation, the function detailed in Figure 42 is used. A total of three SRL16 delay elements are required to compensate for the division, multiplication, and subtraction functions.

$$\boxed{f_{fwdareaLUTs} = \sum_{i=1}^{M-1} \left(M^2 - \left(i \times (i+1) \right) \right) \times \left(3 \times precision \right)}$$

Figure 42: Function to compute LUTs required for the FWD function

Any element in a matrix stage that does not contain arithmetic operators is required to store the results of the previous stage for forwarding to the next stage. In the case of this thesis, the need for the delay elements was not required as flip flops were used to store intermediate results. Recall that the division function was implemented with the restriction that new operands could not be pushed into the function until the previous function is complete. While this eliminates the need for FWD delay elements as described in this section, it does require that storage is used at each intermediate node so that intermediate results are saved and forwarded to their corresponding stages in the pipeline. In the case of this thesis, banks of flip flops were used at each node to store intermediate results.

Figure 43 details the number of flip flops required should one use this approach to implementing a non-systolic version of the Paul-Mickle pipeline.

$$\boxed{f_{fwdareaFLOPs} = \sum_{i=1}^{M-1} \left(M^2 - \left(i \times (i+1) \right) \right) \times precision}$$

Figure 43: Total number of flip flops required for "non-systolic" FWD functions

Overall, the Paul-Mickle architecture requires substantial logic as the matrix size increases. Suppose one were to create a system based on this architecture to solve 20x20 matrices. Figure 42 shows that the final matrix stage alone would require over 38,000 LUTs just to implement the required FWD functions should one desire to implement a systolic pipeline with 32-bit precision! This logic alone would not fit in the Virtex-5 FPGA used in this thesis. The largest Xilinx device available at the time of writing this thesis, the Virtex-5 LX330, would be 18% utilized just by this requirement alone [13]! Therefore, some kind of solution is required so that the Paul-Mickle pipeline remains extensible using FPGA technology. A solution to be considered is partitioning the design across multiple FPGAs.

5.1.4 Design Partitioning

While a 5x5 matrix example is by no means trivial to implement, the three dimensional Crout Elimination pipeline must support larger matrix sizes. Since the design detailed in this thesis indicates that one requires roughly 25,000 logic elements to implement the non-systolic 5x5 solution, any 5x5 systolic function or matrix size over 5x5 shall require multiple FPGAs to handle the demand for more logic. As implied in Chapter 3 and detailed in Section 5.1.3, the amount of logic needed for Crout Elimination on a square matrix grows exponentially as matrix size increases.

To remedy this logic demand, the design can be partitioned across multiple FPGAs. In turn, these multiple FPGAs can be placed on a single PCB. One problem, however, is communication overhead due to data transfers from one FPGA to another. After referring to the Xilinx Virtex 5 data sheets, one can see that modern FPGAs are well equipped to handle these tasks [13].

The number of pins available on modern FPGAs is rather impressive. In FF1738 package, a Xilinx FPGA provides 960 I/O pins available for general purpose use. If a designer requires a wide, parallel bus, a vast number of these pins can be used for that purpose as all I/O have features such as DDR flip flops and output skewing for source synchronous applications. Each I/O pin is capable of a single ended operation at 800Mbps.

Each I/O pair is capable of differential operation should clock frequency requirements dictate a higher speed transfer from one FPGA to another is required. The benefit of LVDS pairs is increased performance at the expense of two I/O being used for each bit. These LVDS pairs can be ganged in parallel for very high speed serial busses operating in parallel. Each LVDS pair is capable of running at a 1.25Gbps rate. Should higher serial rates be required, Xilinx provides up to 24 “Rocket I/O” pins. Each Rocket I/O pin is capable of 3.75Gbps transfers. Overall, chip interconnect in a partitioned design should provide substantial bandwidth.

Another problem with design partitioning is mapping partitioned blocks to the target FPGAs. The VHDL provided in Appendix A assumes targeting the design to a single FPGA. Additional work is required to break the logic down into functional units that can map to available resources on the multiple FPGAs in a partitioned design. Moreover, additional logic is required to map intermediate results to the communication protocol used for chip-to-chip communication.

One possible solution to the partitioning problem is to modify the code taking these partition sizes into account. Referring to the logic provided in Appendix A, the use of VHDL generics demonstrated that parameters could be used to control the size of the pipeline by simply changing a few parameters in the code. Since it would be desirable to have this partitioning process automated, one could use the parameterized nature of the design to create a program that would automatically generate the partitioned units needed to create a larger matrix pipeline. This program could determine the amount of logic used in each matrix stage and determine the best way to pack the design into available resources. Subsequently, the program would be responsible for assigning proper parameters to the logic in the pipeline. While some changes are required to the overall structure of the VHDL presented in this thesis, the overall idea could be implemented using this logic as a basis for more work on partitioning.

5.1.5 FPGA Based Distributed Computing Clusters

For solving "very large" matrix sizes, partitioning as described in Section 5.1.4 might not be enough. Many factors such as PCB real estate, power consumption, and signal integrity are all concerns when designing relatively large PCBs. In these cases, it might be desirable to take partitioning one step farther and divide up the tasks among multiple FPGAs on multiple PCBs.

Such technology exists today and would be advantageous to use for Paul-Mickle pipelines of very large scale.

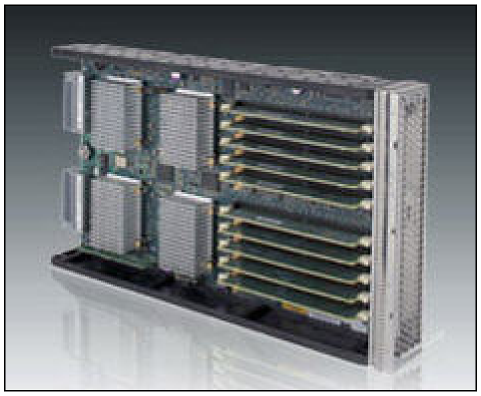


Figure 44: The SGI Reconfigurable Application Specific Computing (RASC) RC100

To address the need for general purpose platforms with multiple FPGAs, vendors are providing solutions that address the need for more logic on a general purpose platform. One such vendor, SGI, provides the Reconfigurable Application Specific Computing (RASC) RC100 blade to fill these needs [18]. The RC100 contains dual Virtex 4 LX200 FPGAs where each FPGA contains 200,488 LUTs and 200,488 flip flops. The FPGAs are interconnected using a proprietary interconnect capable of sustaining speeds up to 6.4GB per second.

The RC100 blades operate in an SGI server. Should a user require additional logic, up to 64 blades may be added providing up to 128 Virtex 4 LX200 FPGAs. This solution provides the user with 25.6M LUTs and flip-flops! A shared memory is available should it be needed to store intermediate results in the event the algorithm is modified in the future to take advantage of such resources.

Despite the vast amount of logic available on this platform, a threshold will be crossed where there will not be enough resources available to implement a solution for a large matrix size on an SGI server. One possible solution to this problem is using networking appliances like the SGI server equipped with a very high speed communication interfaces like 10 Gigabit Ethernet. Xilinx provides a 10Gb Ethernet core for use in their Virtex 4 and 5 FPGAs [20]. This core, coupled with a physical layer device, could allow high bandwidth communication of intermediate results between FPGA servers.

Eventually, another threshold will be crossed where the number of servers required to solve a large sized matrix will become unfeasible due to reasons such as cost, power, and overall area required to install a server farm. It may be desirable to use the concepts presented in Section 2.3 to work around this issue. As FPGA based servers grow in use across the world, it may be possible to apply some of the research in [4] and create a non-dedicated cluster of FPGA based servers to solve these extremely large matrices using the Paul-Mickle architecture.

In the future, it might be possible to use some kind of basic, time sharing algorithm to allocate resources to users requesting access to massive amounts of FPGA resources. Once a user is granted access, the pipeline can be partitioned according to the available resources in the network and the resulting FPGA configuration files can be sent to the target devices. After computations are complete or the user’s time slice expires, the system is then free to accept configuration data from another user. While this idea requires substantial research in design partitioning and system architecture, the concept is feasible using technology available in 2008.

5.2 CONCLUSION

Overall, this thesis demonstrates that it is feasible for the Paul-Mickle pipeline to be implemented in modern FPGAs. While earlier solutions for high speed LU factorization required the use of multiple CPUs, scheduling overhead, and communication time typically encountered in parallel processing solutions, the use of custom logic allows the possibility of implementing an optimal solution to the Crout Elimination problem should latency be the primary concern of the required solution. Custom logic, through the use of FPGAs, alleviates much of the overhead required with parallel processing solutions on CPUs.

The solution presented in this thesis is complete when considering the constraints of the hardware used to implement the design, but more work is required should an end user require a systolic solution or solutions to solve larger matrix sizes. However, it is hoped that the work in this thesis provides an excellent starting point for future work in the implementation of the Paul-Mickle Three Dimensional Pipeline.

APPENDIX A

VHDL SOURCE FOR CROUT PIPELINE

The VHDL for the Crout pipeline is presented. Only the code for the 3D pipeline is presented. The floating point cores are assumed to be a fixed netlist since they are provided by Xilinx and are not available in source form.

A.1 FLOATING POINT MULTIPLY-SUBTRACT FUNCTION

The following VHDL source code details the creation of the FP_MULT_SUB function as described in Chapter 3.

```vhdl
----------------------------------------------------------------------------------
--
-- Title    : Floating Point mutliply then subtract function.
--
-- Author   : Ed Henciak
--
-- Function : For Crout elimination, a multiply followed by subtract function
--            is required for nodes in the "matrix stage" being processed.  These
--            two functions are combined in this component to make instantiation
--            easier at higher levels.
--
----------------------------------------------------------------------------------

LIBRARY IEEE;
  USE IEEE.STD_LOGIC_1164.ALL;
  USE IEEE.std_logic_arith.ALL;
  USE IEEE.std_logic_unsigned.ALL;
```

```
ENTITY fp_mult_subt IS
PORT (

    -- Main Clock & reset
    clk         : IN    STD_LOGIC;
    reset       : IN    STD_LOGIC;

    -- Indicates that new data is present and valid.
    op_nd       : IN    STD_LOGIC;

    -- Input operands...
    quot_in     : IN    STD_LOGIC_VECTOR(31 DOWNTO 0); -- Input quotient from divider.
    mlt_in      : IN    STD_LOGIC_VECTOR(31 DOWNTO 0); -- Value to multiply with quot.
    sbt_in      : IN    STD_LOGIC_VECTOR(31 DOWNTO 0); -- Value from which we subtract
                                                       -- the product of the quotient
and
                                                       -- mlt_in value.

    -- Result of above computations.
    result_rdy  : OUT   STD_LOGIC;
    result      : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)

);
END ENTITY;

ARCHITECTURE rtl OF fp_mult_subt IS

    -- IEEE 754 Multiply component generated in Xilinx Coregen
    -- using ISE9.2SP2
    COMPONENT fp_multiply
    PORT (
        a             : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        b             : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        operation_nd  : IN STD_LOGIC;
        operation_rfd : OUT STD_LOGIC;
        clk           : IN STD_LOGIC;
        sclr          : IN STD_LOGIC;
        result        : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        rdy           : OUT STD_LOGIC
    );
    END COMPONENT;

    -- IEEE 754 Subtract component generated in Xilinx Coregen
    -- using ISE9.2SP2
    COMPONENT fp_subtract
    PORT (
        a             : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        b             : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        operation_nd  : IN STD_LOGIC;
        operation_rfd : OUT STD_LOGIC;
        clk           : IN STD_LOGIC;
        sclr          : IN STD_LOGIC;
        result        : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        rdy           : OUT STD_LOGIC
    );
    END COMPONENT;

    -- Signal declarations
    SIGNAL product     : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL product_rdy : STD_LOGIC;

BEGIN
```

```
    -- Instantiate multiply function.
    fp_mult_0 : fp_multiply
    PORT MAP(
         a              => quot_in,
         b              => mlt_in,
         operation_nd   => op_nd,
         operation_rfd => OPEN, -- Multiplier is always ready
         clk            => clk,
         sclr           => reset,
         result         => product,
         rdy            => product_rdy
    );

    -- Instantiate subtractor.
    fp_subt_0 : fp_subtract
    PORT MAP(
         a              => sbt_in,
         b              => product,
         operation_nd   => product_rdy,
         operation_rfd => OPEN, -- Subtractor is always ready
         clk            => clk,
         sclr           => reset,
         result         => result, -- Difference
         rdy            => result_rdy
    );

END ARCHITECTURE rtl;
```

A.2 PIPELINE MATRIX STAGE VHDL

The following VHDL source code details the creation of a single stage of the Crout pipeline. Please note that the library “float_convert” is a library created for this thesis based on the proposed IEEE1076.3 library. This library is used to convert VHDL “real” types to IEEE754 floating point numbers in VHDL “std_logic_vector” notation. It is not required to synthesize the actual design. It is merely used for debugging purposes. The library “textio_utils_pkg” merely provides a means to print VHDL simulation results to the console. It is not required to synthesize the design.

```
--------------------------------------------------------------------------------
--
-- Title    : 3D Crout Pipeline Stage
--
-- Author   : Ed Henciak
--
-- Function : Creates a single stage of the Mickle-Paul 3D
--            Crout pipeline.
--
--------------------------------------------------------------------------------

LIBRARY IEEE;
    USE IEEE.STD_LOGIC_1164.ALL;
    USE IEEE.std_logic_arith.ALL;
    USE IEEE.std_logic_unsigned.ALL;

-- synthesis translate_off
LIBRARY WORK;
    USE WORK.textio_utils_pkg.ALL;
    USE WORK.float_convert_pkg.ALL;
-- synthesis translate_on

ENTITY crout_pipe_stage IS
GENERIC (

    matrix_m     : INTEGER := 5;
    floatvec_len : INTEGER := 32;
    stage        : INTEGER := 1

);
PORT (

    -- Main Clock & reset
    clk         : IN    STD_LOGIC;
    reset       : IN    STD_LOGIC;

    -- Input operands...
    crout_rfd    : OUT  STD_LOGIC; -- Ready for data...
    crout_din    : IN   STD_LOGIC_VECTOR(((floatvec_len*(matrix_m**2))-1) DOWNTO 0); -
- Input matrix
    crout_ivalid : IN   STD_LOGIC; -- Matrix is valid

    -- Results
    crout_dout   : OUT  STD_LOGIC_VECTOR(((floatvec_len*(matrix_m**2))-1) DOWNTO 0); -
- Output matrix
    crout_ovalid : OUT  STD_LOGIC  -- output is valid.

);
END ENTITY;

ARCHITECTURE rtl OF crout_pipe_stage IS

    -- Constant which represents the number of bits required to represent the
    -- entire matrix at this stage...
    CONSTANT bits_per_row : INTEGER := matrix_m * floatvec_len;

    -- Processed matrix type (makes assignments easier)
    TYPE pmtx_t IS ARRAY (1 TO matrix_m, 1 TO matrix_m) OF
STD_LOGIC_VECTOR(floatvec_len-1 DOWNTO 0);
    SIGNAL mtx_in   : pmtx_t;  -- Input matrix for this stage...
    SIGNAL rslt_m_i : pmtx_t;  -- Result matrix for this stage (wires)...
    SIGNAL rslt_m   : pmtx_t;  -- Result matrix for this stage (flops)...
    -- "Quotient" type is simply an array of std_logic_vectors.
    TYPE quotient_t IS ARRAY (1 TO matrix_m) OF STD_LOGIC_VECTOR(floatvec_len-1 DOWNTO
0);
    SIGNAL quotient : quotient_t; -- Array of quotients for this stage.

    -- "Quotient is ready" vector...
    SIGNAL quotient_rdy : STD_LOGIC_VECTOR(1 TO matrix_m);

    -- The "result ready" flag is actually a 2D array of bits...only one of these
    -- will be used by the logic...the rest will be optimized away.
    TYPE rslt_rdy_t IS ARRAY (1 TO matrix_m, 1 TO matrix_m) OF STD_LOGIC;
    SIGNAL rslt_rdy : rslt_rdy_t;

    -- Signal indicating that mutliply & subtract unit is done...
    SIGNAL mult_sub_done : STD_LOGIC;

    -- Output of this stage is valid
    SIGNAL ovalid : STD_LOGIC;

    SIGNAL ready_for_data : STD_LOGIC := '1';

BEGIN

    -- First, use a generate statement to organize the matrix.  All this is doing
    -- is making the "wires" easier to read...it consumes no logic...
    create_input_mtx_i_0 : FOR i IN 1 TO matrix_m
    GENERATE

        create_input_mtx_j_0 : FOR j IN 1 TO matrix_m
         GENERATE
            mtx_in(i,j) <= crout_din( ( ((j*floatvec_len)-1) + ((i-1)*bits_per_row) )
DOWNTO
                                        ( (j-1)*floatvec_len + ((i-1)*bits_per_row) )
);
         END GENERATE;

    END GENERATE;

    -- Next we have to instantiate either a "keep", a "divide" or a "mult/subtract"
    -- component...these are instantiated based on stage...
    --create_proc_i_0 : for i in STAGE to MATRIX_M
    create_proc_i_0 : FOR i IN 1 TO matrix_m
    GENERATE

        create_proc_j_0 : FOR j IN 1 TO matrix_m
         GENERATE

             -- Instantiate a divider if I = current stage
             -- and j > current stage...
             check_if_divide : IF ((i = stage) AND (j > stage))
             GENERATE

                 -- Instantiate divide function.
                 fp_div_0 : ENTITY WORK.fp_divide
                 PORT MAP(
                     a             => mtx_in(i,j),
                     b             => mtx_in(stage,stage),
                     operation_nd  => crout_ivalid,
                     operation_rfd => OPEN,
                     clk           => clk,
                     sclr          => reset,
                     result        => quotient(j),
                     rdy           => quotient_rdy(j)
            );

             rslt_m_i(i,j) <= quotient(j);

        END GENERATE;

         -- Here we prevent latches in the event quotient(x) is not
         -- used in this stage...the synthesis tool should optimize
        -- this element away since it is unused.
         clear_quotient : IF ((i=stage) AND (j <= stage))
         GENERATE

             quotient(j)     <= (OTHERS => '0');
              quotient_rdy(j) <= '0';

         END GENERATE;

         -- Instantiate a mult/subtract if we're "under the
         -- dividers"....
         check_if_multsub : IF ((i>stage) AND (j>stage))
         GENERATE

             fp_mult_subt_0 : ENTITY WORK.fp_mult_subt
             PORT MAP(

                 -- Main Clock & reset
                 clk         => clk,
                 reset       => reset,

                 -- Indicates that new data is present and valid.
                 op_nd       => quotient_rdy(j),

                 -- Input operands...
                 quot_in     => quotient(j),
                 mlt_in      => mtx_in(i,stage),
                 sbt_in      => mtx_in(i,j),

                 -- Result of above computations.
                 result_rdy  => rslt_rdy(i,j),
                 result      => rslt_m_i(i,j)

             );

        END GENERATE;

         -- Here we simply pass results if no operation is to be performed.
         --check_if_pass : if ( (not((i = STAGE) and (j > STAGE))) and
         --                     (not((i>STAGE) and (j>STAGE))) )
         check_if_pass : IF ((i < stage) OR (j <= stage))
         GENERATE

                 rslt_rdy(i,j) <= '0';
                 rslt_m_i(i,j) <= mtx_in(i,j);

         END GENERATE;

    END GENERATE;

END GENERATE;

-- Always use the "output valid" signal from a mult/sub unit
-- that will be used regardless of stage (i.e. "lower right"
-- unit will always be used...).
```

```
mult_sub_done <= rslt_rdy(matrix_m, matrix_m);

-- Check above...
PROCESS(clk, reset)
BEGIN

    IF (reset = '1') THEN

        ovalid         <= '0';
        ready_for_data <= '1';

        FOR i IN 1 TO matrix_m LOOP
            FOR j IN 1 TO matrix_m LOOP
                rslt_m(i,j) <= (OTHERS => '0');
            END LOOP;
        END LOOP;

    ELSIF (clk'EVENT AND clk = '1') THEN

        -- Defaults...
        ovalid <= '0';

        -- In terms of "real" logic, all values that require no
        -- operation are registered here.  There is also some code
        -- to dispay results for debugging purposes...
        IF (crout_ivalid = '1') THEN

            ready_for_data <= '0';

            -- synthesis translate_off
            banner(" Input of stage #" & image(stage));
            FOR r IN 1 TO 5 LOOP

               printf(image(ieee754slv_to_real(mtx_in(r,1))) & "   " &
                      image(ieee754slv_to_real(mtx_in(r,2))) & "   " &
                      image(ieee754slv_to_real(mtx_in(r,3))) & "   " &
                      image(ieee754slv_to_real(mtx_in(r,4))) & "   " &
                      image(ieee754slv_to_real(mtx_in(r,5))));

            END LOOP;
            -- synthesis translate_on

            -- Register "pass" values...
            FOR i IN 1 TO matrix_m LOOP

                FOR j IN 1 TO matrix_m LOOP

                   IF ( (NOT((i = stage) AND (j > stage))) AND
                        (NOT((i>stage) AND (j>stage))) ) THEN

                       rslt_m(i,j) <= rslt_m_i(i,j);

                   END IF;

                END LOOP;

            END LOOP;

        END IF;

        -- When the quotient is ready, we need to register the
        -- quotients immediately as they will be invalid on the
        -- next cycle.
        IF (quotient_rdy(matrix_m) = '1') THEN

             -- Register quotients....
            FOR i IN (stage+1) TO matrix_m LOOP
                 rslt_m(stage, i) <= rslt_m_i(stage,i);
             END LOOP;

        END IF;

        -- When the multiply/subtract units are done, the current
       -- stage is complete...we can forward it on to the next stage...
        IF (mult_sub_done = '1') THEN

             -- Set ready for data flag
             ready_for_data <= '1';

             -- Output is valid...
             ovalid <= '1';

             -- Register the results of the mult+subtract units.
             FOR i IN 1 TO matrix_m LOOP

                 FOR j IN 1 TO matrix_m LOOP

                     IF ((i>stage) AND (j>stage)) THEN
                         rslt_m(i,j) <= rslt_m_i(i,j);
                     END IF;

                 END LOOP;

             END LOOP;

        END IF;

    END IF;

END PROCESS;

-- synthesis translate_off
-- Simple process to display intermediate results...
PROCESS(clk)
BEGIN

    IF (clk'EVENT AND clk = '1') THEN

        IF (ovalid = '1') THEN

            banner(" Output of stage #" & image(stage));

            FOR r IN 1 TO 5 LOOP

                printf(image(ieee754slv_to_real(rslt_m(r,1))) & "    " &
                       image(ieee754slv_to_real(rslt_m(r,2))) & "    " &
                       image(ieee754slv_to_real(rslt_m(r,3))) & "    " &
                       image(ieee754slv_to_real(rslt_m(r,4))) & "    " &
                       image(ieee754slv_to_real(rslt_m(r,5))));

            END LOOP;

        END IF;

    END IF;
    END PROCESS;
    -- synthesis translate_on

    -- Concurrent signal assignments

    -- Output of the divide function indicates when we can accept the next
    -- matrix.  Always use a divider that will be "used" regardless of the
    -- current stage!
    crout_rfd <= ready_for_data;

    -- Output valid signal...
    crout_ovalid <= ovalid;

    -- Create output vector...
    create_output_vec_i_0 : FOR i IN 1 TO matrix_m
    GENERATE

        create_output_vec_j_0 : FOR j IN 1 TO matrix_m
         GENERATE
            crout_dout( ( ((j*floatvec_len)-1) + ((i-1)*bits_per_row) ) DOWNTO
                                   ( (j-1)*floatvec_len + ((i-1)*bits_per_row) ) ) <=
rslt_m(i,j);
         END GENERATE;

    END GENERATE;

END ARCHITECTURE rtl;
```

A.3 CROUT 5X5 PIPELINE VHDL

The following VHDL source code details the creation of a single stage of the Crout pipeline. Please note that the library “float_convert” is a library created for this thesis based on the proposed IEEE1076.3 library. This library is used to convert VHDL “real” types to IEEE754 floating point numbers in VHDL “std_logic_vector” notation. It is not required to synthesize the actual design. It is merely used for debugging purposes. The library “textio_utils_pkg” merely provides a means to print VHDL simulation results to the console. It is not required to synthesize the design.

```
-------------------------------------------------------------------------------
--
-- Title    : 5x5 Crout Pipeline
--
-- Author   : Ed Henciak
--
-- Function : Instantiates the components for the 5x5 Crout demonstration.
--
-------------------------------------------------------------------------------

LIBRARY IEEE;
    USE IEEE.STD_LOGIC_1164.ALL;
    USE IEEE.std_logic_arith.ALL;
    USE IEEE.std_logic_unsigned.ALL;

-- synthesis translate_off
LIBRARY WORK;
    USE WORK.textio_utils_pkg.ALL;
    USE WORK.float_convert_pkg.ALL;
-- synthesis translate_on

ENTITY crout_5_5_pipeline IS
PORT (

    -- Main Clock & reset
    clk         : IN    STD_LOGIC;
    reset       : IN    STD_LOGIC;

    -- Input operands...
    crout_rfd    : OUT  STD_LOGIC; -- Ready for data...
    crout_din    : IN   STD_LOGIC_VECTOR(799 DOWNTO 0); -- Input matrix
    crout_ivalid : IN   STD_LOGIC; -- Matrix is valid

    -- Results
    crout_dout   : OUT  STD_LOGIC_VECTOR(799 DOWNTO 0); -- Output matrix
    crout_ovalid : OUT  STD_LOGIC  -- output is valid.

);
END ENTITY;

ARCHITECTURE rtl OF crout_5_5_pipeline IS

    CONSTANT matrix_m : INTEGER := 5;

    SIGNAL crout_rfd_i  : STD_LOGIC_VECTOR(0 TO 4);
    SIGNAL crout_dvalid : STD_LOGIC_VECTOR(0 TO 4);

    TYPE crout_vec_t IS ARRAY (0 TO 4) OF STD_LOGIC_VECTOR(799 DOWNTO 0);
    SIGNAL crout_data_i  : crout_vec_t;

BEGIN

    -- Concurrent assignments to drive data into pipeline...
    crout_dvalid(0) <= crout_ivalid;
    crout_data_i(0) <= crout_din;

    -- Instantiate a 4 stage pipeline to solve the 5x5 case.
    crout_5_5_pipe : FOR k IN 1 TO matrix_m-1
    GENERATE

        crout_stage : ENTITY WORK.crout_pipe_stage
        GENERIC MAP(
            matrix_m     => matrix_m, -- This is a 5x5 matrix
            floatvec_len => 32, -- Precision is 32 bits.
            stage        => k   -- We're on stage "K"

        )
        PORT MAP(

            -- Main Clock & reset
            clk         => clk,
            reset       => reset,

            -- Input operands...
            crout_rfd    => crout_rfd_i(k-1),
            crout_din    => crout_data_i(k-1),
            crout_ivalid => crout_dvalid(k-1),

            -- Results
            crout_dout   => crout_data_i(k),
            crout_ovalid => crout_dvalid(k)

        );

    END GENERATE;

    -- Concurrent assignments to drive the outputs...
    crout_rfd    <= crout_rfd_i(0); -- 1st stage RFD indicates when it is clear to
send next mtx.
    crout_dout   <= crout_data_i(4); -- 4th stage is a completed matrix output.
    crout_ovalid <= crout_dvalid(4); -- Use qualifier from stage 4.

END ARCHITECTURE rtl;
```

BIBLIOGRAPHY

[1] Paul, J.M.; Mickle, M.H., “Three-dimensional computational pipelining with minimal latency and maximum throughput for L-U factorization”, Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on] Volume 45, Issue 11, Nov. 1998 Page(s):1465 – 1475, Digital Object Identifier 10.1109/82.735358

[2] Chen, C.-C.; Hu, Y.-H., “Parallel LU factorization for circuit simulation on an MIMD computer”, Computer Design: VLSI in Computers and Processors, 1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on 3-5 Oct. 1988 Page(s):129 – 132, Digital Object Identifier 10.1109/ICCD.1988.25676

[3] Kratzer, S.G., “Massively parallel sparse LU factorization”, Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the 19-21 Oct. 1992 Page(s):136 – 140, Digital Object Identifier 10.1109/FMPC.1992.234896

[4] Endo, T.; Kaneda, K.; Taura, K.; Yonezawa, A., “High performance LU factorization for non-dedicated clusters”, Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on 19-22 April 2004 Page(s):678 – 685, Digital Object Identifier 10.1109/CCGrid.2004.1336698

[5] Wikipedia Article “Grid Computing”, http://en.wikipedia.org/wiki/Grid_computing

[6] Computer Economics, “Internet and Broadband Growth Accelerates Worldwide”, March 2007, http://www.computereconomics.com/article.cfm?id=1206

[7] Stanford University Folding@Home Project FAQ, http://folding.stanford.edu/English/FAQ-main_Folding@home

[8] University of California at Berkeley SETI@Home Project Main Page, http://seticlassic.ssl.berkeley.edu/about_seti/about_seti_at_home_1.html

[9] Xiaofang Wang; Ziavras, S.G., “Parallel direct solution of linear equations on FPGA-based machines”, Parallel and Distributed Processing Symposium, 2003. Proceedings. International, 22-26 April 2003, Page(s) 8pp. Digital Object Identifier 10.1109/IPDPS.2003.1213224

[10] Xiaofang Wang; Ziavras, S.G., "Performance optimization of an FPGA-based configurable multiprocessor for matrix operations Field-Programmable Technology (FPT)", 2003. Proceedings. 2003 IEEE International Conference on 15-17 Dec. 2003 Page(s):303 – 306, Digital Object Identifier 10.1109/FPT.2003.1275763

[11] Wang, X.; Ziavras, S.G., "A configurable multiprocessor and dynamic load balancing for parallel LU factorization", Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, 26-30 April 2004 Page(s):234, Digital Object Identifier 10.1109/IPDPS.2004.1303282

[12] Govindu, G.; Choi, S.; Prasanna, V.; Daga, V.; Gangadharpalli, S.; Sridhar, V., "A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs", Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, 26-30 April 2004 Page(s):149, Digital Object Identifier 10.1109/IPDPS.2004.1303134

[13] Xilinx Virtex 5 Datasheets, http://www.xilinx.com

[14] IEEE1076.3 Working Group for a Floating Point Package for VHDL, http://www.eda.org/fphdl/

[15] IEEE1076.3 Package Download and Status Site, http://www.vhdl.org/vhdl-200x/vhdl-200x-ft/packages/files.html

[16] Xilinx Coregen IP Generator Documentation, http://www.xilinx.com

[17] Xilinx Floating-Point Operator v3.0 documentation, http://www.xilinx.com

[18] SGI RASC RC100 Blade, http://www.sgi.com/pdfs/3920.pdf

[19] Xilinx Virtex 4 Datasheets, http://www.xilinx.com

[20] Xilinx 10 Gigabit Ethernet Media Access Controller Datasheet, http://www.xilinx.com