Accelerating Financial Applications through Specialized Hardware – FPGA

By:

Tri Dang

John Rothermel

Date: December 14, 2007

EXECUTIVE SUMMARY

The need for computing power is continually increasing, especially in investment banks. These institutions use large clusters with many processor cores to approximate the future value of investments. Better computer performance correlates to increased accuracy and speed of calculations, which leads an edge on the investment bank's competitors and increased profit [1].

In the past, as demand for computing power increased, industry was able to keep pace; however, as processor cores increased in speed, more power was required. Higher power demands meant worsening power costs and a need for better heat dissipation. As an alternative to adding more processor cores to a system, Field Programmable Gate Array (FPGA) technology has been proposed as a possible alternative. FPGA advocates advertise fifty times speed improvements on their machines using a traditional CPU and FPGA coprocessor for Black-Scholes calculations, while using less power [2]. This project examined FPGA technology objectively in terms of speed and accuracy improvements for floating point mathematical operations.

The first objective was to evaluate current methods of floating point math in traditional CPUs and FPGAs. This base of knowledge helped us determine possible areas where FPGAs could augment traditional CPU performance. Following this research, we then developed benchmark tests for simple arithmetic on a CPU and with an FPGA coprocessor. Finally, we built on these findings by extending the benchmark tests to transcendental functions like exponentiation, logarithm, and trigonometry.

After researching current methods of processors to compute floating point math, we determined the optimal performance capabilities for the AMD Opteron (10h family) and Intel Core microarchitectures. Comparing both companies' advertized latencies and throughput for

varying SSE instructions yielded a few tables. The following table illustrates latency in clock cycles.

Architecture/Instruction	Add	Multiply	Divide	SqRt
AMD Opteron (10h family)	4	4	16	27
Intel Core	3	5	32	58

Table 1: Latency Comparison of Advertized Required Cycles for Opteron (10h Family) and Core

We then used a clock speed of 2.5GHz to compare theoretical execution times (in nanoseconds) of both AMD and Intel microarchitectures. Many AMD Opteron microprocessors operate between 2.4 and 2.6GHz. 2.5GHz was chosen as a reference point for comparison between the two microarchitectures.

Architecture/Instruction	Add	Multiply	Divide	SqRt
AMD Opteron (10h family)	1.6	1.6	6.4	10.8
Intel Core	1.2	2.0	12.8	23.2

Table 2: Latency Comparison of Advertized Required Time (ns) for Opteron (10h Family) and Core (based on 2.5GHz core)

The final table displays theoretical throughput of different floating point operations in clock cycles. Throughput measures the rate of execution in clock cycles. For example, according to the following table one addition can be executed per clock cycle, due to pipelining and multiple execution units.

Architecture/Instruction	Add	Multiply	Divide	SqRt
AMD Opteron (10h family)	1	1	17	24
Intel Core	1	1	31	57

Table 3: Throughput Comparison (all double precision SIMD)

Both microarchitectures theoretically are able to perform an addition or multiplication every clock cycle, which for a 2.5GHz processor is every 0.4 nanoseconds. An FPGA on the other hand typically runs at about 100MHz. This comparison, however, does not take into account caching and other real world issues.

Results

During the benchmark testing process, we first tested the speed and accuracy of an AMD Opteron 285, running at 2.6GHz. We started with basic arithmetic and then tested transcendental functions. These results include some overhead such as incrementing a pointer to an array of values and jump commands within a loop. The results of these benchmark tests are provided in the following table:

Average time per operation for operations and options (ns/op)								
Options: fastmath (f), loop-unrolling (l), sse (s), x87 (x)								
	s	x	sf	xf	sl	xl	slf	xlf
Add	3.24	3.22	3.22	3.22	3.22	3.22	3.22	3.22
Sub	3.61	3.58	3.60	3.62	3.59	3.59	3.60	3.57
Mul	3.62	3.62	3.62	3.62	3.62	3.62	3.62	3.62
Div	3.61	3.61	3.62	3.62	3.61	3.62	3.62	3.62
Sqrt	10.89	14.78	3.19	3.18	10.88	15.12	3.18	3.19
Sin	3.25	3.24	3.23	3.24	3.24	3.25	3.23	3.22
Cos	3.62	3.62	3.62	3.62	3.61	3.61	3.61	3.62
Tan	3.62	3.62	3.62	3.62	3.62	3.62	3.62	3.62
ExpE	271.47	256.72	3.61	3.61	282.55	258.65	108.04	108.29
LogE	32.48	32.97	3.92	35.72	32.62	32.84	35.67	35.66
Log10	31.95	32.75	3.38	32.43	32.45	32.11	32.52	32.51

Table 4: CPU Benchmark Results

The CPU was able to perform basic arithmetic, including division in three to four nanoseconds, while exponential functions tended to take a longer time. The fast-math compiler option was used effectively to speed up transcendental calculations, but did have some accuracy loss. When calculating trigonometric functions using the fast-math option, frequent not-a-number (NaN) results were recorded, while only an occasional error was recorded for other transcendental functions. All other comparisons yielded no recorded accuracy loss in a sample of ten thousand numbers. We concluded that a CPU is capable of high speed applications, but the fast-math option should only be used for non-trigonometric functions.

Using Impulse C, we were able to program the provided system to communicate with and utilize the included FPGA. The first example run was a matrix multiply example provided by Impulse Accelerated Technologies, which typically showed a ten times speed increase using the FPGA over the CPU. Altering this example, however, to compute basic arithmetic resulted in a three times decrease in speed of the FPGA. This is likely due to our using a larger data set, whose elements are only used once, and that these operations were too simplified to make use of parallelism. Altering the addition benchmark to use a smaller set of data and reading one entire array into the FPGA required us to remove the hardware pipeline. After this change was made, the FPGA again ran about three times slower than the CPU. Figure 1 summarizes our findings from the simple benchmark tests and the matrix multiply (MM) and European Options (E-O Sim) examples.

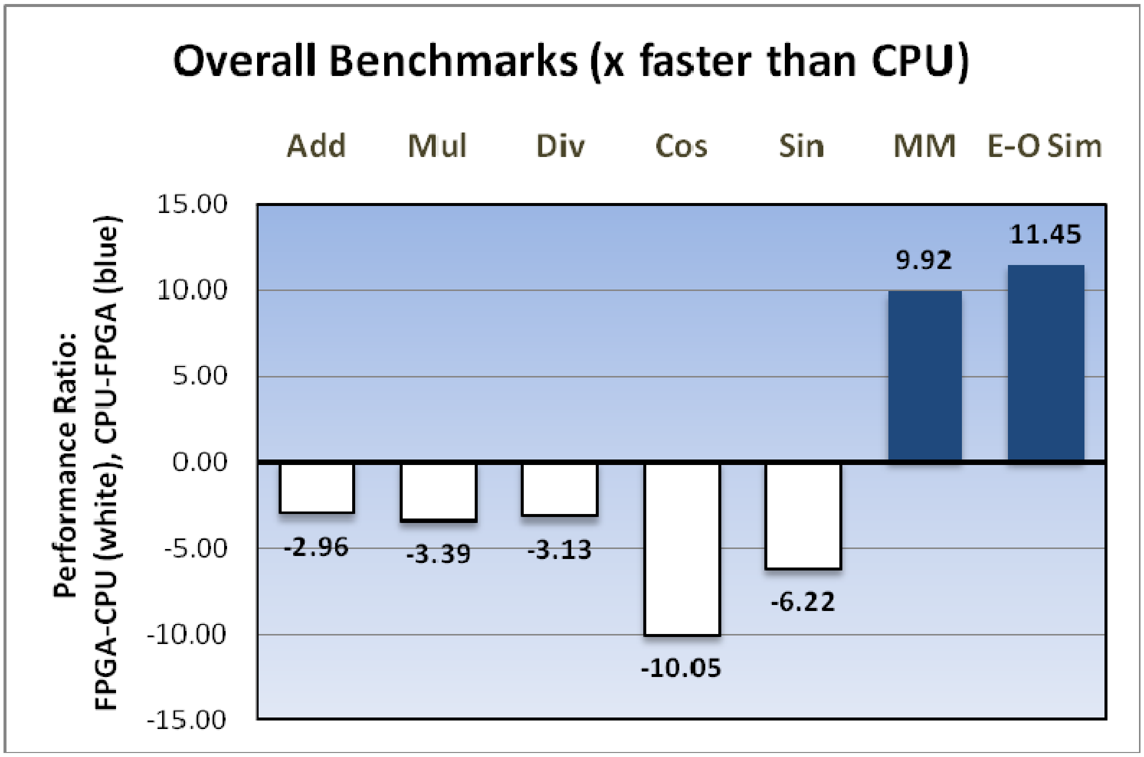


Figure 1: Overall Results (FPGA Times Faster than CPU)

Implementing these functions using Impulse C provided insight into the future of FPGA coprocessors. These conclusions include both the actual observed capabilities of the technology and limitations of the development tools.

1. FPGA coprocessors can significantly increase performance of certain mathematical operations. Certain functions, however, are better able to be run on an FPGA. The inherent ability to parallelize the function and the function's complexity factor into the FPGAs ability to outperform the CPU. Simple arithmetic and large data sets perform poorly on the FPGA compared to a modern processor. Since the FPGA has to overcome a slower clock and I/O overhead, the operations per datum must be considered. More complex operations that can be easily parallelized and use smaller data sets (like matrix multiplication, which incorporates 512 operations per datum, rather than 0.5 for arithmetic) can be accelerated to up to ten times faster than the CPU.
2. The XtremeData XD1000 uses a large Altera Stratix 2 FPGA, but in practice its performance was limited by several factors.
    a. The fabric of the FPGA only allows for a limited number of components. About 32 or so double precision adders or multipliers can be implemented into the FPGA, which only allows for a limited number of operations and limits precision for approximating series. In addition, the Quartus development tools occasionally run out of system memory during building of complex projects, causing it to crash. As a result, Quartus will only allow 16 parallel multiplier/adders, even though it can fit 32.
    b. The SRAM on the FPGA is only about 9Mbit, which limits the amount of data that can be loaded into the FPGA during processing

   c. The shared memory on the motherboard is limited to 4MB. This restricts the size of data on the system.
   d. After implementation, the internal clock of the FPGA typically runs at only 100-150MHz, which is many times slower than a 2-3GHz CPU.

3. Impulse Accelerated Technology's compiler, CoDeveloper, is useful in providing the communication interface between the FPGA and CPU; however, it has its limitations.
   a. Experience with the tools is required in order to optimize code and limit errors.
   b. The tools are fairly new and are constantly changing/evolving, so new developers need frequent support from Impulse, especially at first, before they understand the idiosyncrasies of the tools.
   c. Bugs currently in the alpha version of CoDeveloper 3.0 are substantial. Timing problems make development on the XD1000 system impossible without rolling back to a previous version. The older versions have limited capabilities and need patches to be used without errors.
   d. Current Altera and XtremeData libraries are limited to simple arithmetic. Square root and transcendental functions are not included in these libraries. Implementing these functions requires programming an approximation in VHDL with a wrapper to interface with CoDeveloper, or coding a primitive in C, which CoDeveloper then translates into HDL. In practice, bugs in the compiler did not allow us to program complex primitives (like natural log) in C and resulted in problems when programmed on the XD1000.

Recommendations

Following the research conducted during this project, we developed several recommendations for future research into this technology.

1. Wait for a stable release of Impulse C CoDeveloper, or find an alternative for programming the system in order to avoid problems experienced using the alpha releases.
2. Future researchers should familiarize themselves with the currently available tools and continue the research that this project started. Improving the transcendental projects that we developed, including trigonometric, logarithmic, and exponential functions would be the next logical step. Following these evaluations they should test mathematical models with the FPGA coprocessor. In addition to these functions, testing other mathematical operations that can be easily parallelized would also contribute to this research.
3. Impulse provided a Black-Scholes European Options model for the XD1000 system at the end of this project. Due to time constraints, we were unable to adequately evaluate this implementation for accuracy. A continuation of this project would benefit from evaluating this design.
4. Our ability to test complex functions on the XD1000 was limited by our knowledge of VHDL wrappers and the development tools. Becoming familiarized with this process would open the door to testing many different mathematical operations.
5. FPGA technology is constantly improving. Keeping up to date on their currently available capabilities is essential. The research presented in this project was inconclusive as to the viability of FPGAs in the financial sector; however, improvements over time could make this the way of the future.

A full copy of this report can be obtained by contacting Dr. Jason Choy at JP Morgan Chase, at jason.k.choy@jpmchase.com

Accelerating Financial Applications through Specialized Hardware – FPGA

A Major Qualifying Project Report
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements
for the Degree of Bachelor of Science

By:

____

Tri Dang

____

John Rothermel

Date: December 14, 2007

Report Submitted To:

Prof. Xinming Huang, WPI
Prof. Arthur Gerstenfeld, WPI
Prof. Michael Ciaraldi, WPI
Dr. Jason Choy and JP Morgan Chase Inc.

ABSTRACT

This project will investigate Field Programmable Gate Array (FPGA) technology in financial applications. FPGA implementation in high performance computing is still in its infancy. Certain companies like XtremeData inc. have advertized speed improvements of 50 to 1000 times for DNA sequencing applications using FPGAs, while using an FPGA as a coprocessor to handle specific tasks provides two to three times more processing power. FPGA technology increases performance by parallelizing calculations. This project will specifically address speed and accuracy improvements of both fundamental and transcendental functions when implemented using FPGA technology. The results of this project will lead to a series of recommendations for effective utilization of FPGA technology in financial applications.

In the future, FPGA technology could improve other computing as well. The data compiled during this project could be used to compare the benefits and cost of FPGA implementation for varying applications. For example, high performance computing is often essential for scientific research and military applications. This project will explore the potential for FPGA technology in the future of high performance computing.

AUTHORSHIP

Tri Dang – Tri wrote the FPGA and IEEE754 standards section of the Background. In addition, Tri wrote the FPGA FPU and Impluse C section of the Methodology. Tri also contributed to the editing of the report.

John Rothermel – John wrote the Executive Summary, Abstract, Introduction, and Financial Systems and Challenges section of the Background. He also wrote the AMD Opteron (10h Family) and Intel Core Microarchitecture as well as the Benchmark Testing sections of the Methodology. In addition, he wrote the Results and Conclusions as well as the Recommendations.

ACKNOWLEDGEMENTS

We would like to graciously thank the following organizations and individuals for their help in this project:

- JP Morgan Chase for their involvement in this process, including providing the facilities and tools necessary during the course of the project
- Dr. Jason Choy, our sponsor, whose advice and guidance was invaluable in the development of the project
- Our advisors, Professors Gerstenfeld, Huang and Ciaraldi for their support and guidance
- AMD, Inc. for providing XD1000 systems to JP Morgan Chase and WPI for development
- XtremeData, Inc. for their technical support during setup of the XD1000 system
- Impulse Accelerated Technologies, Inc. for their considerable support during the project using their development tools
- The New York Public Library for making their online databases and books available for our use
- The various universities, organizations, and individuals whose works were cited in this report and provided information essential to our understanding of the project

TABLE OF CONTENTS

EXECUTIVE SUMMARY i

Results iii

Recommendations vii

ABSTRACT x

AUTHORSHIP xi

ACKNOWLEDGEMENTS xii

TABLE OF CONTENTS xiii

TABLE OF FIGURES xvi

TABLE OF TABLES xix

1.0 INTRODUCTION 1

2.0 BACKGROUND 4

2.1 Financial Systems and Challenges 4

2.1.1 Mathematical Models and Simulations Applied in Computer Systems 5

2.1.2 High Performance Computing (HPC) 9

2.2 Field Programmable Gate Array (FPGA) Technology in HPC 16

2.2.1 History of FPGA technology 16

2.2.2 FPGA Architecture 17

2.2.3 Advantages and Disadvantages of FPGAs 22

2.2.4 Available FPGA Programming Languages 24

2.3 IEEE754 Floating point standard 30

2.3.1 Importance and History of IEEE754 Floating Point Standard 30

2.3.2 Definition and Standard Convention 31

2.3.3 Floating point arithmetic 41

2.4 Summary 53

3.0 METHODOLOGY 54

3.1 Current Methods for Floating Point Calculations 54

3.1.1 FPGA FPU 55

3.1.2 AMD Opteron (10h Family) Microarchitecture 69

3.1.3 Intel Core Microarchitecture 77

3.1.4 Summary 80

3.2 Benchmark Testing using Fundamental Mathematical Functions 81

3.2.1 XtremeData Inc., XD1000 System 81

3.2.2 Benchmark Testing on the Opteron 82

3.2.1 Measuring Accuracy 86

3.2.3 Benchmark on FPGA 87

4.0 RESULTS AND CONCLUSIONS 111

4.1 Problems Encountered 111

4.1.1 Set Up 111

4.1.2 Communication 112

4.1.3 Impulse C Implementation 113

4.1.4 XD1000 System Limitations 116

4.1.5 System Memory Limitations on the User PC...................................... 116

4.1.6 Summary... 117

4.2 CPU Benchmark Results... 118

4.3 FPGA Coprocessor Results.. 122

4.3.1 Matrix Multiplication (Double Precision) 122

4.3.2 Addition and Multiplication 125

4.3.3 Division 129

4.3.4 Square Root and Transcendental Functions 131

4.3.5 Impulse Provided European Options Example.................................... 136

4.4 Conclusions 139

5.0 RECOMMENDATIONS... 141

6.0 WORKS CITED 144

TABLE OF FIGURES

Figure 1: Overall Results (FPGA Times Faster than CPU) iv

Figure 2: Monte Carlo Approx. of PI 8

Figure 3: Early Computer Performance (1950-1975) 10

Figure 4: Performance Cost over Time 11

Figure 5: Custom, Hybrid, and Commodity Processor Improvements over Time 12

Figure 6: Prevalence of Custom, Hybrid, and Commodity Based Systems 13

Figure 7: HPC Arch over Time 15

Figure 8: Generic FPGA architecture 18

Figure 9: Configurable Logic Block (CLB) 19

Figure 10: I/O Blocks Circuitry of a FPGA 20

Figure 11: Programmable Interconnects 21

Figure 12: Design Flow- Hardware (left) Software (Right) 25

Figure 13: Level of Abstraction of different languages 26

Figure 14: Hardware designing with Impulse C 29

Figure 15: Flush to Zero (above) and Gradual Underflow (below) 35

Figure 16: FPGA's Addition/Subtraction circuit 59

Figure 17: Leading "1" Detection block 60

Figure 18: Multiplication unit 62

Figure 19: Division unit 65

Figure 20: Square root unit 67

Figure 21: Block Diagram for AMD Opteron (10h Family) Processors 70

Figure 22: Application Programming AMD Opteron (10h Family) Registers 73

Figure 23: FPU for AMD Opteron (10h Family) Microarchitecture 74

Figure 24: Intel Core Microarchitecture 77

Figure 25: XD1000 HW Specification 81

Figure 26: Opteron Benchmark Flowchart 84

Figure 27: Random Number Generator Flow Chart 85

Figure 28: Developing Process with Impulse C 92

Figure 29: Generic top level block diagram for Comparison Performance Project 93

Figure 30: CPU process flow chart 97

Figure 31: Flow chart for IO process 99

Figure 32: Flow chart for mmproc 101

Figure 33: Matrix Multiply on XD1000 (replace!) 123

Figure 34: Matrix Multiplication showing runs (blue) and average (red) 124

Figure 35: Addition Benchmark on the XD1000 125

Figure 36: Multiplication Benchmark on the XD1000 126

Figure 37: Addition Benchmark Results, each run (red) and average (blue) 127

Figure 38: Multiplication Benchmark Results, runs (blue) and average (red) 128

Figure 39: Addition Test Version 2 129

Figure 40: Division Benchmark Results, each run (blue) and average (red) 131

Figure 41: Sine Benchmark Results, Results per Run (blue) and Average (red) 133

Figure 42: Cosine Function Benchmark, Times per Run (red) and Average (blue) 135

Figure 43: European Options Simulation (FPGA Times Faster), Ind Runs (blue) and Average (red) 138

Figure 44: European Options Simulation (CPU Times Faster), Ind Runs (blue) and Average (red) 138

Figure 45: Overall Results (FPGA times faster than CPU) 140

TABLE OF TABLES

Table 1: Latency Comparison of Advertized Required Cycles for Opteron (10h Family) and Core ii

Table 2: Latency Comparison of Advertized Required Time (ns) for Opteron (10h Family) and Core (based on 2.5GHz core) ii

Table 3: Throughput Comparison (all double precision SIMD) ii

Table 4: CPU Benchmark Results iii

Table 5: Summary of Floating Point Representation 33

Table 6: Range of Values for Floating Point Numbers 36

Table 7: Summary of Floating Point Representation. 38

Table 8: Special Operations 39

Table 9: Invalid Operations 40

Table 10: Latency Comparison of Advertized Required Cycles for Opteron (10h Family) and Core 79

Table 11: Latency Comparison of Advertized Required Time (ns) for Opteron (10h Family) and Core (based on 2.5GHz core) 79

Table 12: Throughput Comparison (all double precision SIMD) 79

Table 13: CPU Benchmark Results 119

Table 14: Matrix Multiply Results (10 runs) 124

Table 15: Addition Benchmark Test (10 runs) 127

Table 16: Multiplication Benchmark Results (10 runs) 128

Table 17: Division Benchmark Results (10 runs) 130

Table 18: Sine Benchmark Results 133

Table 19: Cosine Benchmark Results 134

Table 20: European Options Simulation Results 137

1.0 INTRODUCTION

Technology is continually moving forward. In today's world electronics are constantly getting smaller and more powerful. This is especially true in the computer world where speed has been improving exponentially since the 1960s. According to Moore's Law, the number of transistors on a chip doubles every two years. This exponential growth leads to ever increasing computing power at lower cost [3].

In the past, increased computing was achieved by increasing the speed of the processor. Due to physical limitations, however, this approach is no longer practical. Faster processors use more power and generate more heat, which then must be dissipated. In addition, processors are subject to certain bottlenecks due to limited memory addressing and multilayered memory [4-6].

One possible solution is implementing Field-Programmable Gate Array (FPGA) technology as a coprocessor to the existing processor in the system. FPGAs have the advantage of much lower power consumption and the ability to do tasks in parallel while sacrificing flexibility [5, 6]. According to FPGA Acceleration in HPC: A Case Study in Financial Analytics (2006):

> "An FPGA coprocessor programmed to hardware-execute key application tasks can typically provide a 2X to 3X system performance boost while simultaneously reducing power requirements 40% as compared to adding a second processor or even a second core. Fine tuning of the FPGA to the application's needs can achieve performance increases greater than 10X" (p. 2).

It is possible to use the strengths of both traditional processors and FPGAs to take computing to the next level of performance.

In the world of finance, computing power directly translates into money. Investment banks use large clusters of computers to calculate and minimize risk. The Federal Deposit Insurance Corporation (FDIC) then determines the risk-weighted assets (RWA) based on these calculations. RWA refers to the amount of capital within banks that are weighted by risk. For large banks, this will often translate into billions of dollars [1]. Unfortunately, risk calculations must take into account many different situations, often using Monte Carlo simulation. These simulations are computationally intensive and can take days to complete [1]. Improving the computing power of these systems would decrease the time required to run simulations, allowing investment banks to run further models and better approximate risk. This lowered risk would then translate into less money held in capital reserve, which is a special fund where large amounts of money are set aside for special projects and expenses. Lower, yet sufficient capital held in the reserve fund would allow for increased available monetary resources within the bank.

The purpose of this project is to evaluate FPGA coprocessors for financial calculations in order to determine the feasibility of implementing them into existing clusters. This will be done by completing three objectives:

1. Research current methods of computing in microprocessors Intel/AMD and FPGAs to determine physical limitations of both platforms.
2. Test microprocessor and FPGA implementations of different fundamental mathematical functions using benchmarks to find timing differences.
3. Implement complex functions, such as trigonometry and logarithm, whose approximations are based on fundamental mathematical functions, to determine physical limitations and timing differences between FPGA and microprocessor technology.

In order to meet these objectives, we will examine code for different implementations of the algorithms in which we are interested. We will determine the strengths and weaknesses of these methods and then use that data to aid further research. Following this phase, we will use a system to test performance of the CPU alone and with an FPGA coprocessor to determine improvement of the system. These tests will allow us to make recommendations to an investment bank as to further steps in developing this technology if it is found feasible.

In summary, advances in computer technology have increased computing speeds exponentially in past decades. This has usually been achieved by using more transistors and raising processor speed. As processor speeds improve, however, power requirements also increase. FPGA coprocessors have been suggested as a possible solution to continue raising processing power. This increase is especially important in the finance sector where financial risk analysis can translate into billions of dollars profit or loss. This project will address the feasibility of FPGA coprocessors in computing financial risk calculations.

2.0 BACKGROUND

In the following section, we will discuss several important issues in determining the feasibility of FPGA coprocessors in financial applications. First, we address how financial institutions approximate performance of different markets and the challenges inherent in this process. We then examine current methods of high performance computing and how technology has evolved. Next, we discuss FPGA technology, including its strengths, weaknesses, and the development process. Finally, in order to better understand low-level computations, we outline the current floating point standard as defined by IEEE754 as well as mathematical operations using this standard. This information will provide the base of knowledge upon which we built this project.

2.1 Financial Systems and Challenges

Predicting market performance is essential for investment banks. This process allows banks to maximize profits and minimize risk. The FDIC was created by Congress in order to keep the national financial system stable and to ensure customer confidence [7]. The FDIC mandates a deposit amount from banks based on its ability to predict risk and other factors. The deposit could, for instance, fall between five and eight percent [1], which would correspond to billions of dollars in a large investment bank's portfolio. Banks typically determine this risk based on financial models and computer simulation. Unfortunately, randomness is inherent in the market, so simulations need to take into account many possible outcomes.

2.1.1 Mathematical Models and Simulations Applied in Computer Systems

One of the most commonly used models for determining the future value of options is the Black-Scholes model. This model has proven to be fairly accurate over time and is the cornerstone for much of the market prediction models used by investment banks. This model is then often applied in simulation in order to determine the future values based on randomness. In many cases Monte Carlo simulation is used for this purpose [8]. The combination of the Black-Scholes model and Monte Carlo simulation allows for accurate predictions that account for randomness in the market.

2.***1.1.1 Black-Scholes Model***

The Black-Scholes Model was developed by Fischer Black and Myron Scholes around 1970 as a model for finding the future value of options, which were not common then. Although their work was not widely accepted at the time, it has become one of the most widely accepted financial models today [9].

According to Webster's Dictionary, an option is, "a right granted by a corporation to officers or employees as a form of compensation that allows purchase of corporate stock at a fixed price usually within a specified period" [10]. For example, a person receives an option from Company A. This option allows the individual to buy 100 shares of Company A's stock at the current market price of one dollar in one year. If in one year Company A's stock price is at two dollars, the individual would likely exercise the option (i.e. buy the stock), which would allow him to purchase 100 shares of Company A's stock for one dollar each, or one hundred dollars. The shares, however, are worth two dollars each. If the individual immediately sells the option at the current market price, he would earn two hundred dollars less the cost of the shares, or one hundred dollars. If, on the other hand, the stock price in a year was worth fifty cents, the individual would not exercise the option and it would therefore be worthless. An option, therefore, for an individual cannot result in a loss, but only increases in value with a company or becomes worthless.

The Black-Scholes model for approximating future value of options is based on the Brownian motion model, which is a form of Marchov processes [8]. The Brownian motion model is given below:

$$dS = \mu S dt + \sigma S dW$$

Where S, in this case, is the value of a security, μ is the drift rate, σ is the volatility, W(t) is a standard Brownian motion, and dt is a time increment [8].

The Black-Scholes model is a partial differential equation given by the following expression:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

Where the value V(t,s) changes over the interval $0 \leq t \leq T$, where T is the time at maturity. In addition, r is the risk-free interest rate, which is the rate of change for short-term non-risk securities, such as bank accounts [8].

This model, however, makes certain assumptions, which are also its limitations. It first assumes that the market has no transaction costs, which is rarely the case. It assumes market price varies according to the geometric Brownian motion process. It is assumed that the security is perfectly divisible (i.e. fractions of shares are possible). In addition, trading is continuous and the risk free rate is assumed constant. Finally, the no-arbitrage principle is held [8]. This principle states that shareholders pay for the actual cash flow, not earnings [11]. While assumptions are essential in mathematical models, it is important to realize their limitations.

In practice, the Black-Scholes model is often altered depending on the financial institution and other factors. Due to the complexity of the model, computers typically do the computations. Many spreadsheet programs are available today that utilize this model. Although other models are available, Black-Scholes is the best known model for derivatives [12].

2.1.1.2 Monte Carlo Simulation

Monte Carlo Simulation is a simulation method based on random numbers, which is often used in order to determine multidimensional integrals and simulate stochastic (i.e. random) phenomena [13]. As opposed to, for instance, simply using averages, simulation allows an individual to know the probability and range of expected outcomes. This method enables an investor or manager to make better decisions that take into account the variability of the process [14].

One application of Monte Carlo simulation is determining π to a certain precision. This is done by first taking a unit square and then drawing a quarter circle inside it as seen in Figure 2. We then use a random number generator to pick points within the square. If the points are within the area of the quarter circle, they are recorded as hits. If one then takes the ratio of hits to the ratio of coordinates generated and multiplies by four, the result is an approximation for π, which becomes better with more points [13].

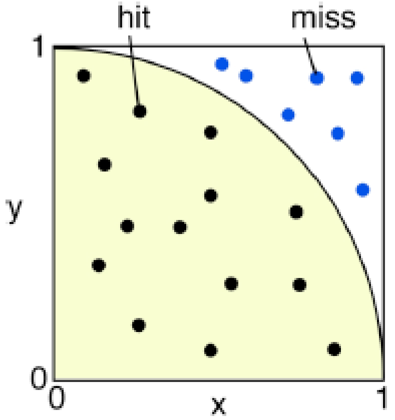


Figure 2: Monte Carlo Approx. of PI

Another example of Monte Carlo simulation is in predicting the market for a new product. First, assume that a new product will sell between ten-thousand and fifty-thousand units and that the probability is defined by a normal curve. A Monte Carlo approach would pick random values between these points according to a normal distribution (i.e. pick fewer values near the edges and more near the center) and then use these values for further analysis. The simulation could then multiply these values by an expected sale price less the production cost per unit to determine profit. Using this simulation, the company would know the range of possible profit and the probability of each range of profit.

While Monte Carlo simulation is useful for evaluating processes that involve randomness, the assumptions that are made can lead to inaccuracy. As in the previous example, we assumed that the sales would be between ten and fifty thousand and that the probability of sales would follow a normal distribution. If these assumptions are incorrect the simulation will be inaccurate. Care must be taken to choose values that properly represent the process.

Monte Carlo simulation is used frequently to model processes that involve randomness. Although it is a powerful tool, care must be taken to make appropriate assumptions during the simulation process. Monte Carlo simulation can be used to provide ranges and probability of outcomes, which allow an individual to make informed decisions.

2.1.2 High Performance Computing (HPC)

High Performance Computing (HPC) refers to the best currently available class of computing systems. Computers, however, are continuously improving, which leads to a changing HPC standard. In this section, we discuss the evolution of HPC and the current standards for supercomputers.

2.1.2.1 History of HPC

The development of computers began during World War 2 when the United Kingdom developed the first fully electronic, digital computer, called the Colossus Machine. This computer was used to break German ciphers and was classified until the 1970s [15, 16]. Following the development of the Colossus Machine, the better known ENIAC was finished in 1945 for calculating artillery ballistics. This trend in supercomputer development for the government would continue for years with the Department of Defense (DoD). The DoD developed the majority of the computers until the late 1950s when the National Security Agency (NSA) began using supercomputers for cryptography [15].

Figure 3 shows early development of high performance computing until the 1970s in millions of operations per second (MOPS). From about 1950 until 1970, the world’s best computers improved from about 0.5 thousand operations per second (KOPS) to 10 MOPS, a difference of 20,000 times.

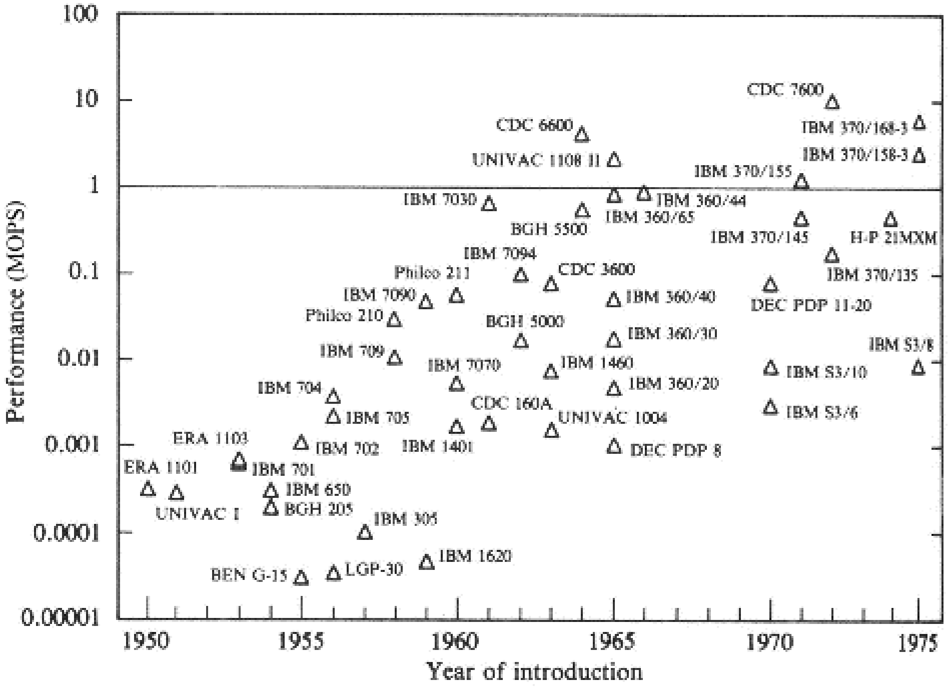


Figure 3: Early Computer Performance (1950-1975)

The next big development in computing would come in late 1971 with the development of the microprocessor [15, 17]. This computer processor developed by Intel was a shift in development from custom processors to lower performance general use processors. While they were not largely used in supercomputers for quite some time, general use processors allowed lower end systems to be built at much lower cost. The trend of increased processing power at lower cost has continued to the present. Figure 4 displays this trend up to 2000 [15].

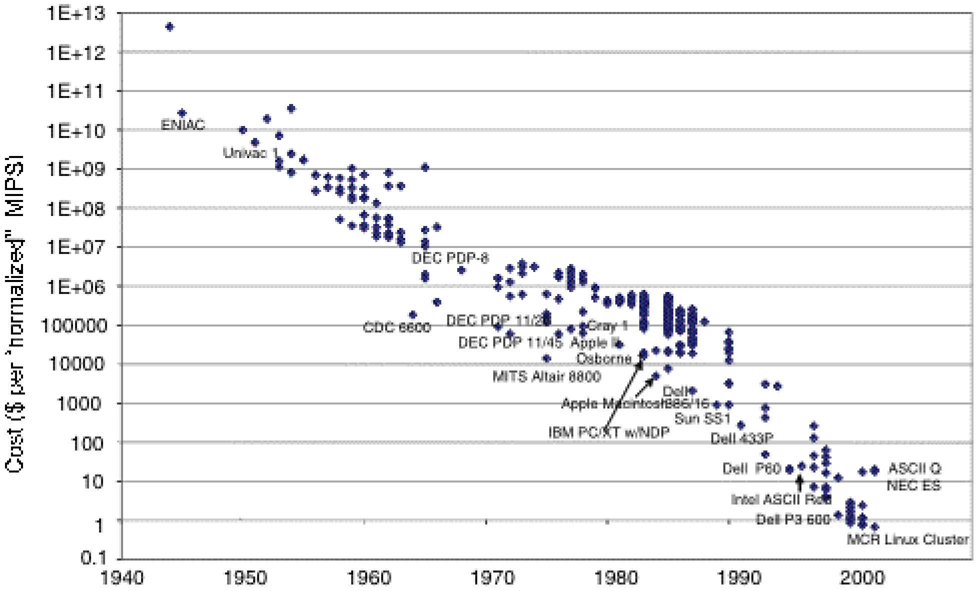


Figure 4: Performance Cost over Time

In 1976 Cray Research developed the Cray-1. This supercomputer utilized an innovation known as vector architecture. This involved storing vectors of floating point data into vector registers when called, which could then be manipulated in an arithmetic unit at much higher speeds. Following vector architecture, massively parallel processor (MPP) systems began to be developed. This architecture involved increasing computing power by using large numbers of individual processors [15]. The increase in performance, however, would not be linear as Amdahl's Law states that the performance gain of additional processors is much less than the sum of the processors (e.g. it will take nine processors to triple the performance of one). [18].

As the 1980s and 1990s progressed, custom processors and connections, once essential for HPC, began to become less practical. Complimentary metal-oxide semiconductor (CMOS) transistors began to be used in commercial processors. Large numbers of CMOS transistors could be placed on a single die, which lead to the microprocessor. These were used increasingly throughout the 1980s and 1990s in certain systems, which could perform nearly as well as customized solutions at lower cost. Figure 5 illustrates performance increases in supercomputers using customized, hybrid, and commodity (general use) processors over time for the world's best computer systems. While custom systems in the mid 1990s were significantly better than commodity systems, the rate of improvement over time for commodity processors was about 111% annually compared to 94% for custom systems [15].

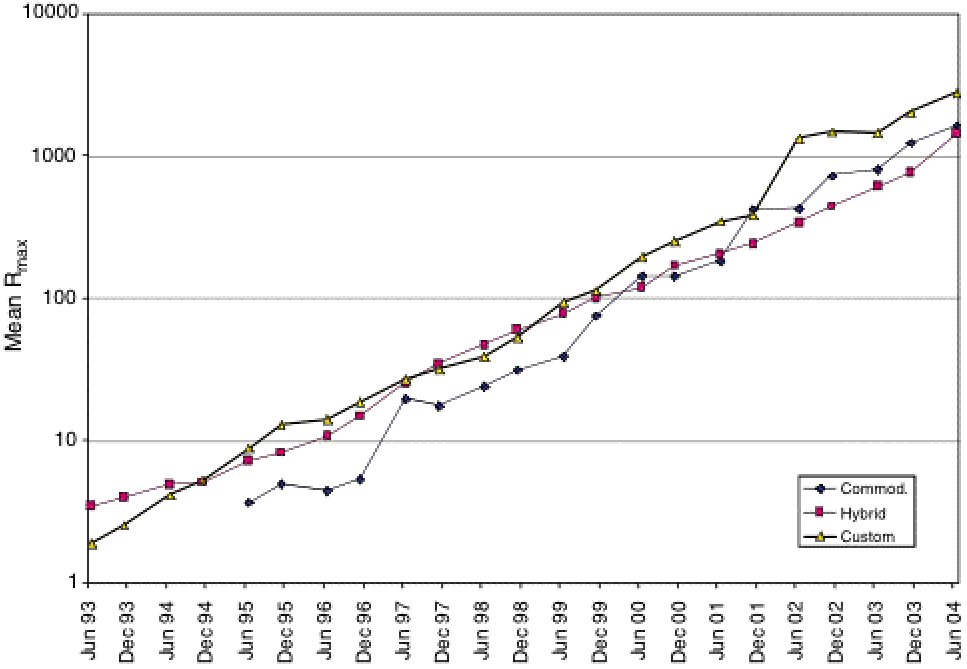


Figure 5: Custom, Hybrid, and Commodity Processor Improvements over Time

Due to the increasing performance of commodity-based systems, customized solutions have largely fallen out of favor as can be seen in Figure 6. The trend of the future appears to be inexpensive HPC by utilizing commodity processors.

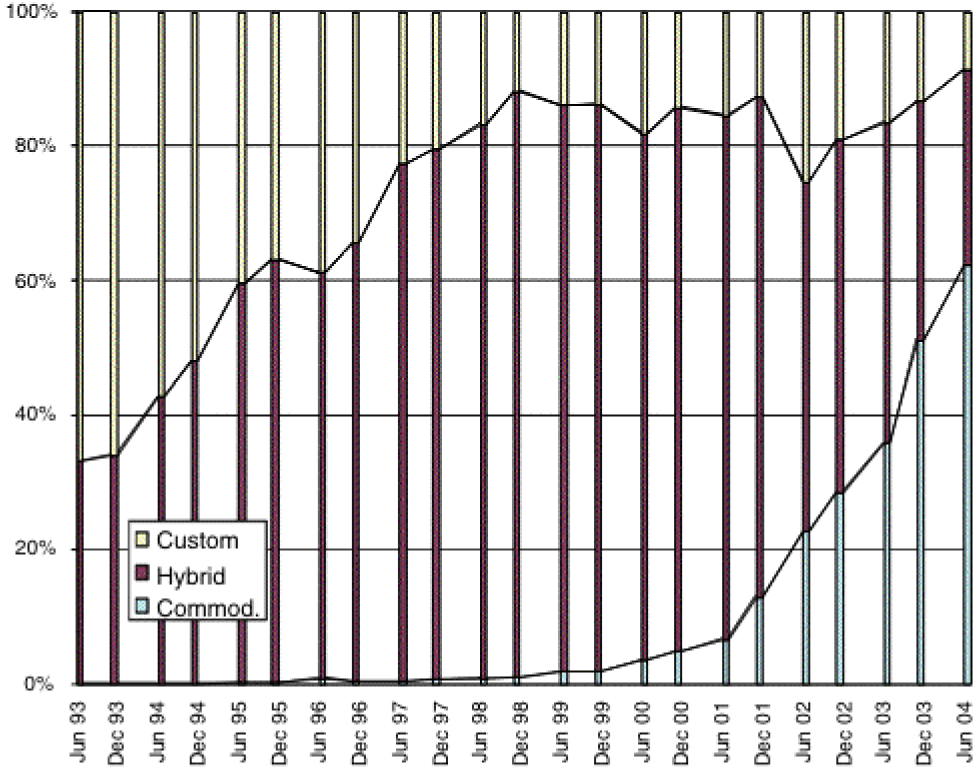


Figure 6: Prevalence of Custom, Hybrid, and Commodity Based Systems

2.1.2.2 Architecture

A supercomputer has several main components. These include processors, memory, the I/O system, and the connections. The basic function of a processor is to execute program functions including arithmetic and logical functions. The memory allows the system to store the current state of the machine. These two components make up a node, which can consist of single or multiple processors and a memory system of varying size. Typically hundreds or thousands of nodes are used in a typical supercomputer. The I/O system allows computers to access peripheral devices such as disks. Finally, the connections allow the system components to communicate with each other. All four components allow the system to work properly [15].

A number of architectures have been used in supercomputers over the years. The first and simplest is a single processor design. While single processors are rarely used even in personal computing currently, they were used in the past before computing demand exceeded their capabilities.

Another system that was used somewhat in the early nineties was single instruction multiple data (SIMD) systems, which could apply a single instruction to vectors of data to increase performance. These systems have since fallen out of favor as companies move to different methods of HPC.

One system, utilized primarily in the early 1990s, was called symmetric multiprocessing (SMP). SMP utilizes multiple processors with a common memory block per node. A system called massively parallel processing (MPP) involves having hundreds or thousands of individual processors working in parallel. Typically SMP and MPP systems utilize custom processors and/or components, which adds to their cost.

Currently the most common architecture for HPC is the cluster. Cluster architecture is achieved by connecting computers via a network so that they work similarly to a single computer. Often clusters are built by connecting several computers with the same hardware; however, clusters can be built using computers with different hardware. The advantage of this architecture is simplicity. Any group of computers can be connected using gigabit Ethernet cards, which are typically available in any new computer, or any other network interface [5, 15]. A variant on the cluster is called the constellation, which is similar, except that the individual computers contain SMP nodes [19]. Constellations seemed to be the wave of the future in the early 2000s, but recently are being used less by the world's fastest computers.

Figure 7 shows prevalence of HPC architectures over time for the 500 fastest computers in the world as rated by Top500 [20]. Cluster computing has increased substantially since 2000. SMP architecture became nearly nonexistent since 2001 in the top performing computers. In addition, MPP and constellation systems decreased significantly in past years. This is likely due to the cost-effectiveness and ease of implementation of cluster computers [5, 15, 21].

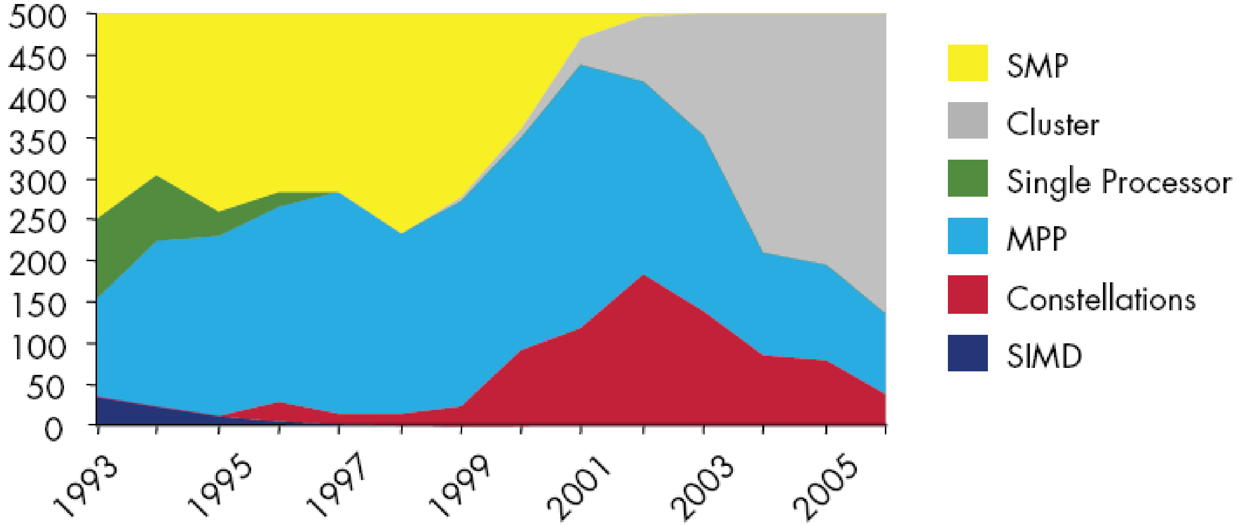


Figure 7: HPC Arch over Time

Although many different architectures and technologies have developed in the past, cluster computing appears to be the way of the future. This architecture is scalable, easily implemented and less costly than its competitors. Cost-effectiveness is achieved by utilizing commodity processors, which do not require significant development to implement, unlike MPP and SMP systems. The future of computing is likely to continue balancing performance, cost, and ease of implementation.

2.2 Field Programmable Gate Array (FPGA) Technology in HPC

Previously we discussed a number of methods that are used in HPC. Certain methods, especially clusters, are preferred in HPC because of their cost-effectiveness. However as a system expands due to increasing demand in speed and accuracy, higher power dissipation is required as the result of increasing clock speeds and additional processors. Although there are solutions that address this problem, such as multi-core processors or improving the cooling system, they have not completely solved the issue of power consumption.

One approach to solving this problem is using FPGA technology. FPGAs are semiconductor devices that contain programmable logic components, logic blocks, and programmable interconnects. FPGAs are programmed in the field rather than at fabrication. Like gate arrays, FPGAs have arrays of uncommitted logic. As the device is programmed, the configuration of the array is determined by applying a large negative voltage to individual connections [22]. Previous FPGA generations can contain up to 60,000 free programmable logic blocks, 64 kilobits of RAM and operate at clock frequencies of a few megahertz [23], while more modern processors have considerably more RAM and may operate at a few hundred megahertz, which significantly increases their capabilities. At first sight, FPGAs may seem inferior to the current generation of processors and their application-specific integrated circuit (ASIC) counterpart. However, FPGAs utilize parallel mechanisms and are fully programmable, which are the keys for its extensive development and applications today.

2.2.1 History of FPGA technology

Field programmable gate array technology originated from complex programmable logic devices (CPLDs) research in the 1980s. In 1984, Xilinx co-founder Ross Freeman invented the FPGA. “The concept requires a lot of transistors, but at that time, transistors were extremely precious", said Xilinx Fellow Bill Carter, who was the eighth employee to be hired in the company in 1984 [24]. However, Ross never worried about this problem, due to Moore's Law. As the result, in 1985, Xilinx released the first FPGA, the XC2064 chip with 1000 gates. Remarkably, by the year 2004, the size of an FPGA has already exceeded ten thousand times this size [25].

As the size of FPGAs increased, schematic design entry became less practical. Evidently, once FPGAs reached a threshold of 100,000 gates, the idea of implementing and verifying the chips through schematic entry became impossible. Consequently, hardware description language (HDL) was introduced to the field. Although HDL was invented nearly three decades ago, due to the lack of synthesis tools, application of FPGAs were limited to simple chips. As the speed of the HDL simulators increased, FPGA design shifted swiftly to HDL design entry [25]. For a better understanding of FPGAs, the following sections will provide an in depth description of generic FPGA architecture.

2.2.2 FPGA Architecture

Although each FPGA has its own specific architecture, the general structure consists of configurable logic blocks, input-output (I/O) blocks and programmable interconnects. In addition, the structure should have circuitry to create a clock signal. Modern FPGAs can also include simple logic blocks such as ALUs, decoders, and memory [26]. According to Zeidman, there are three types of programmable elements in an FPGA: static RAM, anti fuses, and flash EPROM. In short, most of the FPGA structures are variations of the one shown in Figure 8 below [27].

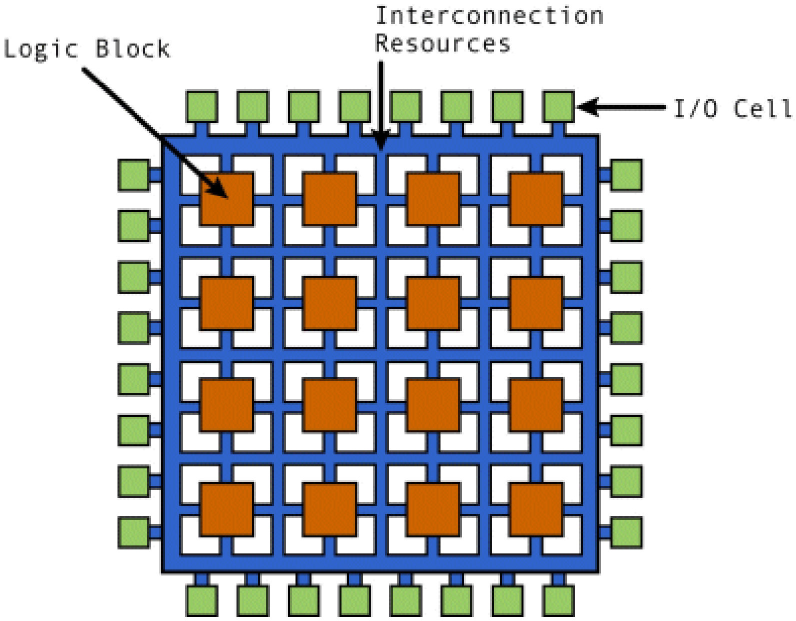


Figure 8: Generic FPGA architecture

2.2.2.1 Configurable Logic Blocks (CLBs)

Most of the FPGA's logic is described in the configurable logic blocks. On average, one FPGA should have a logic circuitry intensive enough to create a state machine. Figure 9 shows an example of one simple configurable logic block [27]. As shown, CLBs are required to have some memory, like RAM, to create combinational logic functions and look up tables. They contain some flip flops to provide the block's logic memory for clock storage elements. In addition, CLBs also consist of multiplexers that help to route the logic within the block to communicate with external resources and to allow options such as reset, clear and input [26].

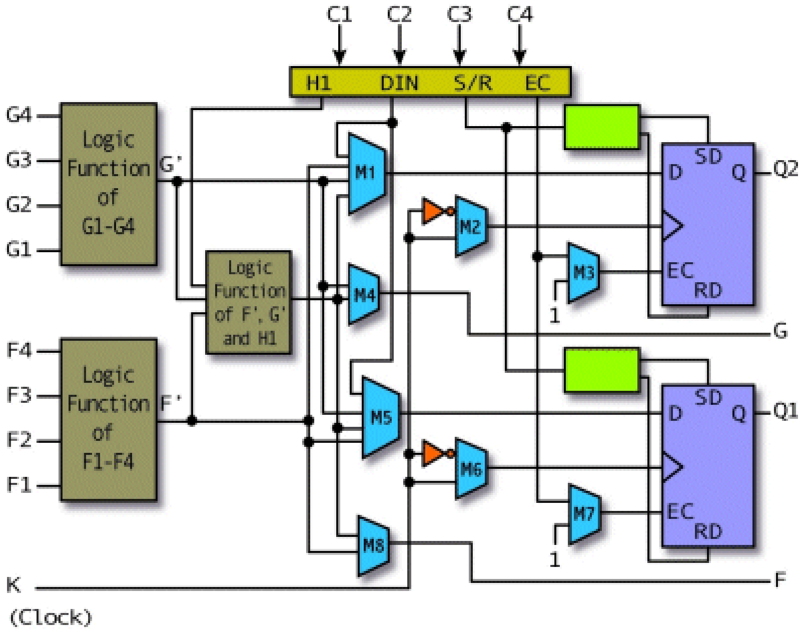


Figure 9: Configurable Logic Block (CLB)

2.2.2.2 Configurable I/O Blocks

Configurable I/O Blocks are used to communicate with external sources apart from the FPGA. Specifically, the block brings signals to the chip and displays the output signal. Hence, I/O blocks usually contain buffers at both the input and output side (typically tri-state for output) and open collector control circuitry. The control circuit has some pull-up resistors and sometimes pull-down resistors to terminate signals as well as buses without discrete external resistors installed to the chip [26].

Designers have the option to select the output as active high or active low, depending on their needs. Moreover, FPGAs also allow designers to adjust the output's slew rate for fast or slow rise time or fall time. I/O blocks almost always have flip-flops on the output side so that the clocked signal can be outputted without delay, which helps to meet setup time requirements of other external devices. Similarly, the input sides also have some flip-flops to reduce hold time requirements of the FPGA [26]. Figure 10 below provides a simple example of the I/O blocks circuitry of an FPGA [27].

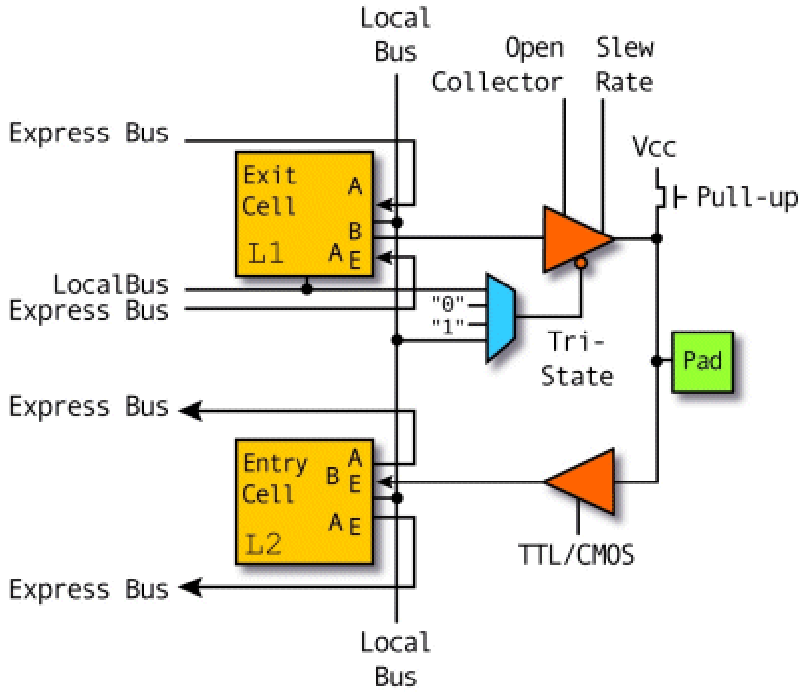


Figure 10: I/O Blocks Circuitry of a FPGA

2.2.2.3 Programmable Interconnects

Another important component of an FPGA's structure is programmable interconnects. This component is used to connect CLBs on the chip. Figure 11 below shows a portion of a programmable interconnects network [27].

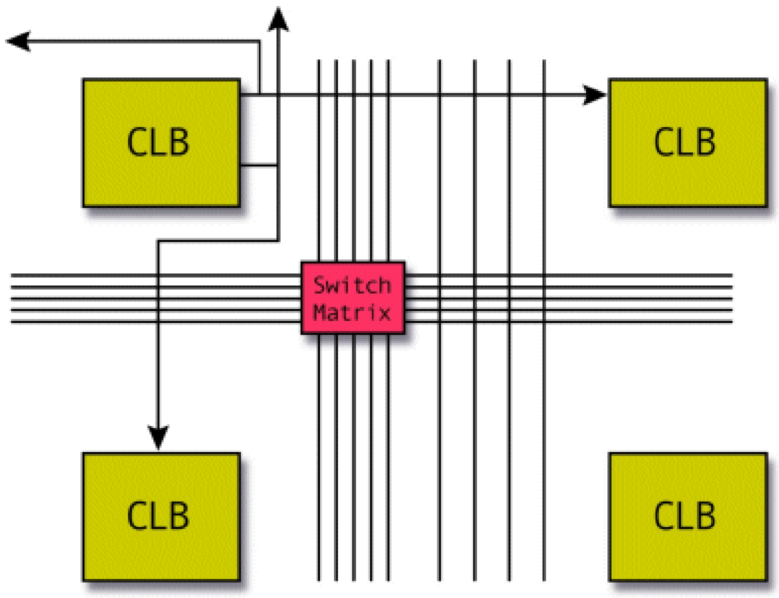


Figure 11: Programmable Interconnects

There are basically two types of interconnects: long lines and short lines. The long lines are used to connect CLBs that are physically far away from each other or are treated as buses within the chip. Specifically, three state buffers are used to connect CLBs with long lines to create buses. There are also special long wires, called global clock lines, which are used to connect clock buffers and clocked elements of each CLB block. These lines are designed to have low impedance for fast propagation times. This design is introduced to prevent skew between clocked signals within the chip. On the other hand, the short wires are simply used to connect closer CLBs [26].

In order to connect CLBs, connections are required to go through many transistors and switch matrices as shown in the figure above. As a result, in contrast to an ASIC where most delays come from the logic in the design, the majority of delay originates from programmable interconnects [26].

2.2.2.4 Clock Circuitry

The last major component of an FPGA is the clock circuitry. This circuit consists of special high drive clock buffers and clock drivers, which are distributed around the chip. The global clock lines are used to connect these drivers and clock input pads to distribute clock signals within the FPGA. Since skew and delay are not guaranteed anywhere except on these global clocked lines, FPGAs must be synchronous in design [26].

2.2.3 Advantages and Disadvantages of FPGAs

In 1984, Ross Freeman invented FPGA technology after being inspired to improve current ASIC technology [24]. Since then, FPGAs have become a promising technology that competes closely with ASICs. It is essential to understand the tradeoffs between these two technologies in order to make an appropriate choice in application.

Even though FPGA technology has become prominent in the semiconductor industry, it is still a new field with areas which require more investment. One of the drawbacks of FPGAs lies at its programmable interconnects component. Programmable interconnects rely on transistors or active buffers controlled by SRAM cells to work properly. These components add impedance into the route and make it impossible to estimate correct propagation delay [28]. For a logic designer, unknown delay can easily become disastrous, often consuming hours to debug. In addition, another disadvantage of FPGAs comes from the concept of the FPGA itself. As Xilinx stated in their "FPGA Embedded Processors – Revealing True System Performance" paper, design tools for FPGAs are complex because of the mixture of hardware and software platform design [29]. Moreover, most FPGA software design tools are immature and are difficult to use. In addition, device cost is also a consideration. Currently, most available processors on the market are less expensive than an equivalent embedded FPGA processor [29].

On the other hand, FPGAs have indeed proven to possess many advantages in the hardware design field. For example, FPGAs save great amounts of design time and cost compared to its ASIC counterpart. While the ASIC design process takes months to complete and one logical design flaw can result in a large expense to the vendor, FPGAs allow designers to change the logic and recompile in minutes with little effort, much like software [30]. Hence, FPGAs clearly allow engineers more options when solving a problem in the early stages of the design process and later adaptability by simply incorporating a specialized instruction set. Moreover, since each logic block, also called a processing element or PE, is designed to be as small as possible, FPGAs can maximize on-chip parallelism, which consequently increases processing time.

Another aspect that is worthwhile considering is an FPGA’s architecture. For instance, the entire on chip logical resources can be used to maximize the packing density of one particular method. Thus, FPGAs do not require a predefined limit on the number of logic cells per chip [30]. In addition, because each algorithm is programmed during in-field use, each algorithm is separately optimized without on chip constraints arisen from other algorithms. As a result, FPGAs can process an algorithm as fast as one or two clock cycles. Designers can also control the chip clock rate independent of any logic designs, which provides more flexibility in approaching a problem.

When mentioning FPGAs, one cannot dismiss the abundant resources available for hardware designers. Many FPGA vendors have begun offering intellectual property (IP) cores which include special functions (this is discussed further in the next section). All IP cores are designed, verified, and characterized for specific tasks. One can always modify available cores by adding or subtracting functions based on need. Most IP cores are also provided by more than one vendor, which allows the programmer more choice. Perhaps most importantly, IP cores reduce the time and manpower required for FPGA design [26].

FPGAs are a powerful tool in computing that are used in many applications. It is important, however, to be aware of their limitations when making design decisions. Many tools exist to improve the FPGA design process, including several programming languages, which are discussed in the following section.

2.2.4 Available FPGA Programming Languages

As mentioned earlier, FPGAs were originally designed by schematic design entry. However, as FPGA chip density increased exponentially, schematic design entry became less practical. Thus, hardware design languages (HDLs) came to play. The two traditional HDLs available are VHDL and Verilog.

The concept and design of these languages are completely different from the pipeline idea of most software programming languages. Figure 12 below provides a clear summary of the differences between the two language families. After designing and coding, HDLs must go through extra processes before implementation onto an FPGA. First, during the synthesis process, the code is translated into a schematic circuit, called a netlist. After synthesis, the HDL compiler will check if the design specified functions correctly in the simulation process. The netlist is then translated into binary form during the translation phase. Next, the mapping process defines the netlist connections and components into CLBs and then the device is modified to fit onto the target FPGA during the place and route phase [31].

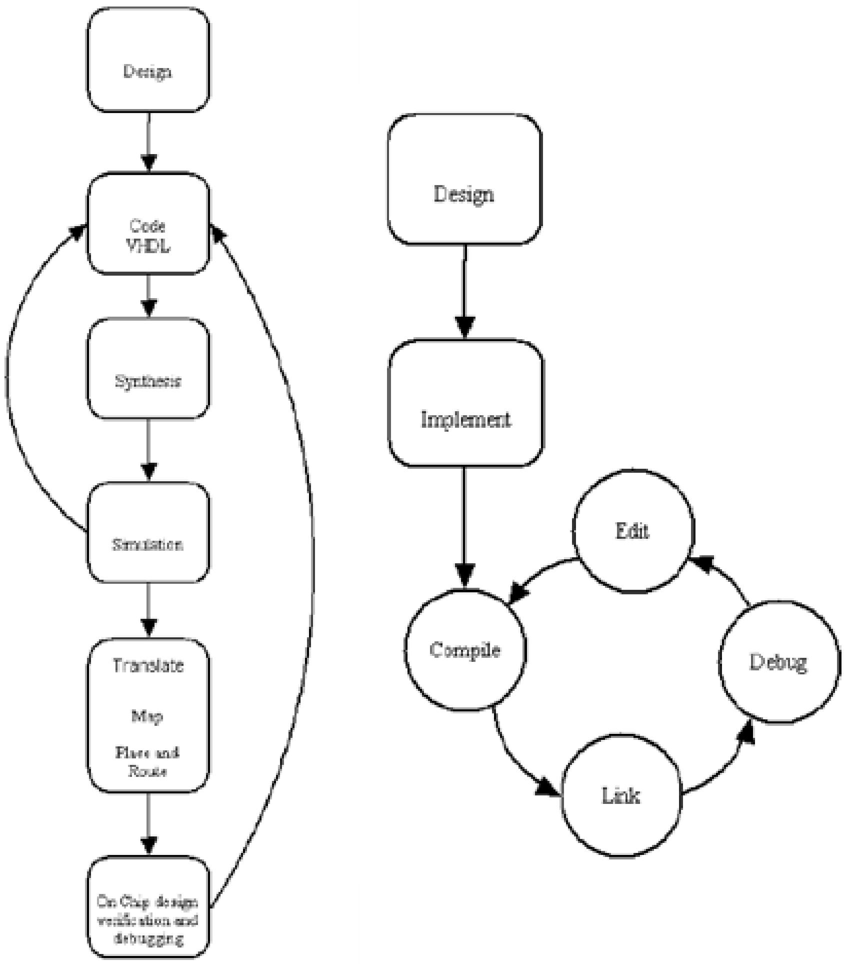


Figure 12: Design Flow- Hardware (left) Software (Right)

2.2.4.1 VHDL

VHDL is one of the two traditional HDLs for FPGA programming. VHDL stands for VHSIC (Very-High-Speed-Integrated-Circuit) Hardware Design Language. VHDL was originally developed by the U.S. DoD in order to keep track of the behavior of ASIC equipment. In 1986, VHDL was transferred to the IEEE for regulation. From that time onward the IEEE standard for VHDL has been revised every five years [32]. This language is still largely used in government-related projects, but is a powerful tool in VHDL development.

As an HDL, VHDL is used to describe electronic hardware at many levels of abstraction. Figure 13 below shows the summary of level of abstractions of some common languages [31]. From the figure, one can see that VHDL supports both low level abstraction, such as Register Transfer Level (RTLs) and Logic Gates, and relative high level abstraction, such as design behavior.

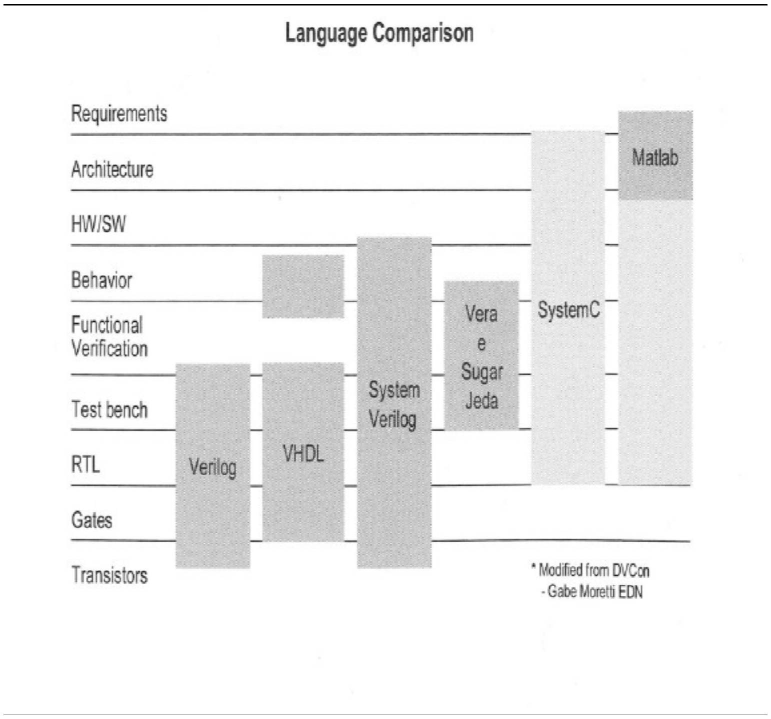


Figure 13: Level of Abstraction of different languages

According to Douglas J Smith in his report "HDL basic training: top-down chip design using Verilog and VHDL", VHDL can place procedures and functions in a package so that they are available for use by any other design. In addition, VHDL allows concurrent procedure calls. With VHDL, designers can use many languages or user defined data types, which is usually an advantage but can also be a drawback because of the need for careful conversion. VHDL also provides users with libraries, which act as storage in the host environment for entities, architectures, packages, and configurations. One can use libraries to manage multiple projects as well [33]. VHDL today is one of the two most common languages for FPGA design.

2.2.4.2 Verilog

Verilog is another highly successful HDL. According to “A brief history of Verilog” from Doulos Ltd., Verilog is rooted from a logic simulator, called Verilog-XL, which was created by Gateway Design Automation Company. In 1990, the language was transferred to public domain with the intention of turning Verilog into a standard HDL. Then in 1995, Verilog was introduced to the IEEE for standardization and, like VHDL, has since been revised every five years [34]. Recently, Accelera, a nonprofit organization, which was formed to maintain Verilog with the IEEE, has introduced an extended version of Verilog, named Systemverilog. As can be seen in Figure 13, Verilog can only support low level abstraction design like RTLs and Logic Gates [31]. However, Systemverilog provides a wider range than VHDL which allows behavioral and functional verification levels.

Verilog is mostly used in commercial applications, unlike VHDL, which is mostly used in the government. This is likely because it was based on the C programming language, which is widely used, whereas VHDL was based on Ada, a military programming language. In addition, Verilog data types are simpler and easier to use. Verilog clearly defines net data types, such as wires and register data types. Secondly, while VHDL is a strong type language, meaning the types of variables are emphasized, Verilog is a PL/I language, which is block oriented. PLI serves as an interface between Verilog models and other languages or Verilog software tools. Both Verilog and VHDL have their place within the FPGA design community. While VHDL is common in government contracts, Verilog is widely applied in the commercial world [33].

2.2.4.3 High Level Languages for FPGA Programming

Due to the parallel nature of HDL, (in contrast to the pipeline tradition of software languages) computer science engineers often find it difficult to implement designs on FPGAs. This limitation can confine FPGA development and productivity. In addition, the need for a solid link between software and hardware increases as FPGA technology grows. Various attempts have been made to achieve this goal; consequently, Electronic System Level (ESL) tools have been developed. Most noticeable successes are Systemverilog and SystemC. Figure 13 above shows that both languages can support a relatively high level of abstraction, such as behavioral and functional verification. SystemC specifically provides the designer with options between hardware/software or architecture level descriptions.

SystemC is a C++ library used for supporting system level modeling. Its libraries are designed to work with C++ to understand the concept of time, delay and concurrent processes. SystemC can also understand signals, ports and hardware data types like bit and fixed point. Along with SystemC, Systemverilog is an HDL with extensive verification capabilities. With Systemverilog, designers can experience more high level options such as design description, functional simulation, and property specification [35].

One additional high level software language for designing FPGAs is named Impulse C. Impulse C and Impulse CoDeveloper tools are used to bring the familiar C language to FPGA based programmable platforms [36]. This software allows computer science engineers and hardware engineers to describe and accelerate parallel algorithms using standard C language. Software programmers can use the automated compiler features of Impulse C and its stream oriented programming methods to design applications without giving low level hardware descriptions. Figure 14 below provides a programming flow for hardware design using Impulse

C [36]. As stated by its developer, Impulse C is not meant to replace HDL for logic design purposes but rather for expressing C algorithms in an FPGA.

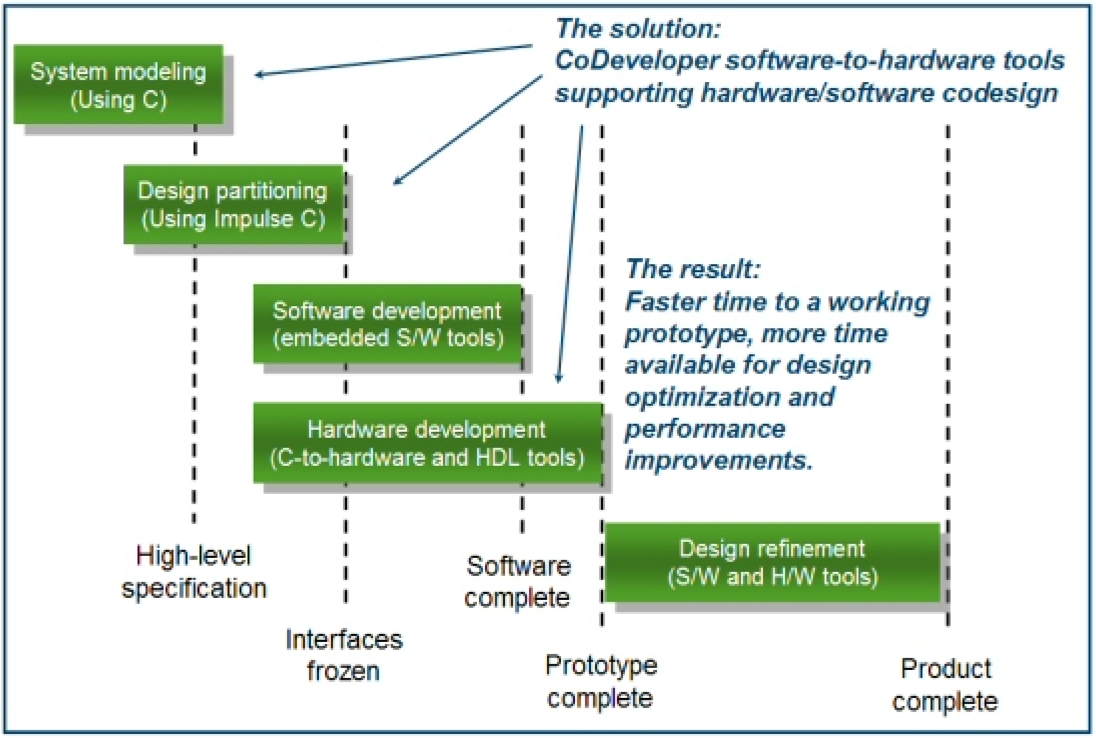

The solution:
CoDeveloper software-to-hardware tools supporting hardware/software codesign
System modeling (Using C)
Design partitioning (Using Impulse C)
Software development (embedded S/W tools)
Hardware development (C-to-hardware and HDL tools)
The result:
Faster time to a working prototype, more time available for design optimization and performance improvements.
Design refinement (S/W and H/W tools)
High-level specification
Interfaces frozen
Software complete
Prototype complete
Product complete

Figure 14: Hardware designing with Impulse C

FPGAs can be difficult to program due to their parallel nature. Several languages, from VHDL and Verilog to higher level languages like SystemC and Impulse C have been developed to make the development process easier.

2.3 IEEE754 Floating point standard

The IEEE754 standard lays out the convention for representing floating point numbers. Before this standard was developed, each platform developed its own method for utilizing floating point numbers. In the following sections we describe the standard in detail, including its precise representation in a binary computer. We then outline some special cases and important considerations when using this standard. Finally, we discuss floating point math, both in general and using the IEEE standard. This information will lend us insight into the computational methods involved in implementing floating point arithmetic.

2.3.1 Importance and History of IEEE754 Floating Point Standard

Most modern computers and calculators have a high degree of accuracy and speed. Users expect consistent and accurate answers for every calculation made on every system. The IEEE754 floating point standard is credited for maintaining consistency in computing today. One can only imagine how disastrous computations would be without standards. Before 1985 different microprocessors would provide different mathematical results. Even with today's set of standards, one minor floating point standard mistake can cause considerable damage. For example, in June 1996 a satellite rocket named Ariane 5 worth over half of a billion dollars crashed into the ground. According to Professor William Kahan, the programming language that designed Ariane 5 ignored the default exception handling specifications in IEEE754. As a result, the control system misinterpreted corrective actions and the rocket's motors exceeded their limits of mounting [37]. Had the overflow standards been followed, the software would have set a flag, delivered an invalid result, which then would have been ignored.

Before the IEEE754 standard, each manufacturer devised their own floating point rules and thus, had their own range and accuracy. Confusion occurred often from one microprocessor to another. For example, one computer can treat a value as non zero for addition and comparisons but as a zero for division and multiplication [37]. The computers, languages, compilers, and algorithms were so varied that simple joint projects would have been nearly impossible.

In 1976, Intel started its own internal arithmetic standard and hired Professor Kahan as a consultant [37]. With Intel, Professor Kahan integrated the best available floating point methods into the i8087' ROM. As the result, in 1985, the IEEE754 standard for Binary Floating Point Arithmetic was finally developed after a decade long effort [37]. Like all other IEEE standards, IEEE754 also needed revision after a certain amount of time. Fifteen years was chosen for IEEE754. This standard has been especially important with the dramatic improvements in microprocessors in recent years. Although the standards have been a great success, it is important to understand that the policies do not provide the "right" answer but rather a consistent one.

2.3.2 Definition and Standard Convention

The following sections describe the IEEE floating point standard in detail. This includes the general representation convention and some special cases that need to be considered. We then discuss issues with floating point operations and mathematical methods.

2.3.2.1 Definition

This section provides a specific definition and explanation for the IEEE754 floating point standard. There are several ways to represent real numbers on computers. Fixed point format places a radix point somewhere in the middle of the digits and is equivalent to using integers to express portions of some unit. A rational represents any value as a ratio of two numbers. Floating point, the most common representation, represents real numbers in scientific notation, which describes a value as a base number and an exponent. According to Steve Hollasch, since fixed point format has a limited window of representation, it cannot express a very high or very small value. Moreover, fixed point format loses precision while doing division. Floating point format, on the other hand, has a wider window of representation and loses less precision during division and thus has become more common [38].

After multiple revisions, the IEEE separated floating point into four main categories, single precision (32 bits), single extended precision (minimum 44 bits, including one hidden bit), double precision (64 bits), and double extended precision (minimum 80 bits, including one hidden bit) [39]. Besides these 4 main categories, modern microprocessors also support some advanced floating point expressions, such as single precision and double precision for complex numbers. Extended formats were introduced for extra precision and exponent range.

IEEE floating point numbers have three main components: the sign, the exponent, and the mantissa, which is also known as a significand.

- The sign bit is always the most significant bit (MSB), or the furthest left bit and describes positive numbers with a 0 and negative with a 1.
- The exponent needs to represent both negative and positive numbers. IEEE754 uses a biased representation to express the exponent of a floating point number. In a case of a single precision number, the exponent has 8 bits, thus the bias is 127 (or 16 bits with a bias of 1023 for double precision). Thus, an exponent of 0 would be described as 127 in the exponent field. A stored value of 300 would indicate 173, because 127 subtracted from 300 results in 173. Two special cases are -127 and +128 which are reserved for special numbers which will be discussed later on.

- The mantissa represents the precision bits of the number. The mantissa consists of a hidden bit and the fraction bits.

It is important to note that the base of the exponent in IEEE754 is always two. Typically during floating point arithmetic, shifting is necessary; however, the performance gained from using a larger base (which would require less shifting) is relatively small. There is therefore little gain in using a higher base. Another standardization decision of note is the hidden bit. Since binary numbers can always be represented starting with a 1 (e.g. 0100 is the same as 100), the expression can be normalized, removing any preceding zeros. Thus, we can assume there is a 1 and save one bit of storage. That format is said to use a hidden bit [39]. Overall IEEE754 states that single precision uses one sign bit, eight exponent bits and 23 bits for the significand. However, when the hidden bit is included, the significand is actually 24 bits, even though it is only encoded as 23.

Table 5 below displays a summary of floating point representation [39]. Note that even though single extended and double extended format only show a minimum 43 or 79 bits, they actually represent 44 and 80 bits, respectively, because of the hidden bit format.

Parameter	Format			
	Single	Single extended	Double	Double-extended
Mantissa , p	24	≥ 32	53	≥ 64
Max unbiased exponent, θmax	+127	≥1023	+1023	>16383
Min unbiased exponent, θmin	-126	≤ -1022	-1022	≤ -16382
Exponent width (bits)	8	≤ 11	11	≥ 15
Format width (bits)	32	≥ 43	64	≥ 79

Table 5: Summary of Floating Point Representation

2.3.2.2 Normalized and Denormalized Numbers

While discussing the mantissa in the previous section, the definition of binary normalized floating point numbers was introduced. Essentially, if the leading digit of the significand is nonzero, then the representation is normalized. Denormalized numbers, however, can also be used. Denormalized numbers were introduced to the floating point standard to fill in the gap between 0 and the smallest value a normalized floating point number can express. Using the binary single precision floating point representation, the smallest normalized value would be 2^{-126}. Denormalized numbers play an important role when dealing with extremely small calculation results. Essentially, using normalized numbers, any real number smaller than 2^{-126} would "flush into 0", which would cause large relative error for normalized number calculations [39]. IEEE754 floating point standards define denormalized numbers as the following: when the exponent bits are all 0s, but the fraction and the leading bit of the mantissa are non-zero, then the value is a denormalized number [38].

Since binary floating point standards always assume that the hidden bit is 1, a floating point number with an exponent of θ_{min} will always have a mantissa that is greater than 1. Therefore, for consistency, the IEEE chose to keep the same exponent value (in the representation) for any floating point numbers with the exponent that is larger than $\theta_{min} - 1$. In the event that the exponent is equal to θ_{min}-1, the denormalized numbers, which are smaller than 1 but larger than 0, will be represented as $0.b_1b_2b_3b_4b_5b_6$ x $2^{\theta min+1}$. The additional one is needed because the denormalized numbers have an exponent of θ_{min} not $\theta_{min} - 1$.

A detailed example of this concept is the representation of 0.75 in binary floating point. In fixed point binary, this number would be represented as 0.11. In order to represent this as a floating point, we would need to shift the decimal point to the right, which results in $1.1x2^{-1}$. Using the rules we have learned so far from IEEE754, we would then represent this number in the following way. The mantissa, in this case would need to be 10..0 and the exponent would need to be -1 (or -128 after the bias); however -128 cannot be represented using 8 bits. Therefore, we need a new method of representation. This is where the concept of denormalized numbers is important, which applies the range of numbers from -1 (non-inclusive) to near zero on the negative side and from near zero on the positive side to 1 (non-inclusive), or mathematically: (-1,0) and (0,1). This representation however limits precision for very small numbers near zero. In our example, 0.75 would be represented: exponent = 00..00, mantissa = 10..00. Since the exponent is 00..00, it is recognized as a denormalized number and so only the mantissa is looked at, which in this case is 10..00, or with the hidden bit included and decimal point added: 1.100..00.

Figure 15 illustrates the range of denormalized numbers [39]. The figure also introduces the concept of gradual underflow. When the result of a calculation becomes too small for normalized numbers and is expressed as denormalized, the behavior is called "gradual underflow". On the other hand, if that result is simply rounded to 0, the behavior is called "flush to zero".

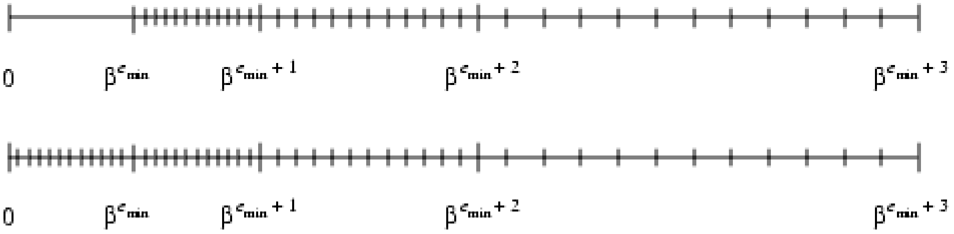


Figure 15: Flush to Zero (above) and Gradual Underflow (below)

2.3.2.3 Precision and Range of Floating Point Numbers

Although IEEE754 is an adequate representation for real numbers on computers, it still has limited precision, depending on the number of bits used. There are two reasons that floating point is unable to express any real number. First, some real numbers are too large or too small to be represented with the number of bits available. In addition, there are real numbers that can be expressed in a finite representation in decimal but have infinite repeating bits in binary. A simple example would be 0.1, whose expression always lies between two floating point numbers (i.e. division by 2 will always result in a non-zero remainder). Binary form cannot exactly represent this value [39].

The range of positive floating point numbers can be split into normalized numbers and denormalized numbers. Table 6 provides the range of representation for single and double precision floating point [38].

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(2-2^{-23})* 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23})* 2^{127}$	$\pm\sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(2-2^{-52})* 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-23})* 2^{-1023}$	$\pm\sim 10^{-323.3}$ to $\sim 10^{308.3}$

Table 6: Range of Values for Floating Point Numbers

Any real numbers that are outside of this range are considered special cases. Specifically, there are five distinct numerical ranges that double precision floating point numbers are unable to represent:

- Negative numbers that are less than $-(2-2^{-23}) \times 2^{1023}$ – negative overflow
- Negative numbers that are greater than -2^{-1074} – negative underflow
- Zero – However, IEEE754 standards reserve special representations for zero
- Positive numbers that are less than 2^{-1074} – positive underflow
- Positive numbers that are greater than $(2-2^{-23}) \times 2^{1023}$ positive overflow.

Overflow means that the absolute values of the real numbers have grown too large to represent. Underflow occurs when the absolute values of the real numbers become too close to zero. Often, underflow is treated as a less serious problem because it just results in accuracy loss.

2.3.2.4 Special Values

IEEE754 standards reserve some special representation to handle special real numbers such as zero, infinity, and not a number.

- Zero cannot be represented because of the assumption that the leading bit of the mantissa is always 1. Thus, zero is implemented with representations that have an exponent field of zero and a fraction of zero. As the result, both representations 1 0000…00000…00 and 0 0000… 00000…00 indicate 0^- and 0^+ (where the leading 1 or 0 is the sign bit), are equal to 0. Although it is not intuitive to separate negative and positive zero, both representations are necessary for some special operations discussed later.
- Much like zero, the IEEE754 standard reserves two representations for infinity. Infinity is expressed with an exponent of all 1s and the mantissa of all 0s. The sign bit distinguishes between positive and negative infinity. The reason IEEE policies decided to express infinity was to enable operations to continue past overflow situations and to allow operations with infinity to be carried out in floating point representation [38].
- The value NaN (not a number) is used to express a value that is not a real number. NaN is described with an exponent of all 1s and a non zero mantissa. There are two types of NaN: QNaN (Quiet NaN) and SNaN (Signaling NaN). A QNaN is an NaN with the MSB of the mantissa set. QNaNs propagate freely through floating point arithmetic operations. QNaNs appear when a calculation is not mathematically defined (i.e.

indeterminate operations). On the other hand, SNaNs are NaNs that have MSB of the mantissa cleared. SNaNs are used to for exceptions and uninitialized variables to trap premature usage or invalid operations [38].

Table 7 below provides a summary of all the values that floating point representation can describe.

Sign	Exponent	Fraction	Value
0	00…00	00…00	+0
0	00…01 to 11..10	XXX	Positive number
0	11…11	00…00	+Infinity
0	11…11	0XXX	QNaN
0	11…11	1XXX	SNaN
1	00…00	00…00	-0
1	00…00 to 11…10	XXX	Negative number
1	11…11	00…00	-Infinity
1	11…11	0XXX	SNaN
1	11…11	1XXX	QNaN

Table 7: Summary of Floating Point Representation.

2.3.2.5. Special Operations

Special operations are operations that deal with special numbers or mathematically undefined calculations. The simplest type would be operations with an NaN, which will result in another NaN. Table 8 below provides definitions to other special operations [38].

Operation	Result
N / Infinity	0
± Infinity * ± Infinity	± Infinity
± non zero / 0	± Infinity
Infinity + Infinity	Infinity
±0 * ±0	NaN
Infinity – Infinity	NaN
±Infinity / ±Infinity	NaN
±Infinity * 0	NaN
√(negative number)	NaN

Table 8: Special Operations

2.3.2.6 IEEE Exceptions

Special operations (according to the IEEE754 standard) will trigger an exception flag. Users are given permission to both read and write the status flag. The status flag will not change unless it is modified by the user [39]. Depending on the situation, the program may carry on after providing a result. However, it is strongly advised that exceptions should be handled by trap handlers. Trap handlers will be discussed in the following section.

IEEE standards divide exceptions into 5 categories: overflow, underflow, division by 0, invalid operation, and inexact. The definitions of overflow and underflow were already provided. Division by zero results when the program is forced to divide by zero, or a small number that has been rounded to zero. Table 9 lists most of the situations that result in exceptions (x indicates the exact result of an operation while α = 192 for single precision and 1536 for double precision). The definition of invalid operations and the inexact case are also provided. In Table 9, rounding is defined as changing the result of operation to an appropriate number [39].

Exception	Result when trap handler is disabled	Result when trap handler is enabled
Overflow	±∞, ± xmax	Round($x2^{-\alpha}$)
Underflow	0, $\pm 2^{emin}$ or denormalized numbers	Round($x2^{\alpha}$)
Divide by 0	±∞	Operands
Invalid	NaN	Operands
Inexact	Round(x)	Round(x)

Table 9: Invalid Operations

2.3.2.7 IEEE Trap Handler

As mentioned above, trap handlers are used to help a program carry on after an exception occurs. For example, if a loop requires making some comparisons but an exception results in NaN, without trap handlers, the code could result in an infinite loop, because a comparison of NaN to a number always returns a false [39].

Trap handlers are proven to be very effective in dealing with overflow or underflow in order to maintain the calculation's precision. IEEE standards introduce a global counter initialized to zero. When an overflow occurs, the trap handlers increment the counter and return a wrapped around exponent value. Similarly, if an underflow occurs, the trap handlers will decrement the counter. The user can then test the counter after the calculations are completed to determine if an underflow or overflow error occurred. IEEE754 standard defines the wrap around process for overflow and underflow as described in Table 9.

2.3.2.8 Rounding Modes

Rounding modes are used whenever the result of a calculation is not exact. Each operation is computed as precisely as possible and then rounded. There are 4 types of rounding: round to the nearest value, round toward 0, round toward +∞, and round toward -∞. Rounding toward -∞ is called a floor function and rounding toward +∞ is called a ceiling function. Note that rounding a positive magnitude overflow toward +∞ would not result in +∞ but the greatest positive representable value. Similarly, rounding a negative magnitude overflow toward -∞ or toward 0 would result in the greatest negative representable number [39].

Rounding modes are used often in interval arithmetic. Normally, interval operation would require inputs as intervals and the expected result in interval form. If two intervals are added or multiplied, the result would be an interval with the left limit of the result rounded toward -∞ and the right limit of the result rounded toward +∞ [39]. Interval arithmetic usually requires calculations in many precision levels because the limits become too large [39].

2.3.2.9 Flags

Flags are introduced by IEEE standards for users to keep track of special circumstances. Each special circumstance should be implemented with a flag for monitoring purposes. There is one status flag for the five exceptions listed above. There is also a flag for the four rounding modes. Flags should be designed for users to both read and write. They only change based on user modification.

2.3.3 Floating point arithmetic

This section will provide an explanation of basic floating point arithmetic. Specifically, we will discuss the operations of addition, subtraction, multiplication, and division. In addition, methods to perform transcendental functions will be briefly discussed in this section. Note that the methods represented are basic concepts in performing floating point calculations for FPGAs and microprocessors. Most floating point calculations are performed similarly to those used for signed magnitude integers [40]. Thus, in each following subsection, the format will be in the order: methodology, decimal example and corresponding floating point example.

2.3.3.1 Addition

In decimal addition, two values need to have an aligned decimal point before the operation can be carried out. The normalization process follows afterwards, if necessary [40].

Example: Decimal format: $3.25 \times 10^{3} + 2.63 \times 10^{-1}$

Step 1: 3.250000×10^{3}
0.000263×10^{3}

Addition: $= \; 3.250263 \times 10^{3}$ (This number is already normalized)

In floating point, we know that:

$3.25 \times 10^{3} = 0$ 10001010 10010110010000000000000
$2.63 \times 10^{-1} = 0$ 01111101 00001101010011111110000

During the floating point aligning process, every time a left shift occurs, the exponent needs to be decremented once. Similarly, a right shift would result in an increment. Although the hidden bit is not displayed, it must be taken into account. Therefore, the left shift should be avoided as it will eliminate the hidden bit, which would result in inaccurate result. Normally, during the floating point aligning process, the right shift is applied.

First, we subtract the exponents to determine the number of shifts required. Since 10001010 – 01111101 = 1101, 13 right shifts are required. Also when the shifting occurs, the hidden bit must be taken into account. Thus, the shifting process for 2.63e-1 (i.e. 2.63 X 10^{-1}) will be as follows:

```
0   01111101   00001101010011111110000 (Original value)
0   01111110   10000110101001111111000  (First shift)
0   01111111   01000011010100111111100  (Second shift)
⋮
0   10001010   00000000000100001101010 (13th shift)
```

Now that the preparation process is completed, we can add the mantissas:

```
     01000101010010110010000000000000
   +
     010001010 00000000000100001101010


 =   010001010100101100101000011010100 , which is 3250.263
```

Note that if the sum overflows, the position of the hidden bit and the mantissa need to be shifted one bit to the right and the exponent must be incremented.

2.3.3.2 Subtraction

The subtraction process is similar to addition. Once again, the aligning process is first. As mentioned in the addition subsection, during the shifting process, the exponent needs to be adjusted accordingly. Once all the preparations are finished, subtraction can be carried out the same way as unsigned binary subtraction. Subtraction finishes with the normalization process if necessary.

The subtraction process requires a few more steps than addition. Magnitude comparison is required to adjust the sign bit. Note that there is a hidden bit that is not displayed in the mantissa, which needs to be taken into account.

Example: Decimal format: $22.5 \times 10^0 - 5.23 \times 10^1 = -29.8$

Floating point format:

$22.5 \times 10^0 = 0$ 10000011 01101000000000000000000 (Value 1)
$-5.23 \times 10^1 = 1$ 10000100 10100010011001100110011 (Value 2)

Again, the aligning process is required in this case. Since 10000100 – 10000011 = 1, only 1 right shift for (22.5) is required. Note that the second operand is greater than the first one, thus the result will be negative.

Shifting process:

```
1   10000011   01101000000000000000000    (Original value of Value 1)
1   10000100   10110100000000000000000    (First right shit, hidden bit is displayed)
```

Subtraction process:

```
 1   10000100       10100010011001100110011
-0   10000100       10110100000000000000000

=        1   10000100      0.01101110011001100110011
```

The result is not in normalized format. Thus, the next step would be normalization, which can be done easily by a left shift. A left shift would bring the hidden bit back to 1 and decrease the exponent by 1. The shifting process for the result will be as follows:

Normalization process:	1	10000100	0.01101110011001100110011	(*Original value*)
	1	10000011	1.11011100110011001100110	(*Left shift once*)

Finally, the result is 1 10000011 11011100110011001100110, which is -29.79999 ≈ -29.78.

2.3.3.3Multiplication

The concept of floating point multiplication is fairly straightforward. There are three major steps that need to be remembered. The first is adding the exponents. The second is completing an "XOR" of sign bits and finally multiplication like unsigned binary multiplication for the mantissa. Since the exponents are handled separately in the multiplication process, aligning is not required. Specifically, the biased exponents will be added, then subtracted by 127. The subtraction can be performed by adding the two's complement of -127, which is 1000 0001.

For the mantissa multiplication, the decimal point of the result is positioned so that the number of decimal places equals the sum of the number of decimal places in the operands. Again, the hidden bit "1" needs to be taken into account. Note that the product of two 24 bit mantissas would result in a 48 bit number. Also since both operands are already in normalized form, $1.b_1b_2b_3b_4 \ldots b_n$, the product should only have two bits before the decimal. The hidden bit of the result then would be the LSB of that part. In order to obtain the 24 bit result (including the hidden bit), a truncation process is required to omit the last 24 right bits. For better explanation, an example is provided below.

Example: Decimal format: $18 \times 9.5 = 171$

Floating point format:

$$\begin{array}{l}18\ \ = 0 \quad 10000011 \quad (1).001\,0000\,0000\,0000\,0000\,0000 \\ 9.5 = 0 \quad 10000010 \quad (1).0011000\,0000\,0000\,0000\,0000\end{array}$$

Addition process for the exponent:

$$\begin{array}{ll} \ \ 1000\,0011 & (3) \\ +\,1000\,0010 & (4) \\ =\ 0000\,0101 & \\ +\,1000\,0001 & (-127) \\ =\ 1000\,0110 & (7) \end{array}$$

Multiplication process for the mantissa:

$$\text{x}\begin{array}{ll}(1).001\,0000\,0000\,0000\,0000\,0000 & (24\ bits) \\ (1).0011000\,0000\,0000\,0000\,0000 & (24\ bits)\end{array}$$

$$= \qquad 01.0101\,01100\,0000 \ldots .\ 0000 \qquad (48\ \text{bits})$$

The hidden bit of the product will be 1 which is the LSB of the non-fractional part. The mantissa of the result would be the first 24 bits counting from the decimal point to the right. Hence, the result of the mantissa multiplication would be (1).010 1011 0000 0000 0000 0000. Since this result is already in normalized form, no additional steps are required. However, if the position of the hidden bit overflows, the mantissa would need to be shifted right and the exponent would be incremented. Since both of the operands are positive, the result will be positive. Finally, the result obtained is: 0 10000110 01010110000000000000000.

2.3.3.4 Division

Unlike other operations, there are two methods to calculate division in floating point: iterative approximation and true division. Iterative approximation means first guessing the reciprocal of the divisor and then doing a multiplication of the dividend with this reciprocal. In other words, iterative approximation converts division into multiplication. On the other hand, true floating point division can be performed through the following steps:

1. Divide the dividend mantissa by the divisor mantissa as unsigned binary division. In order to obtain a 24 bit result, the dividend mantissa needs to have 48 bits, which can be done by adding "0" to the right of its LSB bit.
2. Subtract the biased exponent of the dividend from the biased exponent of the divisor. Like the floating point multiplication, the subtraction process can be done by adding the two's complement of the exponent of the divisor to the exponent of the dividend, then adding 127.
3. Normalize the result.
4. Adjust the correct sign bit.

Example: Decimal form: 121.03/9.1 = 13.3

Floating point format (true division method):

121.03 = 0   10000101   (1).11100100000111101011100
9.1   = 0   10000010   (1).00100011001100110011010

Subtraction process for the exponent

  1000 0101   (6)
+ 01111110   (*Twos Complement of* 3)
= 0000 0011   (3)
+ 01111111   (+127)
= 1000 0010   (+130)

Division process for the mantissa: the mantissa of 121.03 will add 24 more "0s" to the right of the LSB of the mantissa. Then, the unsigned division can be carried out. Since the progress of the division is time consuming, this report will skip the details and just provide the answer in hex format, 1.A99998, which is 0001. 1010 1001 1001 1001 1001 1000. Thus, the result of the mantissa is1.1010 1001 1001 1001 1001 100, which is 13.2916. However, the correct mantissa of 13.3 is 1.1010 1001 1001 1001 1001 101. Some accuracy was lost during the calculations. Whether or not this loss is acceptable depends on the application. The final answer of this example is 0 10000010 10101001100110011001100, which is 13.2916 in floating point format.

Sometimes, the mantissa division can encounter some problems such as the mantissa of the divisor has all 0s even though the floating point representation indicates a nonzero number. For such circumstances, as mentioned in section 2.3.3.7, trap handlers should take over and provide the appropriate result. Below is one special example that requires a division by zero exception.

Example: Decimal form: 56/-16 = -3.5

Floating point format (true division method):

56 = 0 10000100 11000000000000000000000
-16 = 1 10000011 00000000000000000000000

Subtraction process for the exponent:

$$\begin{array}{ll} & 1000\,0100 \quad (5) \\ + & 01111101 \quad (\mathit{two\ C\ of\ biased}\ \exp\mathit{onent}:4) \\ = & 0000\,0001 \\ + & 01111111 \quad (+128) \\ = & 1000\,0000 \quad (1) \end{array}$$

Division process for the mantissa: Since the mantissa of -16 is all 0s, the exception division by 0 should be flagged. According to section 2.3.3.7 and Table 9, the trap handler should return the value of the operand, which in this case is 11000000000000000000000. Therefore, the final result would be: 1 1000 0000 11000000000000000000000 , which is exactly -3.5.

2.3.3.5 Transcendental functions

Transcendental functions are operations that are at a higher level than the basic operations that we discussed in the previous sections. They consist of exponential, logarithmic, and trigonometric functions. Fast and accurate methods for transcendental computation are vital in many scientific computing fields. This section will provide a brief discussion of the design principles that microprocessors, particularly Intel, use to handle transcendental functions.

Transcendental functions are computed by different algorithms and look up tables. Most modern computers follow these three steps to evaluate them: reduction, approximation and reconstruction. During the reduction step, the input of the transcendental function is transformed into a small value that is confined to a known limit. With the newly found value as an input of the transcendental function, the computer will apply approximating polynomials to calculate the result. The second step, therefore, is called approximation. Lastly, during the reconstruction step, the process applies the mathematical identities that are applied in the reduction step to evaluate the real result [41]. Normally, the reduction step requires a number of sequential algorithms, while approximation and reconstruction can exploit parallelism. Thus, most computers use simple reduction methods at the price of complex approximation and reconstruction steps [41].

The following are the seven algorithms typically used to calculate some basic transcendental floating point functions provided by Intel Corporation. For simplicity, the algorithms only focus on numerical results and ignore the undefined range of these functions [41].

- Cbrt (Cube root):
  - Reduction: Given x, compute $r = x$ `frcpa`$(x) - 1$, where frcpa(x) means the reciprocal of x.
  - Approximation: $p(r) = p_1 r + p_2 r^2 + p_3 r^3 + ... + p_6 r^6 \approx (1+r)^{1/3} - 1$

  - Reconstruction: $result = T + T \times p(r)$, where $T = (\frac{1}{frcpa(x)})^{1/3}$. Normally, the value T will be evaluated from look up table.

- Exp (Exponential):
  - Reduction: Given x, compute the closest integer $N = x \times \frac{128}{\ln(2)}$ then computes $r = (x - NP1) - NP2$, where P1 + P2 approximates $\ln(2)/128$
  - Approximation: $p(r) = r + p_1 r^2 + p_2 r^3 + ... + p_4 r^5 \approx \exp(r) - 1$
  - Reconstruction: $\mathrm{Re}\,sult = T + Tp(r),$ where $T = 2^{\frac{N}{128}}$. The value T can be calculated by the following expression, $2^{\frac{N}{128}} = 2^M + 2^{\frac{K}{8}} + 2^{\frac{J}{128}}$. The first parameter is obtained by scaling and the other two parameters are found by look up table.

- Ln (Natural logarithm):
  - Reduction: Given *x*, compute $r = x$ `frcpa`*(x) – 1,* where frcpa(x) means the reciprocal of x.
  - Approximation: $p(r) = p_1 r^2 + p_2 r^3 + ... + p_5 r^6 \approx \ln(1 + r) - r$
  - Reconstruction: $\mathrm{Re}\,sult = T + r + p(r),$ where $T = \ln(\frac{1}{frcpa(x)})$. Normally, T is obtained from look up table.

- Sin and Cos

- Reduction: Given x, compute the closest integer $N = x \times \frac{16}{\pi}$ for sine function. For cosine function, $N = x \times \frac{16}{\pi} + 8$. Then compute $r = (x - NP1) - NP2,$ where P1+P2 approximates π/16.
- Approximation:
  - For sine: $p(r) = r + p_1 r^3 + ... + p_4 r^9 \approx \sin(r)$
  - For cosine: $q(r) = q_1 r^2 + q_2 r^4 + ... + q_4 r^8 \approx \cos(r) - 1$
- Reconstruction: $\mathrm{Re}\,sult = C \times p(r) + (S + S \times q(r)),$ where

  $C = \cos(N\frac{\pi}{16}) \quad and \ S = \sin(N\frac{\pi}{16})$

- Tan
  - Reduction: Given x, compute the closest integer $N = x \times \frac{2}{\pi}$. Then compute $r = (x - NP1) - NP2,$ where P1+P2 approximates π/2.
  - Approximation:
    - If N is odd, $p(r) = r + rt(p_0 + p_1 t + ... + p_{15} t^{15}) \approx \tan(r)$
    - If N is even, $q(r) = (-r)^{-1} + r(q_0 + q_1 t + ... + q_{10} t^{10}) \approx -\cot(r)$

  With $t = r^2$

  - Reconstruction: If N is even, the result is p. If N is odd, the result is q.

- Atan (arctangent):
  - Reduction: no reduction step is needed
  - Approximation:

$$If\ |x| < 1 \quad then \quad p(x) = x + x^3(p_0 + p_1 y + ... + p_{22} y^{22} \approx a\tan(x),\ where\ y = x^2$$

$$If\ |x| > 1 \quad then \quad q(x) = q_0 + q_1 y + ... + q_{22} y^{22} \approx x^{45} a\tan(1/x),\ where\ y = x^2$$

$$compute\ c^{45},\ where\ c = frcpa(x)$$

$$u(\beta) = 1 + u_1\beta + u_2\beta^2 + ... + u_{10}\beta^{10} \approx (1-\beta)^{-45},\ where\ \beta = x \times frcpa(x) - 1$$

- Reconstruction:
    - If absolute value of x is smaller than 1, then return p(x)
    - If absolute value of x is greater than 1, then
- $\mathrm{Re}\,sult = sign(x)\frac{\pi}{2} - c^{45} \times u(\beta) \times q(x)$

2.4 Summary

Financial institutions employ various techniques to determine risk and simulate market conditions. Often, the Black-Scholes model and Monte Carlo simulation are used for this purpose. Due to the computational intensity of these calculations, HPC systems are required to run simulations in a practical amount of time. Previous technology relied on large numbers of microprocessors, which have historically increased in speed over time; however, this method is becoming less practical as heat and required power are increasing operating costs. FPGA technology has the ability to do certain mathematical functions in parallel, unlike individual microprocessors; however, FPGA programming and development can be difficult, utilizing an array of different languages and techniques. This technology is currently unproven in the financial sector. This project will address the ability of FPGAs to implement financial models. The methodology of the project is discussed in detail in the next section of this report.

3.0 METHODOLOGY

The purpose of this project was to investigate using FPGAs to accelerate financial applications. In order to meet this goal, we determined three objectives. The first was to research current methods of computing in current microprocessors and FPGAs to determine the physical limitations of both platforms. This process provided the knowledge necessary to begin testing FPGAs as coprocessors. Secondly, we used benchmarks to determine quantitatively the speed and accuracy improvements in utilizing an FPGA for fundamental mathematical functions. This information was then built on in the final objective where we tested transcendental equations for speed and accuracy improvements. These data allowed us to make recommendations for continuing research and implementation of these systems. The following sections describe our methods in meeting these objectives.

3.1 Current Methods for Floating Point Calculations

Commercially available computers have been performing floating point operations for years. The first available floating point unit (FPU) was the 8087 offered by Intel to supplement the 8086, which could only simulate floating point operations in software. The 8087 FPU was automatically detected by the computer. The system would offload floating point operations to the 8087, freeing the 8086 to do further calculations [42]. Since then, FPUs like the 8087 have been integrated into existing processors.

In the following sections, we first discuss floating point arithmetic in FPGAs. This information provides a basis for understanding floating point computing in hardware. Following this discussion, we explore the FPUs in two commercially available processor microarchitectures. Specifically, we discuss the AMD Opteron (10h family) microarchitecture and Intel Core microarchitecture.

3.1.1 FPGA FPU

In this section, we discuss implementation of FPGA floating point arithmetic circuits for common operations such as addition/subtraction, multiplication, division and square root. This special application of FPGAs is called a floating point unit (FPU) and executes floating point arithmetic similarly to microprocessors.

Before getting into the specifics of floating point arithmetic, it is important to discuss a few essential topics. These include rounding and extra bits. Since there are always a limited number of bits, arithmetic operation of the mantissa can result in values that do not fit the given bits. Hence, the default rounding mode, "round to the nearest even (RNE)" will be discussed in this section. In order to implement this mode, three additional bits beyond the LSB of the mantissa are required. They are, in order from the MSB to the LSB, the guard bit (G), the round bit (R), and the sticky bit (S). The first two bits are used for normalization while the third bit is the OR of all the bits that are lower than the R bit. The rounding process adds "1" to the LSB of the mantissa. There are two special rounding cases. The first case is when G = 1, and the result of R OR S = 1 for any LSB of the mantissa. The second case is when G =1, the result of R AND S = 0 and the LSB of the mantissa is 0. Other cases would result in truncation.

For easier explanation, this section is separated into four main subsections: addition/subtraction, multiplication, division and square root.

3.1.1.1 Addition/Subtraction

An FPGA addition/subtraction unit closely follows the steps which were discussed in the addition/subtraction background section. The first step involves exponent handling. The unit calculates the absolute value difference between the two exponents and sets the greater exponent as the result's exponent. After that, the unit shifts the mantissa to the right which corresponds to the lower exponent. The number of shifts equals the difference between the two exponents.

After that, addition or subtraction will be carried out depending on the effective operation. Normalizing the mantissa result and adjusting the resulting exponent completes the operation, if it is necessary. Figure 16 below provides one example a floating point addition/subtraction circuit with its many components [43]. The circuit is designed so that one addition or subtraction can be calculated after every three clock cycles.

In the first cycle, the following tasks are covered: unpacking, determining effective operation, calculating absolute difference between the two exponents, comparing two operands A and B, and swapping the operand's mantissa if necessary. Once the circuit receives its inputs (A, B) and the operation, the unpacking block will separate the sign bit, exponent bit, and the mantissa bit. It also checks the format for zero, infinity and NaN. Then, the effective block will determine the corresponding operation for the mantissa based on the signs of the operands and the operation input. Meanwhile the exponent block will calculate the absolute difference $|E_A - E_B|$ using two cascaded adders and a multiplexer. The exponent block is also required to determine which exponent is greater. The output of the exponent block is the result's exponent which is the higher exponent of the two. As the exponent block determines which exponent is higher, it implicitly defines which operand is greater. In case both exponents are equal, a mantissa comparison is triggered. The significand comparators consist of seven 8 bit comparators that operate in parallel and an additional 8 bit comparator to process their outputs. If operand B is greater than A, a mantissa swapping process and sign adjustment are carried out. This swapping process can be done by multiplexers. Consequently, both significands are extended to 56 bits by the G, R, and S bits [43].

In the second cycle, the align block, significand add/subtract block, and leading "1" detection block are used. Due to the swapping process, the significand of B will always be shifted. This simplifies the circuit considerably since it only needs to execute alignment shifting once. The shifting process is implemented by six stage barrel shifters. Each stage can clear the bits which rotate back from the MSB to do the alignment. Each stage also calculates the OR logic of the bits that are shifted out and cleared. The output of that OR logic is called partial sticky bits. The S bit is calculated as the OR logic of these six partial sticky bits and the value that is in the sticky bit position of the output result. The barrel shifter is designed so that the 32 bit stage is followed by the 16 bit stage, and is repeated for faster speeds when calculating the sticky bit S. The output of the shifter is a 56 bit aligned significand. After this, the fast ripple carry adder computes either $S_A + S_B$ or $S_A - S_B$. Since there is a significand comparison in the first cycle, it is assured that S_A is always greater than S_B and thus the result is never negative. As discussed in the previous sections, the significand result, S_R may not in normalized form. There are four main circumstances one can encounter. The first case is when S_R is normalized. The second case is when S_R is subnormal and needs to be shifted left. The third case is when S_R is supernormal and requires right shifting. The last case occurs when all the bits of S_R are "0". For the first two cases, the leading "1" detection block can solve the problem. For the supernormal case, the leading "1" detection block is overwritten; however, this case can be easily handled by checking the carry-out of the significand adder. If $S_R = 0$, then it is treated as normalized. The leading "1" detection block can be designed as shown in Figure 17 [43].

Lastly, the third cycle completes the finishing tasks. First is the normalization process, which is handled by the alignment barrel shifter. If S_R is normalized, the data is passed through to the packing block. If S_R is subnormal, it is first right shifted and rotated to remove the three additional bits from the operand and is then left shifted until it is normalized. If S_R is supernormal, it is right shifted once, and the sticky bit is recalculated. Note that as the shifting occurs, the exponent bits are also adjusted. One adder is needed to adjust the exponent from normalization. If S_R is normalized, the exponent result, E_R is passed through the block to the packing block. If S_R is subnormal, E_R is reduced by the same number of left shifts required for normalization. If S_R is supernormal, E_R is incremented by 1, which can be done by another cascaded adder. Since there are two adders, a multiplexer is used to determine the appropriate result. In this cycle, rounding is also carried out. Rounding is performed exactly as discussed earlier. Sometimes, rounding may result in a significand with all 0s, 10.00….000. The result is reasonable and normalized since the mantissa that is actually displayed in the floating point representation is only the fraction of the mantissa with a hidden bit "1" understood but not included. Thus, the bits from MSB-1 to the LSB already show the correct values [43].

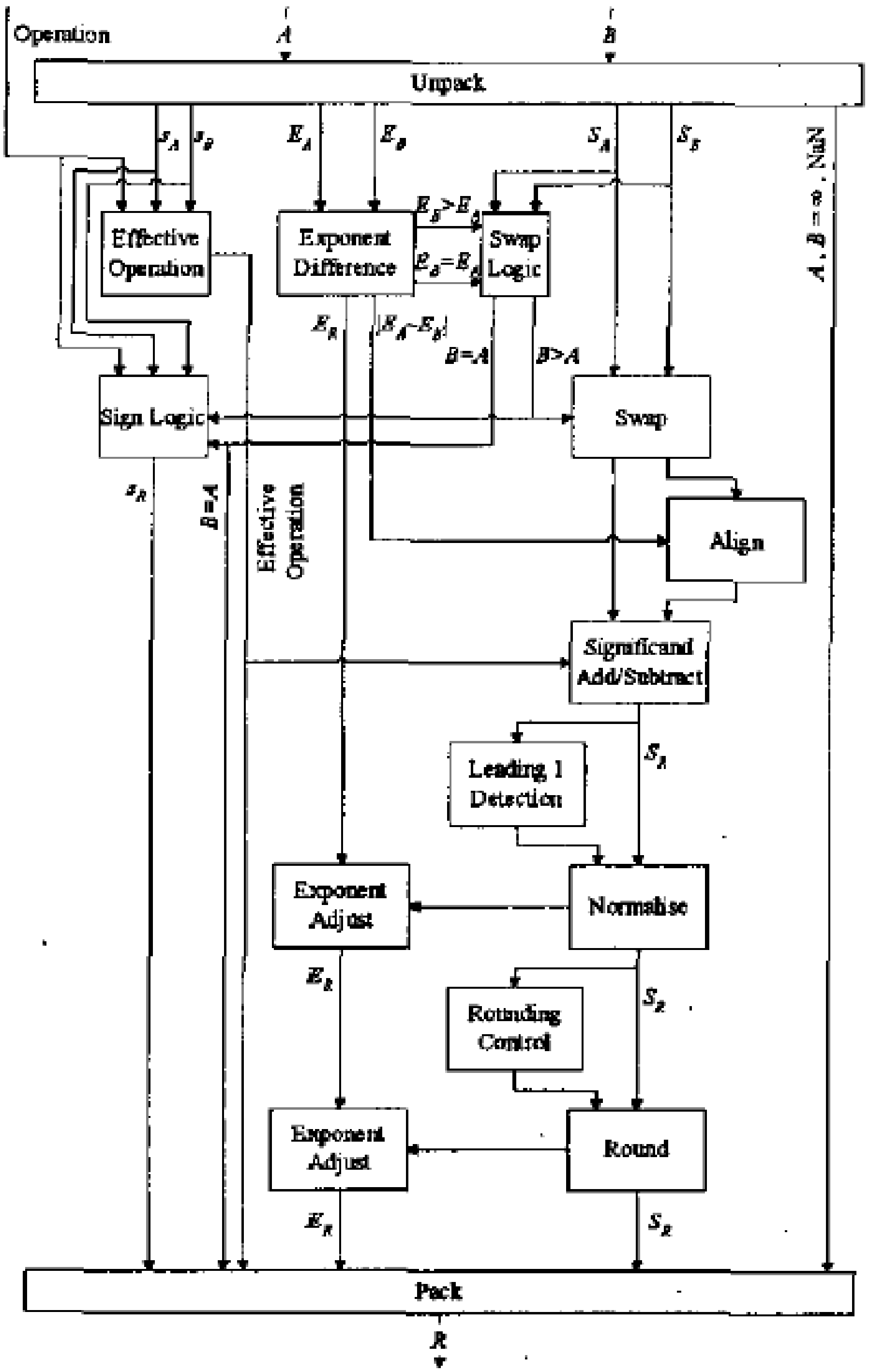


Figure 16: FPGA's Addition/Subtraction circuit

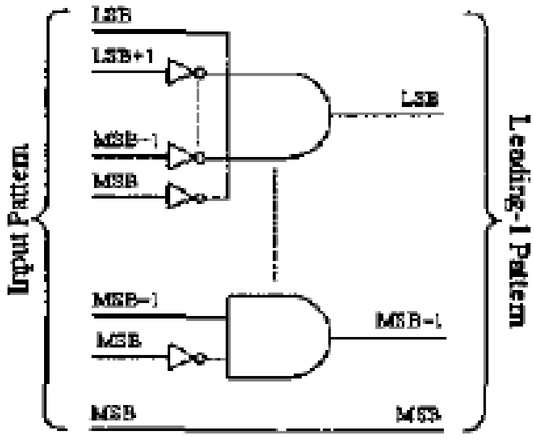


Figure 17: Leading "1" Detection block

Before packing and displaying the result, the packing block is required to do some exception checks for underflow or overflow. The block also needs to check if the result is a positive zero in the case of subtraction of two equal numbers, according to the IEEE standard. In addition, the result needs to be appropriate when dealing with infinity or NaN as discussed in the background section [43].

3.1.1.2 Multiplication

Floating point multiplication in circuitry is less complicated than addition/subtraction. The product's exponent is the result of the sum of the two operand's exponents minus the biased exponent. The product's significand is the product of two operands significands. Once the calculation is finished, normalization and rounding occur along with exponent adjustment. Unlike an adder, this circuit operates with two clocks: a primary clock on which latencies are based and an internal clock which is twice as fast. The internal clock is used to multiply the significands. This unit requires ten clock cycles to complete one multiplication. The block diagram of this unit is provided in Figure 18 below [43].

In the first cycle, both input A and B are unpacked and checked for zero, infinity, and NaN. The sign bit of the result can be determined by an XOR of the signs of A and B. Since the sign of the operation is already taken into account, the unit treats both operands as positive numbers. Finally, the circuit evaluates the product exponent E_R by a fast ripple carry adder [43].

The second cycle completes the exponent calculation by removing the excess biased exponent using the same adder from the previous cycle. The significand multiplication process is also started in this cycle. This process lasts until the eighth cycle. The multiplier is designed based on the Modified Booth's 2-bit parallel multiplier recoding method. The circuit is implemented using a serial carry save adder array and a fast carry adder for the combination of the final carry bits into the final sum bits. Specifically, the carry save adder array contains two cascaded carry save adders, which produce four sums and two carry bits after every internal clock cycle (the faster clock). A 4-bit fast carry adder then combines the carry-in from the previous combination with the four sums and the two carry bits that are produced from the adder array in the previous fast clock cycle. In addition, the circuit also needs to store the OR result of these partial combinations because they will become the OR inputs to calculate the S sticky bit [43]. Since what matters is just the OR result from these partial combination results, the circuit does not need save any partial combinations except the final one, which is the actual answer.

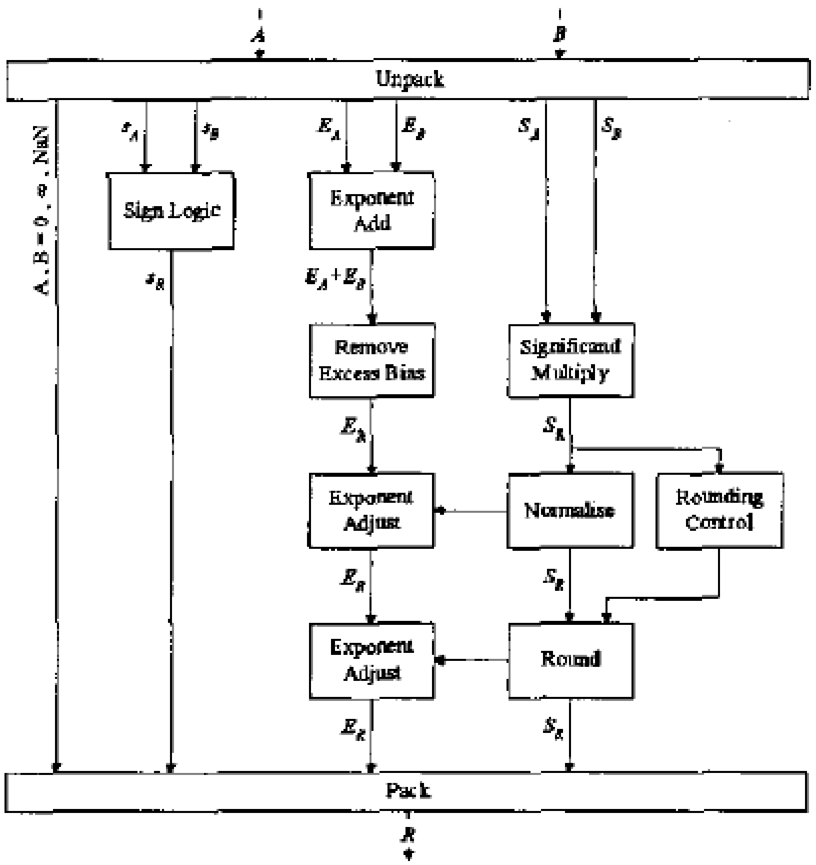


Figure 18: Multiplication unit

In the ninth cycle, the final sum and carry bits are produced. A multiplication of two 53 bit operands would result in a 106 bit result, which would be in the form: $b_1b_2.b_3b_4b_5$…..$b_{57}b_{58}$…b_{106}. Bits b_1b_2….b_{56} are obtained from the last partial combination. However, as mentioned above, bit $b_{57}b_{58}$…b_{106} are not saved. Thus, their OR results can be replaced into the vacant places from the 57th bits. Normalization is also completed in this cycle. If b1 = 0, the result is normalized and the 56 bit final significand result after rounding would simply be the $b_2.b_3b_4$….b_{57}. If b1 = 1, the result is shifted right once and then is rounded. In this case, after rounding, the result would be $b_1.b_2b_3$…b_{56}. The 56th bit would be the OR result of the original 56th bit and any bits after the 56th bit. The shifting process can be done by using some multiplexers to switch by b_1. Also, the exponent is incremented by 1 for every right shift, using the same adder that was used before during the exponent calculation. Normally, the circuit does not wait for the final result to determine if the output is supernormal. Instead, it will perform the adjustment right at the beginning of the cycle and then use multiplexers to decide the correct answer. Similarly, the rounding process also starts at the beginning of the cycle and makes the appropriate decision once b_1 is calculated [43].

In the last cycle, another rounding process is carried out. This process transforms the 56 bit result of the ninth cycle into a 53 bit answer. The rounding process is the same as mentioned earlier. Again, normalization is no longer required even when the rounding process results in a 10.000…0 because the fraction bits are already correct. However, the exponent answer needs to be incremented by 1, using the same exponent adder as before. It is also important to check for zero, infinity, and NaN as well as underflow and overflow, which are performed in this cycle as well before packing [43].

3.1.1.3 Division

The concept of division is explained in the following paragraphs. The exponent of the answer is the result given by $E_R = E_A - E_B + \textit{Biased Exponent}\ (1023)$, assuming that A is the dividend and B is the divisor. The quotient's mantissa is the result of a division between A and B. After the result is calculated, the normalization and rounding processes begin. The most time consuming process in this block is the division circuit. One calculation can require up to 60 clock cycles to complete. An example block diagram of this unit is shown in Figure 19 below [43].

Similar to the multiplication unit, in the first cycle, operands A and B are unpacked and checked for exceptions such as zero, infinity, and NaN. The sign can be determined easily by an

XOR logic gate of the sign of A and B. Also, in this state, the circuit finds out the difference between two operands' exponents, using a fast ripple carry adder [43].

In the second cycle, the fast ripple carry adder adds the biased exponent into the first cycle's exponent result, which completes the exponent calculation. The significand division process also starts in this second cycle. During the initial part of the division process, the remainder of the division is set equal to the dividend's mantissa, S_A. Then, the divisor S_B is subtracted from the remainder. If the result is either positive or zero, the MSB of the quotient S_R becomes 1 and the remainder become the newly found result. Otherwise, the MSB is 0 and the remainder is not replaced. In that case, the circuit will shift the remainder left one bit. Again, the new remainder then is subtracted from the divisor significand to calculate the MSB-1 bit of the quotient. This process continues until the quotient is filled with 55 bits. The 56th bit is a sticky bit which will be calculated separately in the 57th clock cycle. The last two (54th and 55th) bits are the G and R bit respectively. The significand unit consists of two registers for the divisor and the remainder, a fast ripple carry adder to compute subtractions, and a shift register for the result's mantissa. Every clock cycle, the unit produces one S_R bit, thus, it takes 55 clock cycles to complete one division process [43].

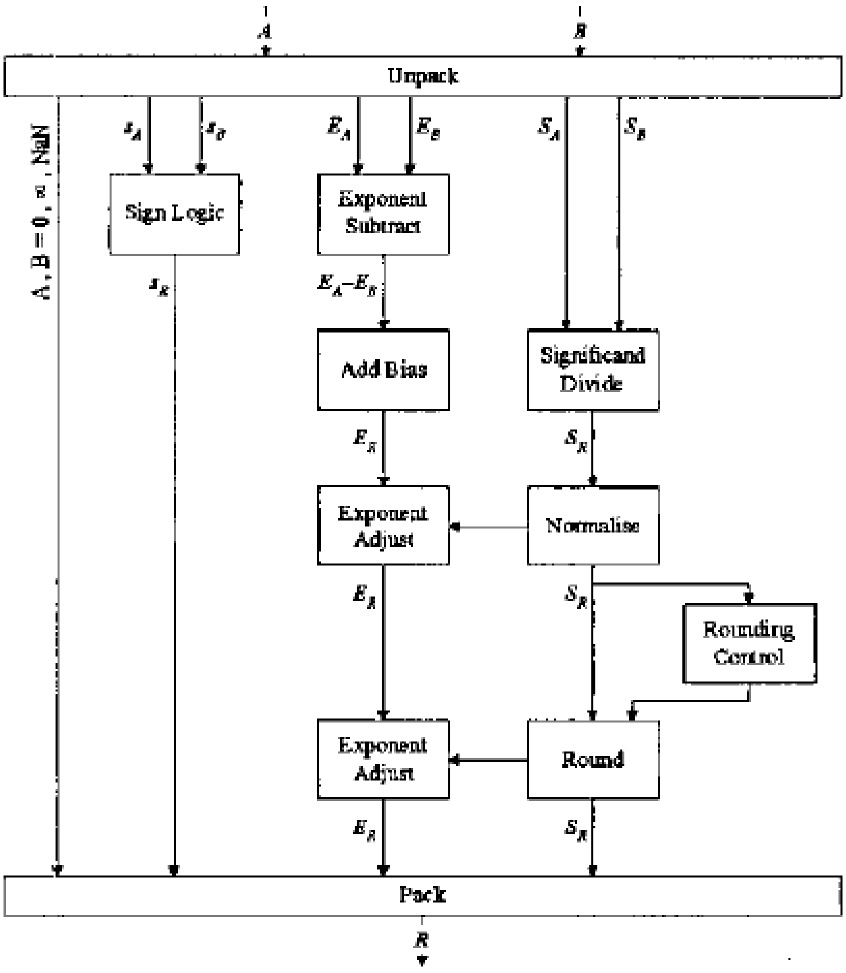


Figure 19: Division unit

As mentioned in the previous paragraph, the 56th sticky bit is calculated in the 57th clock cycle. The bit is simply a result of an OR logic of all the remaining bits in the remainder register after the division is finished. Since both A and B are in normalized form, the quotient significand can be either normalized or subnormal which requires one left shift and a decrement from the quotient's exponent. The exponent adjustment can be performed by the same adder that was used to calculate the exponent earlier. Since the quotient is already in the shift register, shifting can be done easily without extra hardware. Normalization is executed in this cycle but the exponent adjustment is performed in the next clock cycle (the 58th). During the 58th cycle, the 56 bit quotient significand is also transferred into the divisor register which is connected to the adder from the division circuit [43].

In the 59th clock cycle, the rounding process transforms a 56 bit input to a 53 bit final answer. The same instances as before (where the significand becomes 10.000…000) may occur, but, again, no normalization is required [43].

In the 60th clock cycle, the exponent adjustment (incrementing the exponent by 1) is performed in case the rounding process results in all 0s in the significand. In the last cycle, format checks are executed to check for zero, infinity, NaN, underflow, and overflow. These checks are carried out before packing to the output. The division circuit is fairly small compared to multiplication and addition/subtraction, but requires a considerable amount of time to execute [43].

3.1.1.4 Square Root

The square root operation is not as frequently used as the other three operations listed above, because most designers aim for low cost implementation, rather than low latency for this operation. Figure 20 below provides an example circuit for the square root operation. This circuit takes 59 clock cycles to complete one square root operation [43].

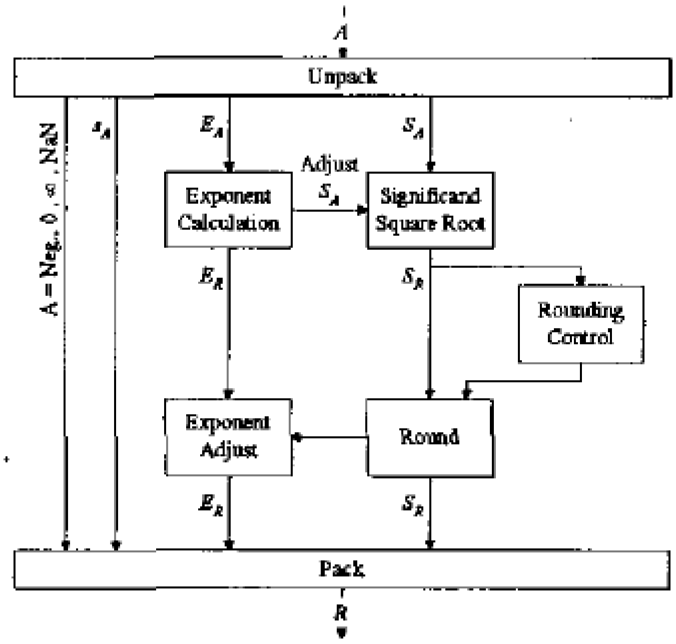


Figure 20: Square root unit

The concept for handling the square root operation requires several steps. First, the biased result's exponent is calculated by the equation below [43]:

$$E_R = \begin{cases} \dfrac{E_A + 1022}{2} & if\ E_A\ is\ even\ (and\ left\ shift\ S_A\ once) \\ \dfrac{E_A + 1023}{2} & if\ E_A\ is\ odd \end{cases}$$

For the significand calculation, the result's significand is denoted by b₁.b₂b₃b₄b₅… for explanation purposes. Each b_n can be computed from the following equations [43]:

$$X_{n+1} = \begin{cases} 2(X_n - T_n) & if\ b_n = 1 \\ 2X_n & if\ b_n = 0 \end{cases}$$

$$with\quad X_1 = \frac{S_A}{2};\ T_1 = 0.1;\ n = 1,2,3,....;\ and\ T_{n+1} = b_1.b_2b_3...b_n01$$

$$b_n = \begin{cases} 1 & if\ (X_n - T_n) \geq 0 \\ 0 & otherwise \end{cases}$$

Essentially, the equations compute the result's LSB first by finding the difference between X_1 and T_1 which is assigned as 0.1, base 2, (0.5 in decimal) initially. If the difference is either positive or zero, the LSB of the result is 1, otherwise it is 0. With the newly found bit, T_2 is updated to $b_1.01$ and X_2 is also updated using the first equation. After that, the LSB-1 bit is calculated and the process goes on until the result reaches 55 bits. The last 56th bit is the sticky bit which is calculated with an OR logic of all the bits in the final remainder of the significand square root calculation [43].

In the first cycle, the circuit does the normal unpacking and checking tasks like other operations. The exponent calculation also starts in the first cycle using a fast ripple carry adder. Division by two can be implemented simply by discarding the LSB of the numerator, which is forced to be even [43].

The significand calculation starts in the second clock cycle. Since the significand of operand A is already in normalized form, the result's significand of the square root operation will be in normalized form [44]. In a sense, the square root algorithm is similar to division. For the circuit shown in Figure 20, it takes one cycle to compute one bit of the result's significand. Thus, the significand calculation lasts until the 56th clock cycle. The main components of the significand square root unit are two registers for X_n and T_n and a fast ripple carry adder. The T_n register is designed so that each flip flop has its own enable input. This implementation helps each b_n bit to be stored in its correct position, to control the reset and to set inputs of the two next flip-flops in the register so that the correct T_n is formed after each clock cycle. The significand result will have 55 bits and will be stored in the T_n register. The last two bits of the significand result are the guard bit G and the round bit R [43].

In the 57th cycle, the sticky bit is calculated as mentioned before. In the 58th cycle, the 56 bit result is rounded to the 53 bit final answer, using the same adder that is used in the significand square root circuit. Again, no normalization needs to be done, but the exponent adjustment, incrementing the answer's exponent by 1, is required, which is performed in the 59th clock cycle, if necessary. Since the result's exponent is divided by 2, there will never be an overflow or underflow; however, a format check for negative values, zero, infinity and NaN must be carried out before packing the output.

The previous sections discussed floating point operations in FPGAs. The next two sections discuss CPU microarchitecture and how floating point operations are executed in two specific microarchitectures.

3.1.2 AMD Opteron (10h Family) Microarchitecture

AMD Opteron (10h family) microarchitecture is the newest structure for AMD's microprocessors. They are using this microarchitecture for the next generation Opteron processors and others as well. The structure of this microarchitecture is discussed in the following section in order to better understand how floating point operations differ between AMD and Intel microprocessors.

Figure 21 shows the overall microarchitecture for the AMD Opteron [45]. The design includes three levels of caches: L1 instruction, L1 data, L2, and L3. In addition, the microarchitecture uses an ALU for integer operations and an FPU for floating point operations. The FPU consists of two components: the FP Scheduler and the FP Execution Unit [45]. These components will be discussed in more detail in this section.

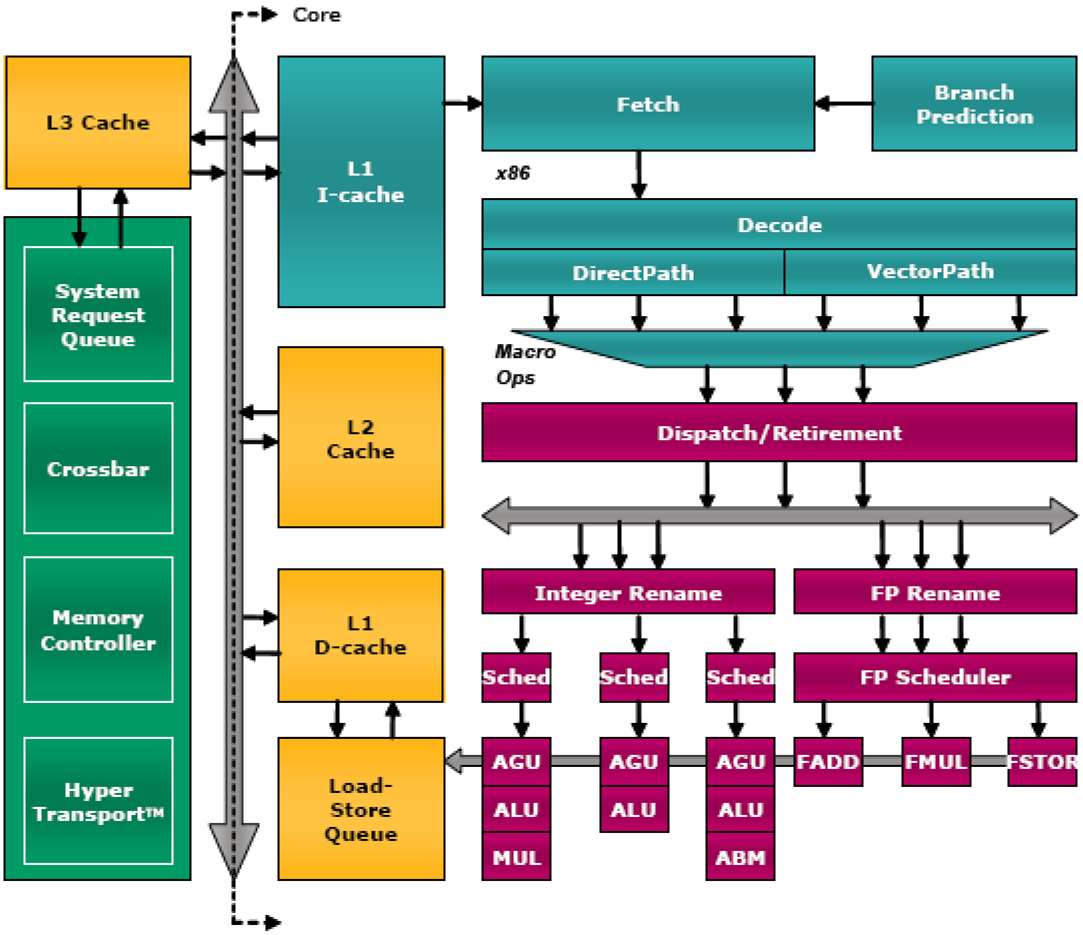


Figure 21: Block Diagram for AMD Opteron (10h Family) Processors

3.1.2.1Caches

AMD Opteron microarchitecture utilizes three levels of caches. The L3 cache is shared between all cores of the CPU and allows for fast communication between the cores. The L2 cache is the next level which holds recently executed instructions and other data that the processor may need. This cache speeds up processing by keeping likely needed data close to the processor in static memory, rather than in the dynamic memory that is used outside the CPU and is much slower. Finally, the L1 I-cache and L1 D-cache use even faster static memory than the L2 cache and act as a buffer for the instructions and data to be written back to the memory for the L1 I-cache and L1 D-cache respectively. The numbers of the caches refer to the distance between the CPU and the given cache (i.e. the L1 cache is closer to the CPU than the L2 or L3). When the CPU requests information for an instruction, it will first look at the L1 cache. If the data is not found in this cache, it will then move to the L2 cache and will continue to move outward to the system memory if necessary. The closer the information is located, the faster it can be accessed, since the further the memory is located from the CPU, the slower it will be [45, 46].

Specifically, the AMD Opteron has one L3 cache for all the cores within the multi-core CPU. Each core has its own L2 and L1 caches. The L1 I-cache implements out-of-order execution. This form of instruction refers to how the CPU treats each instruction in the pipeline (which contains a certain number of instructions in a certain order). Out-of-order instruction will execute other independent instructions when it hangs up on a previous instruction, which typically occurs when the needed data is not immediately available. Using this form of execution, the processor is more efficient because it is almost constantly busy.

3.1.2.2 Registers

A register is a memory block that the CPU uses to retrieve and operate on data from the L1 cache [47]. Usually, these are built on flip-flops which can be accessed faster than any of the caches. Modern CPUs have various registers that serve different purposes [48]. We are mostly interested, however, in the special registers used for floating point operations. This section will discuss these registers in general as well as specifically in the AMD Opteron (10h family) of processors.

The first FPU coprocessor was the 8087, which supplemented the 8086. As transistor sizes decreased, this functionality was implemented within the processor itself. In order to utilize the x87 FPU, however, a new set of instructions was needed. Intel added this instruction set and several new stack-based registers to implement this new technology. As time progressed, a technology called MMX was developed to do integer SIMD, which executes the same operations on a vector of data, rather than a single unit. This technology added additional instructions and 64-bit registers that could be divided to represent eight 8-bit, four 16-bit, or two 32-bit integers. The registers, however, were aliased onto the x87 stack-based registers, so only an x87 or MMX instruction could be executed at a time [49].

Following MMX, 3DNow! and SSE were developed by AMD and Intel, respectively, for additional SIMD operations. SSE added an additional eight 128-bit registers that were not aliased and so could be used independently. 3DNow! was similar in many ways to SSE, except that it used the same registers as MMX and x87. Since the original SSE was introduced, new SSE instruction sets have been added. Both SSE and 3DNow! added additional instructions and functionality.

Figure 22 illustrates the some of the registers used in the AMD Opteron [50]. The eight 64-bit MMX and FPU registers are included for back-compatibility, but are not recommended to be used. The original eight 128-bit SSE registers (named XMM0 – XMM7) are supplemented by an additional eight 128-bit registers (XMM7 – XMM 15), but are only available in 64-bit mode [50].

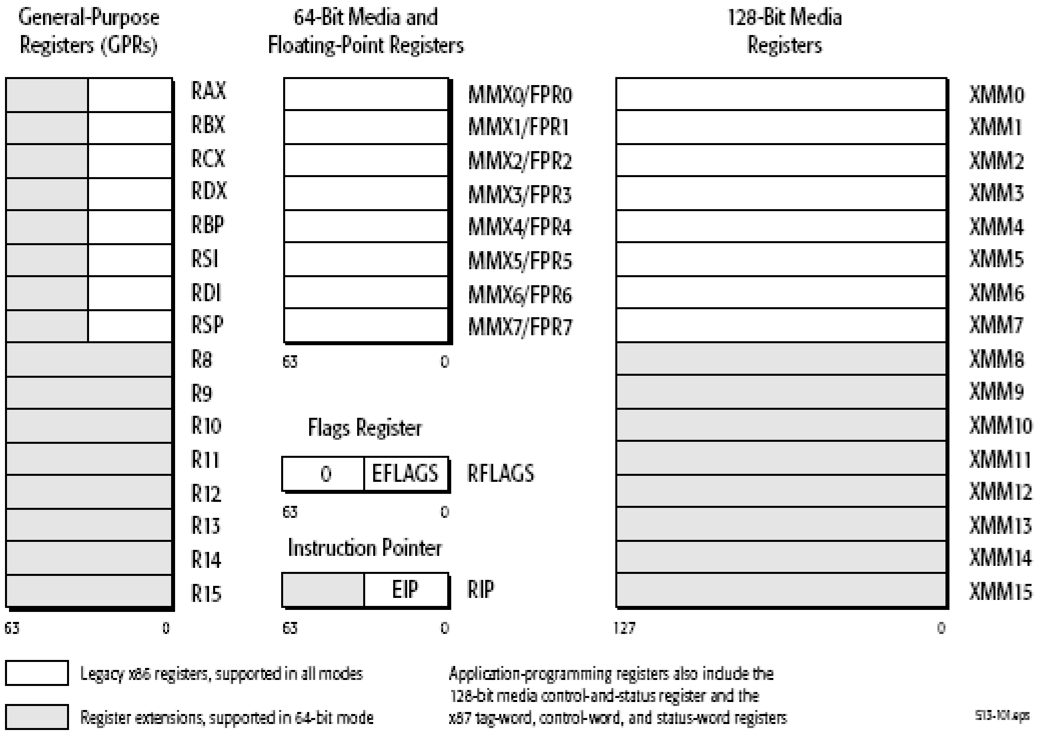

General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
64-Bit Media and Floating-Point Registers
MMX0/FPR0
MMX1/FPR1
MMX2/FPR2
MMX3/FPR3
MMX4/FPR4
MMX5/FPR5
MMX6/FPR6
MMX7/FPR7
63
0
Flags Register
0
EFLAGS
RFLAGS
63
0
Instruction Pointer
EIP
RIP
63
0
128-Bit Media Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0
Legacy x86 registers, supported in all modes
Register extensions, supported in 64-bit mode
Application-programming registers also include the 128-bit media control-and-status register and the x87 tag-word, control-word, and status-word registers
513-101.eps

Figure 22: Application Programming AMD Opteron (10h Family) Registers

AMD recommends using SSE instructions, which utilize the XMM registers, for floating point and SIMD operations because MMX and x87 instructions and their associated registers are included for back-compatibility and are not as efficient [50]. When applications are written in SSE, information must be loaded from external memory into the XMM registers and then is operated on in the FPU. The result is then returned to the registers for writing to memory or additional computations.

3.1.2.3 FPU

The main components of the AMD Opteron FPU are the scheduler and execution unit. The scheduler utilizes out-of-order execution and can calculate three macro-operations every second. It can hold 36 instructions of 12 lines containing three macro-operations each. These instructions are then sent to one of three components of the execution unit. These components are FADD, FMUL, and FSTORE. Figure 23 illustrates the dataflow between the different components of the FPU [45].

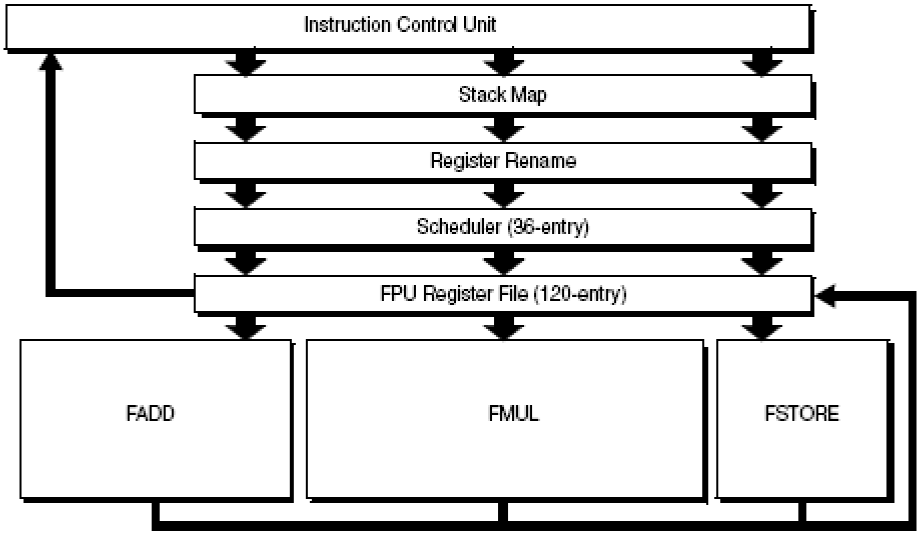


Figure 23: FPU for AMD Opteron (10h Family) Microarchitecture

Generally, FADD is used for floating-point addition and subtraction, while FMUL is used for multiplication and division. FSTORE is used for moving data and other operations. These three units are typically referred to as pipes. All three pipes can be used simultaneously. If a given set of instructions includes an equal mixture of additions and multiplies (which take the same amount of time to process), the FPU will execute these instructions at the speed of just the additions or multiplications (e.g. 12 additions and 12 multiplications will, ideally, execute as fast as 12 additions alone) [45].

Instructions can be given to the FPU using a number of different instruction sets. These include instructions in x87 floating-point, 3DNow!, MMX, and SSE (up to 4a). AMD provides expected latencies for each instruction in these sets. This information assumes:

1. The instruction is located in the L1 cache and has already been fetched and decoded with the operations in the scheduler.
2. The memory operands (i.e. registers, memory locations, or other data) are in the L1 data cache.
3. The execution and load-store unit resources are not needed for other operations [45].

According to AMD, if these assumptions are held, a double precision floating point addition or multiplication will require four clock cycles (or six if the operands are in memory rather than in the registers). For a 2.5GHz processor, this would translate to an addition or multiplication requiring 1.6ns (or 2.4ns). On the other hand, division requires at least 16 cycles (or 18 if the operand is not in the registers), which would correspond to a processing time of 6.4ns (or 7.2ns). Finally, a square root operation takes a minimum of 27 cycles, or 10.8ns. Actual performance, however, is difficult to determine, since in reality the processor must load and decode the instructions, which may require loading from other memory locations. What this information does provide, however, is the best case scenario.

Latency, however, is only part of the issue. Due to the parallelized nature of the FPU, certain operations can be executed in parallel. Even though an addition may require four operations that each take a clock cycle each (i.e. a latency of 4 clock cycles), if the four operations can be done in parallel, the throughput will be one operation per one clock cycle (or

1/1). In fact, this is exactly the case for an AMD Opteron processor. Addition, subtraction, and multiplication have theoretical throughputs of 1/1 (i.e. one operation per clock cycle). Division, on the other hand, has a throughput of 1/13 for single precision, single data instructions and 1/17 for double precision, double data instructions. Finally, square root has a throughput of 1/16 for single precision, single data, and 1/24 for double precision, multiple data.

The issue of single data versus multiple data brings up another issue in the theoretical throughput of a system. A multiple data operation takes all the numbers held within an XMM register and executes the operation on all the data within the destination register. Essentially, one operation can be executed on eight single-precision numbers (four entries in both registers), or four double-precision numbers (two entries in each register). When this factor is taken into account, four single-precision numbers can be added/subtracted/multipled to/from/by four other single precision numbers in one clock cycle. However, if only two individual numbers are operated on, the required time in still one clock cycle.

Latency, theoretical throughput, and SIMD need to be considered when comparing FPUs. An FPU may be able to execute an instruction in fewer clock cycles (i.e. have lower latency); however, if another FPU with higher latency can execute some instructions in parallel, it may in fact be faster (i.e. have higher throughput). In addition, different FPUs may have different size registers or ability to perform SIMD. These issues are essential in understanding FPU operation and speed.

3.1.3 Intel Core Microarchitecture

Intel Core microarchitecture utilizes slightly different methods for floating point operations. Core microarchitecture is used in Intel Xeon processor 5300, 5100, 3000, and 3200 series as well as Core 2 Duo, Core 2 Extreme, and Core 2 Quad processors. The following sections discuss this microarchitecture, specifically the FPU module and differences between this microarchitecture and AMD's 10h family of microprocessors.

One notable difference between Intel Core and AMD Opteron microarchitecture is that Intel only uses two caches, as opposed to the three used by AMD. Intel shares the L2 cache between all cores (rather than the shared L3 cache and independent L2 cache of AMD). Figure 24 illustrates the Core microarchitecture [51].

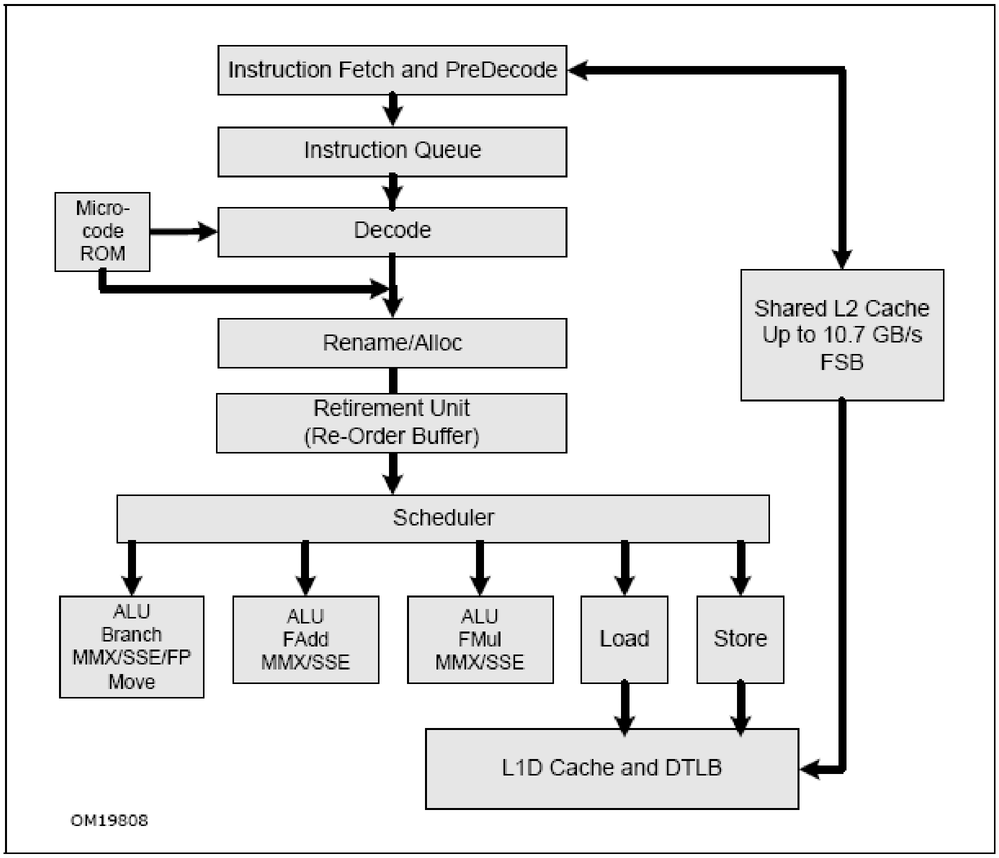

Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Shared L2 Cache Up to 10.7 GB/s FSB
Scheduler
ALU Branch MMX/SSE/FP Move
ALU FAdd MMX/SSE
ALU FMul MMX/SSE
Load
Store
L1D Cache and DTLB
OM19808

Figure 24: Intel Core Microarchitecture

One of the most notable differences between Opteron and Core microarchitecture is the execution unit. While AMD utilizes a separate ALU and FPU (allowing integer and floating point operations to execute simultaneously), Intel essentially has a combined unit that utilizes five pipes. The first three pipes (numbered 0, 1, 2) can either execute an ALU instruction (e.g. integer addition, or logical compares), or can execute certain floating point operations. Pipe 0 is responsible for moves and branches, Pipe 1 completes floating point addition/subtraction, and Pipe 2 can compute floating point multiplication or division. Pipes 3 and 4 are used entirely for load and store functions, respectively. Using this structure, each core of the CPU can execute up to three ALU operations simultaneously, or a floating point move, addition/subtraction, and multiplication/division, or any combination of ALU/FPU operations that follow this structure [51].

Core microarchitecture is nearly identical to Opteron (10h family) for the SIMD registers. Like AMD, Intel utilizes eight 64-bit MMX registers that are aliased on the x87 stack-based registers. In addition, Core utilizes eight 128-bit XMM registers (for SSE instructions) in 32-bit mode and sixteen 128-bit XMM registers in 64-bit mode [52].

Intel compiled a list of latencies for each operation, similarly to AMD. According to these numbers, a single floating point addition instruction requires 3 clock cycles, or 1.2ns in a 2.5GHz core. A double-precision floating point multiplication requires 5 clock cycles, or 2.0ns, while a single-precision floating point division would require 32 clock cycles, or 12.8ns. Finally, a basic square root function needs 58 clock cycles, or 23.2ns [51]. Table 10 and Table 11 below summarize the advertized latencies for similar operations in Opteron and Core microarcitectures [45, 51].

Architecture/Instruction	Add	Multiply	Divide	SqRt
AMD Opteron (10h Family)	4	4	16	27
Intel Core	3	5	32	58

Table 10: Latency Comparison of Advertized Required Cycles for Opteron (10h Family) and Core

Architecture/Instruction	Add	Multiply	Divide	SqRt
AMD Opteron (10h Family)	1.6	1.6	6.4	10.8
Intel Core	1.2	2.0	12.8	23.2

Table 11: Latency Comparison of Advertized Required Time (ns) for Opteron (10h Family) and Core (based on 2.5GHz core)

Throughput values also need to be taken into account. Intel Core microarchitecture specifies the throughput of addition, subtraction, and multiplication for double precision single data and multiple data at one cycle (1/1). Division, however, for double precision numbers (both single and multiple data) is rated at a theoretical throughput of 31 clock cycles (1/31). Finally, square root instructions have specified throughput values of 57 (1/57) for both single and multiple data. The throughput specifications are listed below in Table 12 for both AMD and Intel microarchitectures.

Architecture/Instruction	Add	Multiply	Divide	SqRt
AMD Opteron (10h Family)	1	1	17	24
Intel Core	1	1	31	57

Table 12: Throughput Comparison (all double precision SIMD)

Both Opteron and Core microarchitectures advertise fewer clock cycles for certain floating point operations. AMD seems to have a clear advantage in throughput; however, in order to determine real world functionality of these processors other factors need to be taken into account. The cache, clock speed, and numerous other issues affect the actual speed of the operations. In addition, the mixture of instructions could favor one microarchitecture over the other, due to the pipe structures. A real world test would need to be completed to compare both companies' microarchitectures.

3.1.4 Summary

After reviewing current methods for evaluating floating point arithmetic in currently available commercial processors and FPGAs, several key differences emerge. First, FPGAs are clocked at a much lower rate than most CPUs. A CPU can achieve clock speeds in excess of 3.4 GHz, whereas the fastest FPGAs currently on the market cannot exceed 600MHz [53]. This translates to a difference of nearly 6 times. In addition, a CPU can execute a floating point addition or multiplication in as little as a single clock cycle on multiple data values, or less than 0.4ns. For FPGAs to be practical at lower speeds, they would need to incorporate a large amount of parallelism. Rather than utilizing a system with a single CPU core that can execute one multiplication and one addition at the same time in under 1ns, an FPGA could perhaps be used to complete multiple floating point operations in 50ns.

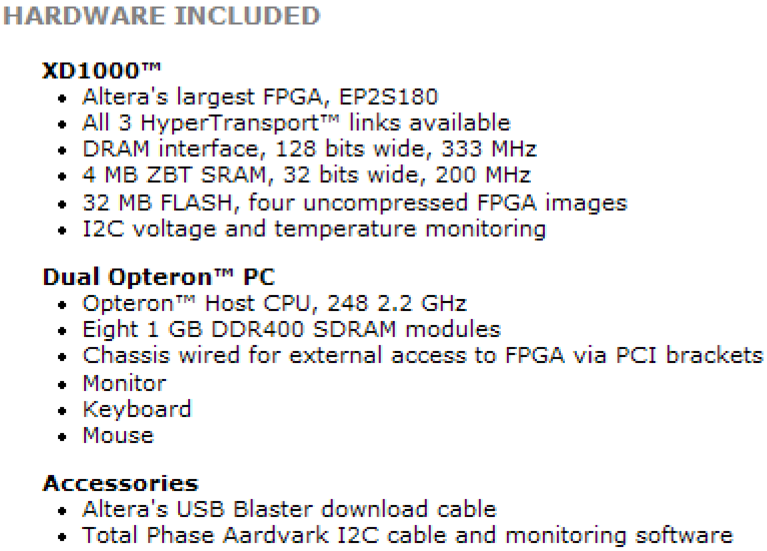
In summary, the strength of FPGAs lies in its ability to massively parallelize tasks. Due to this issue, we will need to implement as many FPUs on the test FPGA as possible. In addition, we will need to take into account how the compiler translates our given code. From the data gathered on AMD and Intel microarchitectures, it is important to utilize SSE instruction sets. Moreover, two times the registers are available when operating in 64-bit mode. These considerations will all come into play during the benchmarking process.

3.2 Benchmark Testing using Fundamental Mathematical Functions

After researching CPU and FPGA architectures and their methods for implementing floating point math, we began the process of benchmark testing both platforms. The following sections describe the system that was used during these tests as well as the programming process. The following chapter will discuss the results of the benchmarks.

3.2.1 XtremeData Inc., XD1000 System

The XD1000 system was chosen for testing during the course of this project. The system was provided by AMD, Inc., and was built by XtremeData, Inc. The system includes a Tyan S2892 motherboard with two CPU sockets. One socket contained an AMD Opteron 285 Dual Core processor (at 2.6GHz), while the other utilized an Altera Stratix II FPGA. Since the FPGA and CPU were both placed in CPU sockets, they connected via the HyperTransport (HT) bus on the motherboard, which allowed for low latency communication. In addition, the processor and FPGA were each provided 4GB of DRAM. More detailed specifications for the XD1000 system are provided in Figure 25 [2].



HARDWARE INCLUDED

XD1000™
- Altera's largest FPGA, EP2S180
- All 3 HyperTransport™ links available
- DRAM interface, 128 bits wide, 333 MHz
- 4 MB ZBT SRAM, 32 bits wide, 200 MHz
- 32 MB FLASH, four uncompressed FPGA images
- I2C voltage and temperature monitoring

Dual Opteron™ PC
- Opteron™ Host CPU, 248 2.2 GHz
- Eight 1 GB DDR400 SDRAM modules
- Chassis wired for external access to FPGA via PCI brackets
- Monitor
- Keyboard
- Mouse

Accessories
- Altera's USB Blaster download cable
- Total Phase Aardvark I2C cable and monitoring software

Figure 25: XD1000 HW Specification

3.2.2 Benchmark Testing on the Opteron

In order to test a CPU's ability to perform floating point operations, we developed a simple benchmark test. Figure 26 below illustrates the flow of the program. The benchmark test includes options. One can either generate a set of random numbers or read from a file that was already generated. Using the latter option results in an overall speedup of the program. In addition, a user can specify which operation to complete. The first version of the program included addition, subtraction, multiplication, division, and square root, while the second version included sine, cosine, tangent, exponent base e (natural number), log base e, and log base ten. The output of the benchmark program was the duration of the test in microseconds.

The first essential programming obstacle was generating a large number of floating point random numbers. We decided to implement single precision numbers during the tests and that the list of random numbers generated needed to be long enough to require at least one second of operation. Since the system clock is only accurate to within 10ms, a one second test allows 1% precision, which would be sufficient for our purposes.

The list of random numbers was generated using the GNU Scientific Library (GSL). This process is outlined in Figure 27 below. First, we generated a random single precision number between zero and one. Following this, we generated a random integer between -127 and 127. The floating point number was then multiplied by two exponentiated by the random integer. The result was a random positive floating point number that fell within the range of the IEEE754 standard for single precision floating point numbers. This process was then repeated in a loop until the specified number of floating point numbers was generated. In our testing, we found that 300 million numbers was adequate to keep the processor busy for one second when computing basic arithmetic.

After generating the list of random numbers, the floating point decimal and hex representations were written to files for later use. The utilized libraries generated the same list of random numbers each time; however, due to time concerns, the list was saved and then imported into the program, rather than regenerated each time.

Following random number generation, the process was fairly straightforward. First, the program would read the system time and then perform the specified operation in a loop, moving through the array of random numbers. After the loop finished, the program would read the time again, find the difference in microseconds and output the time to the screen.

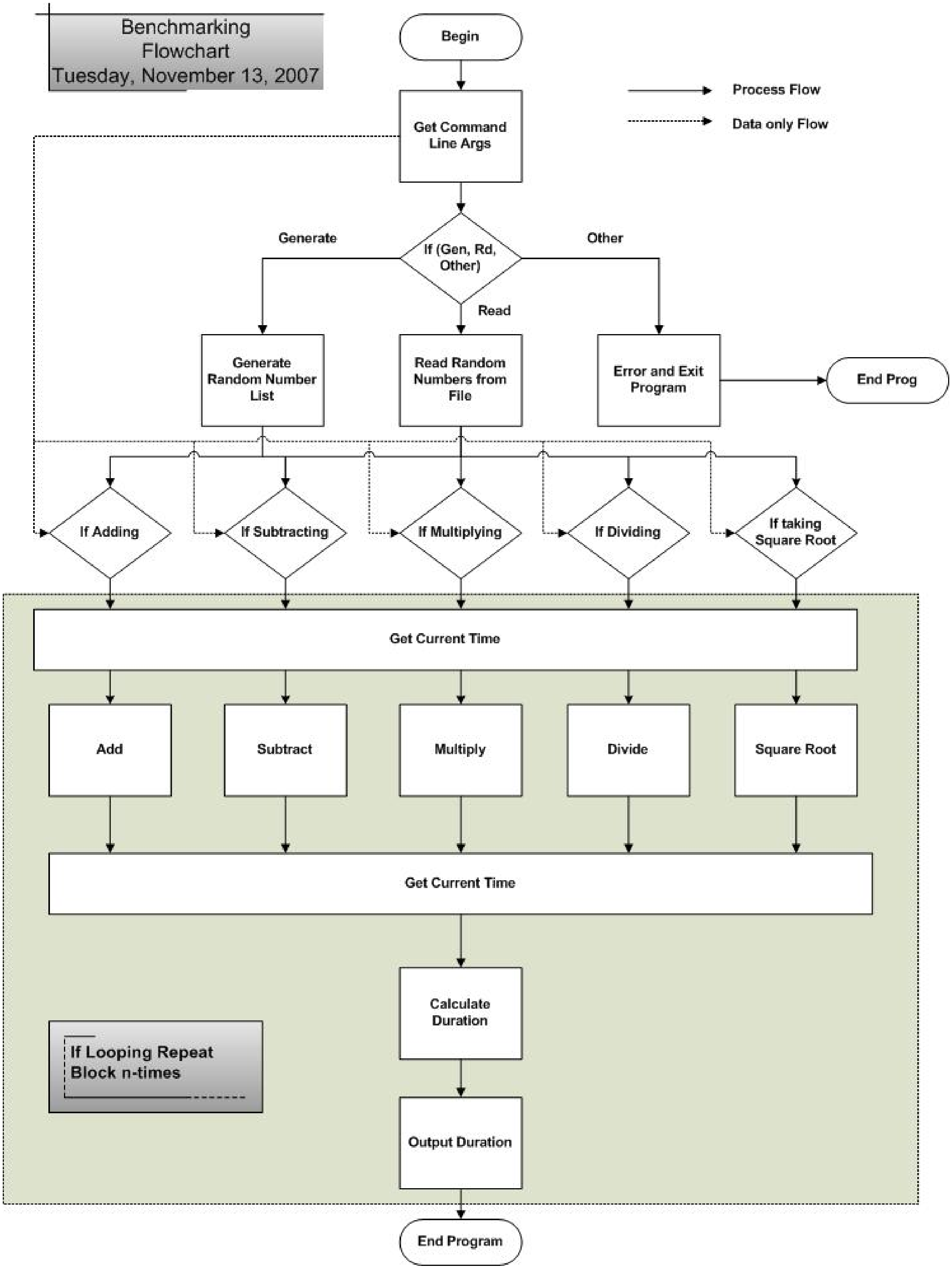


Figure 26: Opteron Benchmark Flowchart

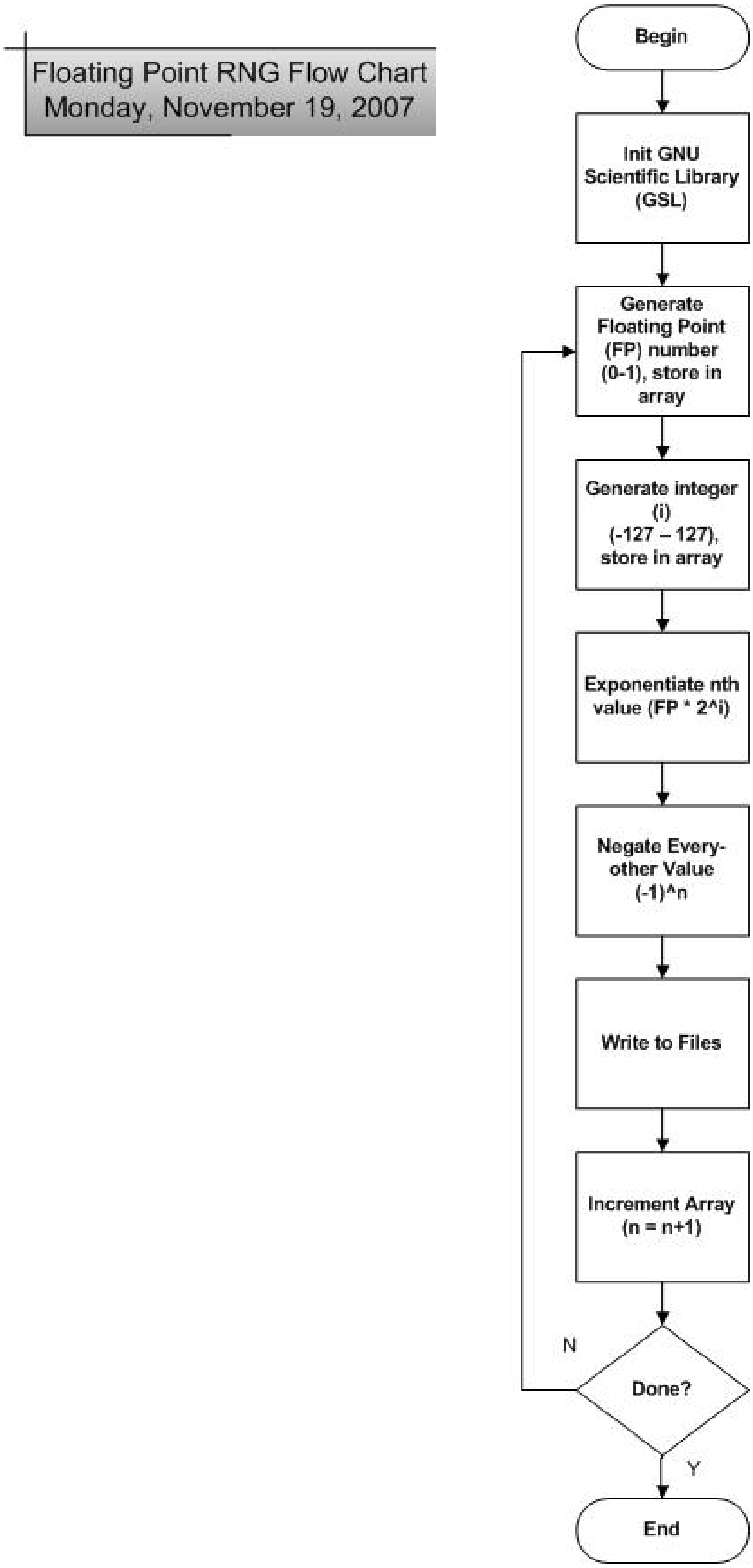

Floating Point RNG Flow Chart
Monday, November 19, 2007
Begin
Init GNU Scientific Library (GSL)
Generate Floating Point (FP) number (0-1), store in array
Generate integer (i) (-127 – 127), store in array
Exponentiate nth value (FP * 2^i)
Negate Every-other Value (-1)^n
Write to Files
Increment Array (n = n+1)
N
Done?
Y
End

Figure 27: Random Number Generator Flow Chart

For testing purposes the code was slightly modified. Unfortunately, generating a new list of random numbers for every operation required a considerable amount of time (several minutes for a 300 million number list). Reading these numbers from a file reduced this time, but still required a few orders of magnitude more time than the operation itself. We decided then to loop the operations on the already loaded list of random numbers to expedite testing. Each operation was performed 100 times providing us with much more data with a negligible additional time requirement. Following this alteration, a script was also written that would write the output of the 100 iterations to a text file as well. Finally, this script was altered so that after one test finished and the result was written to a file, the next operation would start running automatically and would write the result to another file. This process allowed us to gather more data with less error.

The final consideration in testing the Opteron's floating point performance was compiler optimizations. The GNU C/C++ (GCC) compiler was used during these tests. GCC provides a myriad of code optimizations that can be used to increase performance. We decided to look at several compiling options. First, we decided that it was important to consider x87 instructions as well as SSE. This option can be specified during the compilation process. Second, the -ffast-math option was utilized to examine improved speed. Finally, a loop unrolling option was utilized to look at improved speed. Every combination of these options was examined, which required eight tests to be run for each operation.

3.2.1 Measuring Accuracy

Although speed was the primary consideration of these tests, the accuracy of the results was also an issue. In many applications, increased speed comes at the cost of inaccuracy. In order to test the accuracy of calculations, several changes were required. First, the code was altered so that the results of the calculations, not the duration was written to a file. The time component was removed as it was no longer needed. Next, the code was altered to read from a list of 10,000 numbers, rather than 300 million. This greatly improved the speed of the comparisons and would still allow for a high level of confidence. Finally, a new program was written that would read the results in and output the accuracy.

Unfortunately, accuracy with floating point numbers is difficult to determine. How can you determine which values are right and wrong? In addition, how do you represent these differences? We determined that measuring accuracy relative to a certain method would be the best option. The reference option we used was the x87 list without any further optimizations, as this is the most fundamental option. The other lists of results would be compared to the x87 list. The program would output two results. First, it calculated the relative error (i.e. percent difference). After this, it calculated the difference in number of representations (i.e. how many possible floating point representations exist between the reference and given result). These calculations allow one to determine the relative accuracy of the program given certain options. These results could then also be used after testing on the FPGA was completed, allowing us to determine both speed and accuracy improvements using this system.

3.2.3 Benchmark on FPGA

Now that we have a better understanding of the performance of the Opteron when dealing with floating point operations, the next step is to do some comparisons with the FPGA. There are two main objectives:

1. Gaining the ability to create HDL designs that can perform floating point calculations, transforming the FPGA into a floating point calculator.

2. Finding a way to communicate between the Opteron and the FPGA in order to do a comparison. The XD1000 system was created with the HyperTransport bus for this purpose. An effective method to exploit this usage was important.

This section is divided into five subsections. The first discusses the approach we chose in order to accomplish the second objective. The following subsection provides a working example from Impulse Accelerated Technologies to demonstrate their results, which includes a performance comparison and communication between the FPGA and the Opteron. Throughout this section, an explanation on how the design is built and how the HyperTransport was used will be provided. The last three subsections describe our own designs for floating point calculations.

3.2.3.1 Impulse C set up

XtremeData included test code written in C to perform overall checking. These tests were mainly used to check for abnormal behavior inside the system before and after FPGA configuration. Specifically, these tests included memory, interrupt, and HyperTransport testing. This code used the HyperTransport bus to communicate between the FPGA and Opteron. However, the code was complicated and provided few comments. In addition, it did not show how to synchronize HDL designs from user PC with the CPU. In other words, answers to questions such as “How are the pins of the FPGA connected to the HyperTransport bus?”, “Where does the FPGA store the imported and calculated data?”, “What are the timing constraints when dealing with data transfer?” were unknown.

With little information provided in the reference design, Impulse C language was chosen for our approach in meeting this objective. The advantages of Impulse C were made clear during our research. In order to design the simplest program on the XD1000, a developer must follow several steps. First, a developer would need to create a floating point addition HDL and place and route the pins of the FPGA to the design, which entails keeping track of about one thousand available pins of the FPGA for a system like the XD1000. Provided he does not run into any problems while designing the device, he still has to write software code to communicate with the system, which must incorporate timing constraints from the hardware design and data transfer. He would also have to devise a method to move data in and out of the FPGA while keeping track of memory constraints. Fortunately, Impulse C implements these steps in two C programs and VHDL libraries. One C program is used for hardware description, which lays out the communication as well as the high level design of the project. This uses HDL designs from the included libraries or can utilize designs from the individual developer. The other program describes the software which controls the processor. In our case, this was used to compare the results and control data. Impulse C eased considerably the challenges presented by this project.

In general, writing a program in Impulse C involves several steps. First, the developer writes the hardware and software side code, which must include I/O between the CPU and FPGA. This can be accomplished using memory blocks. The developer can program the hardware to read in data from the system memory to the FPGA SRAM or can stream it from a shared memory location external to the FPGA. Just like with cache memory on a CPU, the closer the data is to the hardware, the faster it can be accessed. If possible, reading the data into the FPGA internal memory is usually fastest.

After writing the software and hardware C code to successfully implement the I/O and to do the chosen math operations, the developer simulates the code by building a simulation executable. The Impulse C compiler (called CoDeveloper) takes the software and hardware code and tries to run it as if it were implemented in the target system. This simulation is limited; however, as it cannot simulate all conditions. If the program passes simulation, the developer then generates the HDL using the compiler. This translates the hardware C code into HDL for programming onto the target platform. If this process executes properly, the user then must export the hardware, which, in the case of the XD1000, starts the XD1000 Platform Support Package which connects the generated logic with the XtremeData HyperTransport and memory interfaces. Finally, it generates a batch file to complete the hardware generation process. The last phase using CoDeveloper is exporting the software, which copies the generated C code to a software directory and generates a make file [54].

At this point, the CoDeveloper portion of development is completed. The project, however, is not yet ready to be run on the system. The sof file needs to be built to program the XD1000 FPGA. This is done using the command line. After moving into the Quartus directory of the Impulse project, the developer runs "run_quartus.bat," which is a script file that was generated during the hardware export process. This batch file runs Quartus and builds the sof file, which is used to program the FPGA. This process usually takes at least an hour, but frequently will require four or more hours to complete [54]. Often errors in the design, including implementing math functions that cannot fit within the FPGA are discovered during this process.

Once the FPGA sof is completed, the developer can open the file in the Quartus programmer and transfer it to the FPGA on the XD1000. After booting the operating system, the developer then has to copy the software files from the software directory of the project from the User PC to the XD1000. After being copied, the user in some instances must alter the make file. During the development process, we needed to add "-lrt" to the last line of the make file to get the timing. After these changes are made, the developer opens a terminal and navigates to the directory where the software was copied. He then makes the project by typing "make". Finally, if no errors were reported, issuing the command "./run_sw" will run the executable that was just built. If no bugs exist, the program will properly run in software and will communicate with the FPGA as it does its computations [54]. This process is summarized in Figure 28.

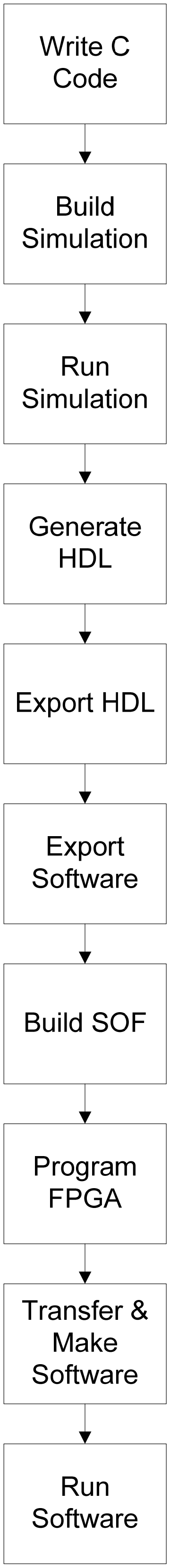


Figure 28: Developing Process with Impulse C

3.2.3.2 Matrix Multiplication

In order to better understand Impulse C and FPGA capabilities, Impulse Accelerated Technologies provided us with a matrix multiplication example. In this program, the design performs a 512x128x512 matrix multiplication and then adds with a 512x512 matrix C. The goal of this design was to compare the speed and accuracy performance of the FPGA and the Opteron. A developer can change the size of the matrices with some understanding of the memory constraints of the FPGA and HyperTransport bus. System constraints are discussed in the Results and Conclusions section.

Figure 29 provides a top level block diagram of the system. The design consists of three main blocks, the CPU process, calculator processes (named mmproc), and I/O process (named ioproc).

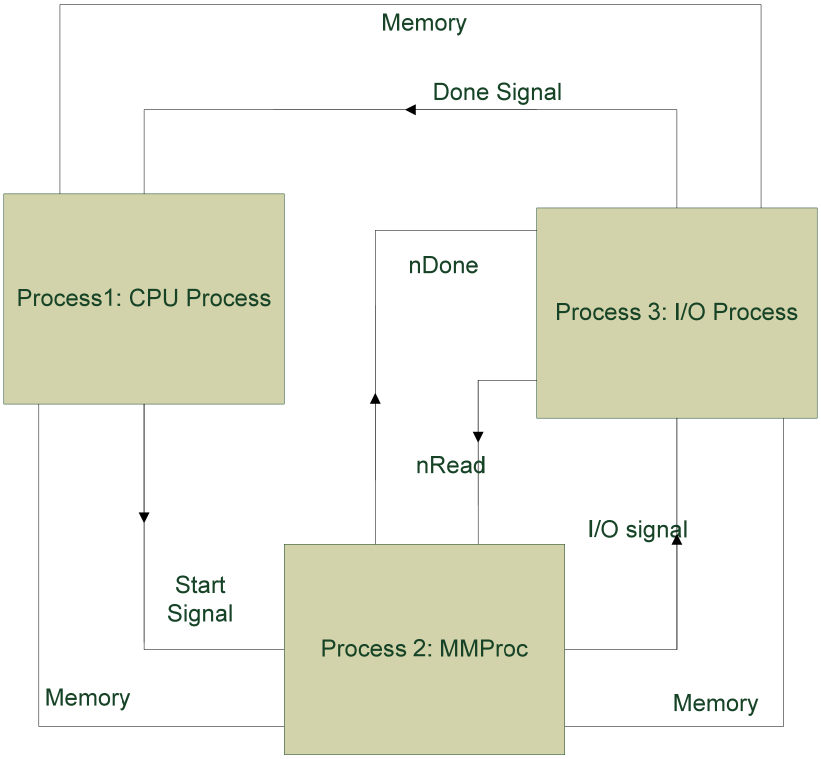


Figure 29: Generic top level block diagram for Comparison Performance Project

Each block communicates with each other using signals and memory blocks which contain data. The data contained in these blocks are double precision floating point numbers. When the system is turned on, all the blocks are initialized with appropriate adjustments, such as loading data and preparing appropriate memory spaces. Then, the CPU process posts a start signal to the calculating process and waits until it receives the done signal. Once the calculating process mmproc is ready, it posts an I/O signal to start the I/O process. After that, the calculating process and the I/O process will communicate to each other, transferring and processing data, through the variables nRead and nDone. nRead keeps track of the number of rows that have been read, while nDone records how many rows have been calculated. Once the computation finishes, ioprocess will post a done signal to the CPU process at which point it continues with the remainder of the software side program. The remainder of the program often recalculates the computation in order to compare the results of both the FPGA and CPU.

In this example, the designers decided to store the data in a specific way. The CPU creates three matrices: A(512x128), B(128x512) and C(512x512), stored as arrays. The matrix notation can be understood as matrix_name(row, column). On the Opteron side, each element of the array is a double precision floating point number and is stored in the system memory. On the FPGA side, the designers decided to store all the numbers of matrix B in the FPGA's local memory as an array called Bin. Each element of this array contains 16 double precision floating point numbers. To obtain data from the CPU memory, ioprocess creates two arrays Ain and Cin, which only take two rows of the matrices A and C at a time. Each element of these arrays also contains 16 double precision floating point numbers. Besides memory constraints which are listed in the Problems Encountered section of the next chapter, there were also some other limitations of note. For example, N_UNROLLED is a constant which represents how many operations will be done in parallel and A_ROWS represents the number of rows in matrix A. A_ROWS therefore has to be a multiple of N_UNROLLED. In addition, memory addressing requires A_COL and N_UNROLLED to be powers of 2. Also, even though the FPGA has enough resources to fit 32 parallel double precision multiply and add units (MACs), Quartus II tools can not assign enough block RAM to feed all of them. Due to this limitation, N_UNROLLED cannot be greater than 16. However, using single precision, 32 parallel MACs is possible.

Through careful data management, mmprocess carries out 16 MAC operations at a time. In addition, the design is implemented so that ioprocess can read inputs A and C by rows and store the output back to C in parallel with the computation done by mmproc. For example, while mmproc is computing C[1][ ] + A[ ][1]xB[1][ ], ioprocess is reading C[2][ ] and A[ ][2] and storing the results from C[0][ ] + A[ ][0]xB[0][ ]. For a better understanding of this design, flow charts for each process are provided in the following sections.

3.2.3.2.1 CPU process

As shown in Figure 30, the CPU processes are fairly simple. There were four matrices required in this software. Three matrices were created to store operand values while the last matrix was used to store the reference results computed by the microprocessor. After generating a list of random numbers and initializing the matrices, the CPU process calculates the reference results and time required for the microprocessor to finish the task. Then, the CPU prepares the appropriate data and signals the FPGA to compute the result.

During this process, the block only sends a start signal to the mmproc block and waits for the done signal from the ioproc block, which is triggered once the computation finishes. This cycle marks one complete calculation. Depending on the value of N_ITERATION, both the microprocessor and the FPGA will repeat the same operation for N_ITERATION times. When both results are ready, the CPU process begins its time and accuracy comparison. The CPU process also provides how many floating point operations are completed and how much time the microprocessor and FPGA needed to complete the calculations. The accuracy comparison checks relative error based on a threshold. The threshold of error here was 1e-5. If any value in the resultant matrices has a relative error larger than the threshold error, the accuracy test fails and displays conflicting results. The equation to compute relative error is provided below:

$$relative\_error = \frac{|C[x] - D[x]|}{|D[x]|}$$

In this equation, array C[x] stores the results of the FPGA while D[x] stores the reference results from the microprocessor. In code, one should note that the timing comparison is only done when the FPGA was actually used. It is reasonable because even though the computer simulation can imitate the functionality of the FPGA, it cannot determine timing for the FPGA. On the other hand, the accuracy test has been proven to be reliable and useful to debug the algorithms.

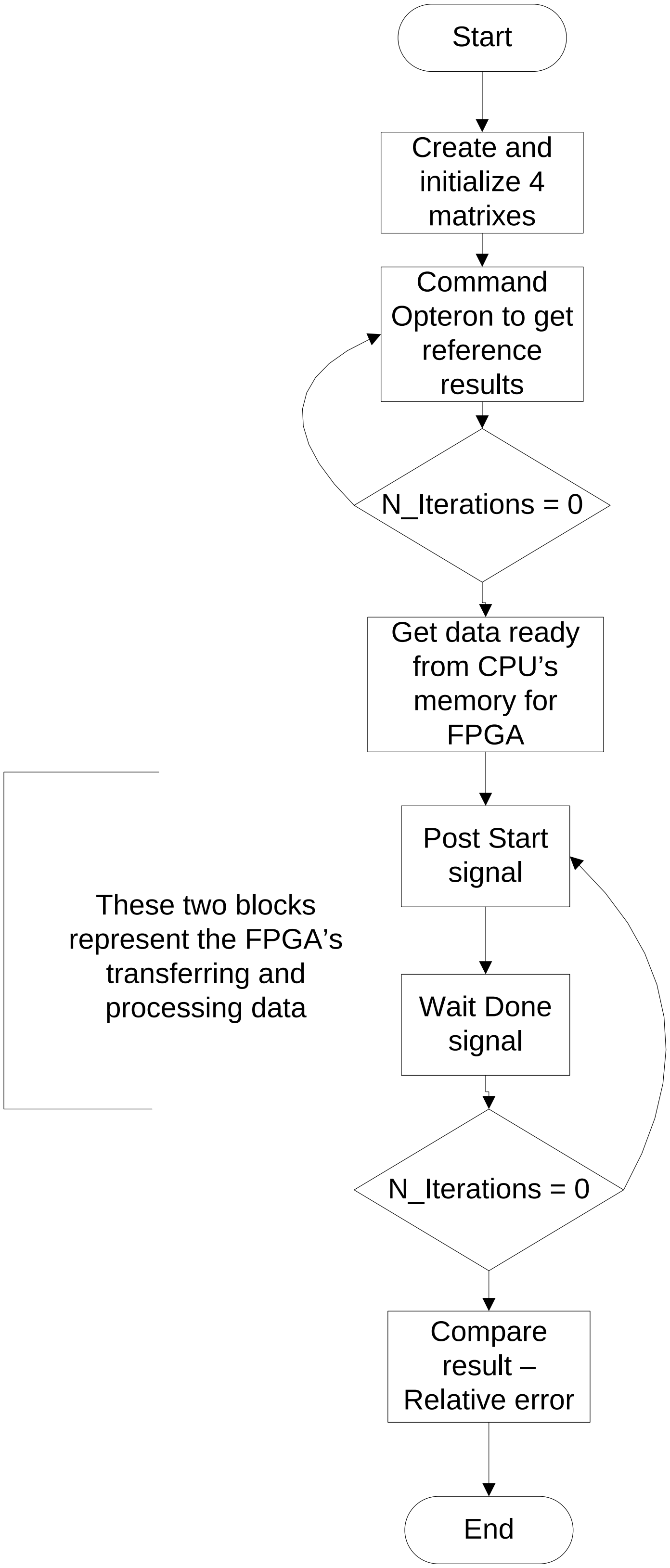

Start
Create and initialize 4 matrixes
Command Opteron to get reference results
N_Iterations = 0
Get data ready from CPU's memory for FPGA
Post Start signal
Wait Done signal
These two blocks represent the FPGA's transferring and processing data
N_Iterations = 0
Compare result – Relative error
End

Figure 30: CPU process flow chart

3.2.3.2.2 I/O Process

The I/O process is used to read in memory from the CPU into the FPGA for processing and to write back results into memory. As mentioned above, ioproc is designed to do both tasks in one working cycle. Specifically, when the block is powered, it will initialize a memory offset to correctly point at the data sets and then wait for the start signal from mmproc. Once the signal is received, ioproc will read two rows the first time and read one row for all the remaining iterations. The reason for this is to make sure the data available for mmproc is constantly working without waiting for ioproc to load data between computations. Once each row is processed, ioproc will receive the results and store them back into the memory before reading another row. Note that for each read and write operation the offset updates in order to maintain correct data management. As mentioned above, ioproc and mmproc communicate with each other through two variables: nRead and nDone. nRead will signal ioproc when the entire matrix computation has finished and then ioproc posts the done signal to the CPU process. Since all input and output arrays, Ain, Bin, and Cout, are designed to store numbers in two rows, it is necessary to determine which row will be stored, written or sent out for mmproc to process. A flow chart of this block functionality is provided in Figure 31 below.

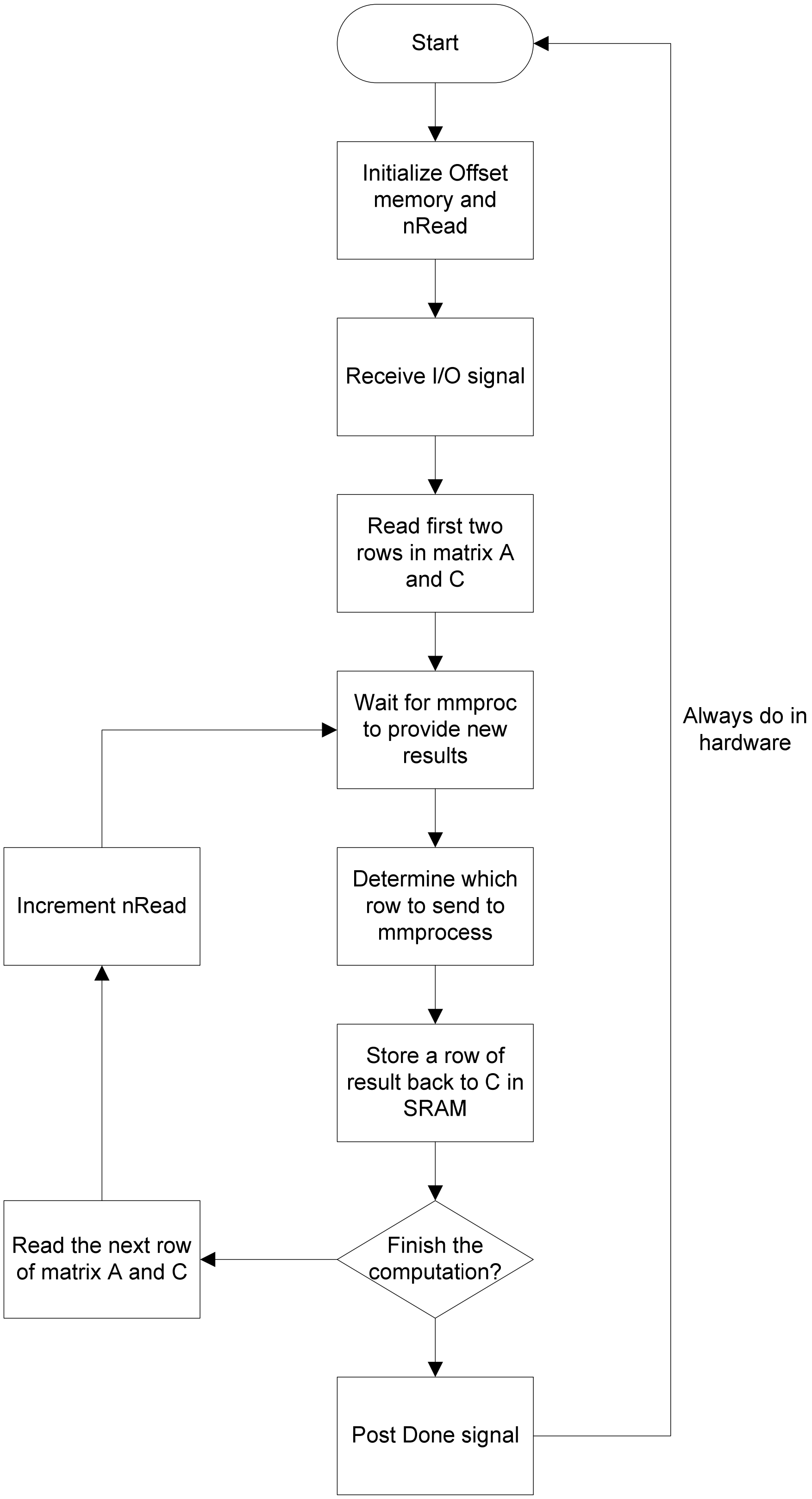

Start
Initialize Offset memory and nRead
Receive I/O signal
Read first two rows in matrix A and C
Wait for mmproc to provide new results
Determine which row to send to mmprocess
Store a row of result back to C in SRAM
Finish the computation?
Read the next row of matrix A and C
Increment nRead
Post Done signal
Always do in hardware

Figure 31: Flow chart for IO process

3.2.3.2.3 Calculating Process - mmproc

The calculating process is the most complicated process of the three, mostly because it requires careful data management while processing each group of operands. Mmproc starts when it receives the start signal from the CPU process. It begins with a memory offset and nDone initialization then sends out a go signal to ioproc to start loading data. Mmproc then reads matrix B and stores it into the FPGA's local memory for faster access. As a result, the block has to create a wide enough array to store the B values and requires careful manipulation in order to extract the appropriate B values corresponding to the A and C values. For example, mmproc has to determine which row it is computing. It also needs to identify which array element it is using because one element of input arrays contains 16 numbers. Because of the rather complex nature of matrix multiplication, mmproc also needs to decide if the result is final or temporary since one result value is related to all the operand numbers of that respective row and column. The final result is stored in the Cout array while the temporary results are stored in the Ctmp array. The details of the data management are provided in the commented code (see Appendix file).

Note that mmproc checks for when the entire computation is finished. Through communication with ioproc using nDone and nRead, and by keeping track of which row it is processing, mmproc can determine if the entire computation is finished. However, it is ioproc's task to send out the done signal to the CPU process. Mmproc simply resets the nDone variable and is ready for the next computation if it is required to recalculate the matrix. In addition, if the program runs more than once, mmproc must reset the memory offsets as well as reload input array Bin. A flow chart for mmproc is provided in Figure 32 below.

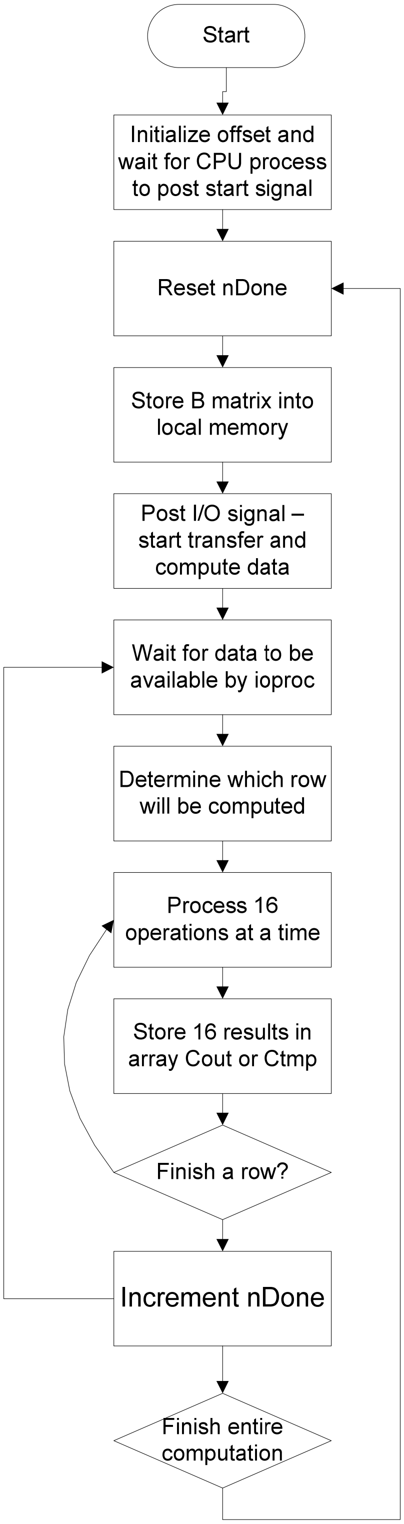

Start
Initialize offset and wait for CPU process to post start signal
Reset nDone
Store B matrix into local memory
Post I/O signal – start transfer and compute data
Wait for data to be available by ioproc
Determine which row will be computed
Process 16 operations at a time
Store 16 results in array Cout or Ctmp
Finish a row?
Increment nDone
Finish entire computation

Figure 32: Flow chart for mmproc

3.2.3.3 Primitive Operation Benchmark

After thoroughly going over the matrix multiplication example provided by Impulse, we developed several simple arithmetic benchmarks. These were based on the matrix multiply example since our knowledge of Impulse C was still limited and our code would be prone to error. We changed the matrix multiply example so that it would operate on a larger set of data (two sets of 2^{17} double precision numbers). In addition, we built multiple versions of the program, including addition and multiplication with a different number of iterations and varying data sets.

Generally, the block diagrams for the addition and multiplication projects are similar to matrix multiplication shown in Figure 29. All of them have three process blocks: the CPU process to control the flow between hardware and software, ioproc to read and store data in the FPGA, and mmproc to process data. Ioproc and mmproc still communicate using the two variables nRead and nDone.

Previously, the matrix multiply example read in one entire array of numbers into the FPGA SRAM and read the other arrays in rows from the FPGA board. However, the number of inputs we chose too large to be read in all at once, so we changed the code to read both arrays with shared memory from the CPU. There are two main reasons that we decided to increase the number of operands. As discussed in the background chapter, financial firms are interested in leap performance in computing complex algorithms such as Black-Scholes equation. Those simulations require considerably large sets of random numbers. Thus, the first reason is to determine how many resources the FPGA has available and whether the FPGA is ready to be treated as a coprocessor in financial world. The second reason was to increase the accuracy test's efficiency. As the number of operands increase, the chance to produce inaccurate results increases.

The general functionality of each process was still the same. There were some minor changes required in data management in order to ensure proper calculation. One problem arose in the calculation block mmproc. Since these projects no longer use local memory as the matrix multiplication example did, as soon as mmproc receives a signal from the CPU process it sends a signal to ioproc immediately. We needed to add a command co_par_break() in order to help the hardware distinguish between two signals. After determining the memory management, we simply changed the hardware section to do addition or multiplication instead of multiply/add operations. The software side of the program calculated the arithmetic on the CPU as well and compared the timing and numerical results of both the CPU and FPGA. The results of these benchmark tests are provided in the Results and Conclusions section.

3.2.3.4 Division Benchmark

Developing a division benchmark required some additional changes from the previous programs. Since division is a more difficult operation, it requires more space in hardware. Changing the parallel operations from 16 to eight was necessary so that it would fit on the FPGA. Technically, the FPGA was large enough to fit 15 dividers. However, because the hardware memory can only process with number of data of a multiple of 2, the next smaller block has to be eight. Several changes had to be made to make this possible. Specifically, each element of input and output arrays were changed to fit eight numbers. Methods to extract and insert the bits for each floating point number from one element of the array, PUTELEM() and GETELM(), also needed to be updated. In addition, we had to use the XtremeData Whitebox libraries instead of the XtremeData Altera libraries, since the latter did not contain a division primitive.

Similar to other projects, the block diagram of the project is the same as other primitive projects. It still consists of CPU process, ioproc and calculating process. The flow chart of each block is also similar to what is shown in Figure 30, Figure 31, and Figure 32. The changes mentioned above are simply to ensure proper calculation. After these changes were made, the division benchmark was built and simulated properly, allowing us to test the capabilities of the XD1000 system while doing division.

3.2.3.5 Square Root and Transcendental Functions Benchmarking

The next phase of the benchmarking process was developing several transcendental benchmarks. We tested sine, cosine, and natural log. The Altera and XtremeData libraries in Impulse C do not currently provide transcendental primitives. The only way to implement these functions was to either:

1. Develop VHDL code, write a wrapper to interface with Impulse C, add code to the hardware program.
2. Write C code using primitives, tell the compiler using a #PRAGMA to generate the lines in hardware as a primitive.

We initially tried to implement a square root function in VHDL; however, creating a wrapper file proved tremendously time consuming as it has to follow a certain set of interfaces while maintaining the correct functionality of the square root file. Specifically, in order to include one external HDL file into a project, one needs to follow three steps. First, the developer needs to make a directory called “userip” and copy the HDL files there. CoDeveloper is designed to look for and read any external HDL file in this folder. Second, since the files in the userip folder are assumed to be compatible with CoDeveloper, the designer must create an appropriate HDL interface for each external HDL file so that CoDeveloper can synchronize the file with the hardware description in the project. It is important to understand how the included HDL file operates as well as the name of the HDL entity that will be used. Impulse C defines a few categories for external entities. For each category, there is a specific interface that must be followed so that the compiler understands. For example, “logic” indicates there is no clock included in the entity’s behavior while “async” refers to entities that need multiple clock cycles to produce results. A double precision floating point divider, thus, would need to be marked as “async” and would need to strictly follow an interface like the example below.

In the hardware C code of the project, one needs to create a function to call the divider as shown:

```
Double my_division( double data, double datab){
  #pragma CO implementation my_division async    // this line will point to the HDL divider
                                                 // in userip
   Return (dataa/datab);            // this is solely for computer simulation to produce
                                      // corresponding results
}
```

At this point, the interfaces are firmly defined by the HDL compiler. This includes signals, timing, etc. for correct interaction between the generated HDL by the Impulse C compiler and the external HDL. For the example above, the HDL interface would have to look like the one shown below:

```
Entity my_division is
    Port (
          signal reset          : in std_logic;
          signal clk             : in std_logic;
          signal request        : in std_logic;
          signal dataa          : in std_logic_vector(63 downto 0);
          signal datab          : in std_logic_vector(63 downto 0);
          signal r_e_t_u_r_n : out std_logic_vector(63 downto 0);
          signal acknowledge: out std_logic));
     end;
     …..          // Description code to link this interface with external floating point entity
```

The third step is to integrate the generated Megacore within the entity "my_division," which would include instantiating the Megacore and integrating all signals required by it with those of the above interfaces. This is the most time consuming step, since it also includes times spent in HDL simulation, such as ModelSim to ensure correct operation.

As a result, Impulse advised us to consider the second option. Using Taylor and other series approximations, we were able to build these functions using primitives. Since mathematical approximations for sine, cosine and natural logarithm are fairly straight-forward, we decided to program these functions first. Unfortunately, due to many problems with the alpha version of Impulse C, we spent all the remaining time debugging and fixing these problems and did not have the chance to complete the square root.

Changing the memory management was also necessary, however, since these primitives only take a single number (e.g. sin(1.4) or tan(-0.3), not 4.3 + 9.32). Removing one array from the program solved this issue. After simulating the results and including an adequate number of terms to achieve reasonable accuracy, we built the programs onto the XD1000 system. The results are outlined in the next chapter.

3.2.3.5.1 Sine function

Since Impulse C and XtremeData do not currently have HDL files to compute the sine function, we decided to build our own sine function based on Taylor series. Taylor series for both sine and cosine functions only approximate accurately when within a certain range, sharply deviating after a certain point. Due to this issue, we normalized the input values to within the range of negative π/2 and π/2 inclusive. It is reasonable that our hardware device should be able to cover all the steps, normalization and calculating results. However, during our trials, we discovered that the FPGA has very limited resources to cover both procedures while maintaining a reasonable accuracy level at the minimum relative threshold error of 1e-5. Due to this, we decided to let the computer produce the random numbers and normalize them before they were stored in memory. As we do not know precisely how the microprocessor computes trigonometry, we hoped this method would prevent both the CPU and the FPGA from going through the normalization process and thus maintain a reasonable comparison.

The flow of the transcendental projects was the same as the matrix multiplication example. The sine project consisted of three block processes: the CPU process to control CPU and FPGA flow, ioproc to receive and write data in and out of shared memory, and mmproc to calculate the results. The flow of this project was the same as previously. There was only one change in this project - the function to describe the sine calculation. Thus, the block diagram and flow charts of this project are similar to the ones in Figure 29, Figure 30, Figure 31, and Figure 32.

Although using Taylor series is one simple approach to evaluating trigonometry, there are many ways to implement the sine function. The equation we used to approximate sine function is shown below:

$$Sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

Additional terms provided more accurate results. However, one additional term would result in increasing the size of the project greatly. Since we can only use primitive operations, the power function was not an option and thus an expression such as x^7 required seven multipliers. Thus, creating a function with a sufficient number of terms became impossible and would crash Quartus during the fitter step.

We fixed this problem by using recursive loops to decrease the number of operators. This method proved to be effective since the compiler only created as many operators as it could find in the function. Thanks to the nature of looping, the compiler would use the same resource over again instead of creating more hardware. Since a divider is much larger than an adder and a multiplier, they were avoided by multiplying by the reciprocal of the coefficients. Using this method, we were able to fit eight sine functions into the FPGA, which could be used in parallel.

For sine function, five terms with normalized inputs would produce accuracy up to 1e-7 threshold relative error. One way to optimize this function is to include a #PRAGMA CO PIPELINE inside the two loops. One loop was used to calculate the various powers of x while the other loop combined the values of both arrays to return the results. However, even though the Impulse team expected a #PRAGMA directive to work, our version of CoDeveloper would not compile and thus we had to leave this option out. The code for this project is provided in the Appendix file.

3.2.3.5.2 Cosine function

Similar to the sine function project, the cosine project also used a normalized input and a Taylor series to approximate the results. The equation for the cosine function is provided below.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \ldots$$

We encountered the same problems with this project as in the sine project, which were fixed using similar methods. However, unlike the sine function, the cosine function requires seven terms to produce an accuracy of 1e-5 threshold relative error. In addition, we created two versions of the project. Even though #PRAGMA CO PIPELINE would not work inside the primitive function that calculates cosine, we were still able to exploit pipeline behavior when parallelizing eight cosine functions at a time. One version was used to calculate the cosine result without any pipeline behavior and the second version tried to exploit the pipeline option. The code for these projects is provided in the Appendix file.

3.2.3.5.3 Natural logarithm

The natural logarithm proved to be more complicated than sine and cosine. One reason for this was that there were no efficient approximations found for natural log whose input's absolute value is larger than 2. As the result, we had to create two additional functions which could extract the exponent value and mantissa value of the input data. Our approach for natural log was first to calculate the binary logarithm of an input and then multiply by the reciprocal of the binary log of value e. Using this approach the natural logarithm became an easier challenge. Extracting the exponent value provided the integer result of the binary logarithm. The fractional part of the result could be calculated by a simple approximation of binary log [55]. Since the mantissa of any floating point number is always smaller than two, we can use an equation to approximate the fractional part of the input.

$$\log_2(1+x) = 1.4425449290x - 0.7181451002x^2 + 0.4575485901x^3 - 0.2779042655x^4 \\ + 0.1217970128x^5 - 0.0258411662x^6$$

The method to obtain the exponent value and mantissa value of one number requires mostly bit manipulation. However, extracting the mantissa proved to be an easier task. First, we cleared all the bits except the 52 mantissa bits and then cast them with 1023 which is 0x3FF for the sign and exponent bit. By doing this, we essentially turned the exponent of the number into zero, clearing the sign bit and thus found the positive mantissa value. To obtain the exponent value, we first AND the number with 0x7FFFFFFFFFFFFFFF to clear the sign bit. Then we right shifted 52 bits to obtain the biased exponent value. Upon finding the difference between the biased exponent and 1023, we received the integer value of the exponent. We then had to convert the integer representation into a float, which can be complicated. Thus, we were forced to use an operation called FX2REAL64(x,d) (Impulse believed this operation was unpredictable and might cause a hang up in the FPGA). This command converts a fixed point representation x with d fractional bits into a double precision floating representation. Another option is using casting to a double data type; however, this option was not a viable solution in this version of the development tools.

Other than these three functions (obtaining the exponent value, obtaining the mantissa value, and calculating natural log), the remainder of the program was the same as the others. The overall project still had 3 block processes: CPU process, ioproc, and mmproc. The flow chart of each process was also the same, except the calculation process would have to call the three new functions when calculating the natural logarithm of each number. One issue worth noting is that the equation we used in this process was somewhat inaccurate; the simulation could only pass with threshold relative error of 1e-3. After simulating the results and including an adequate number of terms to achieve reasonable accuracy, we built the programs onto the XD1000 system. The results are outlined in the next chapter.

4.0 RESULTS AND CONCLUSIONS

Testing speed and accuracy improvements using an FPGA coprocessor on the XD1000 system provided useful insights into the future of this technology. The following section describes the problems encountered during this process as well as the results of our benchmark tests.

4.1 Problems Encountered

The following section describes many of the problems that were encountered during the course of the project. This information is included as a finding in order to evaluate the maturity of the tools currently available for FPGA development. These issues include setting up of the system, communication between the FPGA and CPU, Impulse C implementation, and limitations of the XD1000 system.

4.1.1 Set Up

Upon arrival at the project center, the XD1000 system was not fully set up for development. The system ships largely assembled with Fedora Linux 6 installed and a test application included. The only hardware assembly required was connecting the JTAG and Aardvark cables for programming the FPGA and CPLD respectively, which attach to ports on the back of the system. The majority of the set up was required on the User PC, which is used to program the XD1000 system.

The User PC was a Dell Precision 490, provided by JP Morgan, with two dual core Intel Xeon processors and 4GB of RAM. The included operating system was Windows XP Professional SP2. In order to begin development, we first installed Altera’s Quartus II, which is used for programming Altera’s FPGAs. In addition, ModelSim was installed for simulation purposes before programming the system. Both programs required licenses before they could be used. In addition, the XD1000 system required a license for the HyperTransport bus. Obtaining these licenses was somewhat difficult, especially with XtremeData. The time required to obtain licenses detracted from the already limited time available for the project.

After the licensing issues were resolved we continued by configuring Quartus II for development on the XD1000 system. This process included specifying system memory and the HyperTransport bus used, which varies depending on model. Due to an incorrect setting during the configuration process, we were unable to program the system for several days. After working with XtremeData for a period of time, this issue was finally resolved. To program the XD1000, the system was first halted. While improperly configured, Quartus would transfer a sample project to the FPGA via the JTAG, lighting up an orange LED on the FPGA. The remaining four LEDs on the FPGA would then turn on, indicating a problem. Upon restarting the system, the XD1000 would beep and would need to be shut down completely to delete the volatile image on the FPGA. Once we properly configured Quartus, the system would program similarly, but after the data transfer process, the LEDs would turn off and the system would either reset itself, or would need to be reset manually, but would not beep upon booting Linux.

4.1.2 Communication

Sending information between the CPU and FPGA proved to be a difficult task. The supplied test code from XtremeData demonstrated this process. The test code, in addition to other processes, sent data to the FPGA via the HyperTransport bus and then measured the latency and bandwidth. This was done using the supplied drivers. Unfortunately, the code contained few comments and was too difficult to understand based on the code alone. Rather than trying to learn the drivers, we decided to utilize Impulse C, which was recommended to us by XtremeData. The Impulse C CoDeveloper was able to generate VHDL and the communication code that was necessary to use the XD1000 FPGA as a coprocessor. While Impulse C aided communication, it was not without problems, which are discussed in the following section.

4.1.3 Impulse C Implementation

The Impulse C CoDeveloper tool allows a programmer to code low level operations in C, including communication between a CPU and FPGA coprocessor. This extension of standard C, however, did require some effort to learn. Impulse C utilizes custom functions that signal processes to start and wait on the FPGA and CPU as well as share data between the two blocks. Due to the limited time available during this project, we decided to modify a provided matrix multiplication example to do simple arithmetic. We would then expand on this program to do more complicated math.

As we developed an Impulse C program based on the matrix multiply example, we ran into several problems. First, licensing and configuration issues caused some initial downtime. Since Impluse C uses Quartus to program the FPGA and synthesize the HDL, it was essential that these programs work together. Quartus generated a license error initially when running the batch file generated by Impluse C. A configuration file, we found, had to be altered to look for a standalone license rather than a server based one. In addition to these initial problems, Impulse C generated syntax errors during simulation in certain cases where legal C syntax was used. For example, a for loop can be written as for(int i=0; i<5; i++); however, Impluse C would require the integer variable "i" to be declared outside the loop.

When the program would finally build and simulate properly, other minor issues with generating or exporting the HDL files were encountered. One of the first problems was that a project opened under Impulse C must be contained in a directory without spaces in the name. Having a project in a file with spaces would result in errors. In addition, once the project was successfully built and the sof file was generated for programming the FPGA on the XD1000, we discovered that, in the case of the matrix multiply example, the FPGA's calculations were ten percent or more off from the CPUs calculations. This error was due to a timing problem in the latest version of Impulse C CoDeveloper for the XD1000. To resolve this issue, we had to roll back to a previous version of CoDeveloper, which required considerable time to configure. Multiple errors were encountered, involving files and patches that were required for building the sof file. These problems were finally resolved when Impulse provided us with an older version of CoDeveloper with a custom patch that would make it work in large part like the most recent version. This mixed alpha release of CoDeveloper allowed us to build sof files with proper timing (which permitted accurate, repeatable results) and was fairly easy to use during the building process.

Occasionally, although the system would completely build and generate the sof file and would simulate properly, the program would not work when we implemented it into the XD1000. The times that this happened, typically a problem with communication or memory management was the culprit. A co_par_break was inserted by Impulse Accelerated Technologies after we reported that the program would hang up when implemented in the XD1000 system. The simulator did not catch a problem, but the experienced developers at Impulse added the line on a hunch. This function separated a signal wait and signal receive function. When these functions were not separated by a line of code, they would cause the system to hang up. Adding co_par_break fixed this problem. Another problem we encountered involved memory management. When changing the program to do eight simultaneous calculations instead of 16, we had to change a size declaration. The system simulated properly and did not report errors during the building process; however, when we implemented the program in the XD1000, the results were incorrect. These issues were frustrating since they were difficult to detect and building the sof file often takes hours. An improved simulator would be very helpful in these situations.

Impulse C CoDeveloper is still immature and requires considerable patience and time to use. The job of programming an FPGA and CPU to work together on a task is much more difficult than programming a CPU alone. In addition, the process of generating an sof file requires hours even for simple programs involving only a few components. More complicated designs can require nearly a day. While this process is the most time consuming, nearly a dozen steps in all are required to get a design from C code to executing on a system and any step in this process can generate errors that need to be resolved.

Beyond the initial problems with Impulse C during the set up and building of projects, we also ran into some development limitations. Using the custom CoDeveloper that we were provided was difficult beyond the simplest tasks due to library limitations. CoDeveloper is able to work with multiple platforms like the XD1000 and provides libraries for each one it supports. In some cases, multiple libraries can be selected, depending on need. In our case, we were able to use the Altera provided libraries and the XtremeData libraries. The Altera version, however, did not provide division, which seriously limited our development. Changing over to the XtremeData libraries allowed us to do division; however, the library still lacked even a simple square root. Due to these limitations, all functions that were more complicated than simple arithmetic needed to be written in C code (or VHDL) with appropriate preprocessor directives (using #PRAGMA CO PRIMITIVE or #PRAGMA CO IMPLEMENTATION respectively).

Using C code proved easier, but the limitations of the FPGA were still a concern. Many transcendental approximating series require many multiplications. Too many terms resulted in needing more multipliers on the FPGA than could fit. In order to get around this problem, we had to use “for” loops to express in CoDeveloper that we wanted to use fewer multiplier/adders at the expense of time. However, only certain algorithms are able to be separated in this manner.

XtremeData recommended Impluse C as the easiest tool to use for development on the XD1000. The problems we encountered during the development process, however, indicate that the tools still have a long way to go before they are viable for the majority of developers.

4.1.4 XD1000 System Limitations

The XD1000 system itself has some limitations that make it more difficult to program. More than just the limited fabric and pins, the memory available on the FPGA itself and the FPGA board required us to alter our designs. Initially, our Impulse C program read in one entire array of data (2^{20} double precision numbers) with which to compute various operations. The memory required for this data, however was about 67Mbit, while the FPGA’s internal SRAM was only 9Mbit. We then changed the program to stream data from the SRAM on the board; however, this memory was limited to about 4MB. To fix this, we decreased the array size to 2^{17} or about 786KB. Using this smaller set of data, however, required us to repeat the calculation hundreds of times so that the program would run long enough that we could get accurate timing results.

4.1.5 System Memory Limitations on the User PC

The User PC supplied by JP Morgan ran Windows XP Professional (32 bit) with Altera Quartus II 7.1 (32 bit). Due to the 32 bit nature of the operating system, we were unable to utilize more than about 2-3GB of system memory while building the sof in Quartus. This seriously limited some builds, including the provided matrix multiplication example. The FPGA, according to the comments, should be able to allocate 32 simultaneous MACs; however, since Quartus runs out of memory during the fitting process, it can only allocate 16. This limitation causes the capabilities of the FPGA to be under-evaluated. According to CoDeveloper, a 64 bit version of Quartus (along with a 64 bit OS) would fix this problem, allowing Quartus to utilize 16GB or more memory. This will be an essential step as FPGAs continue to grow exponentially in size over the next few years [56, 57].

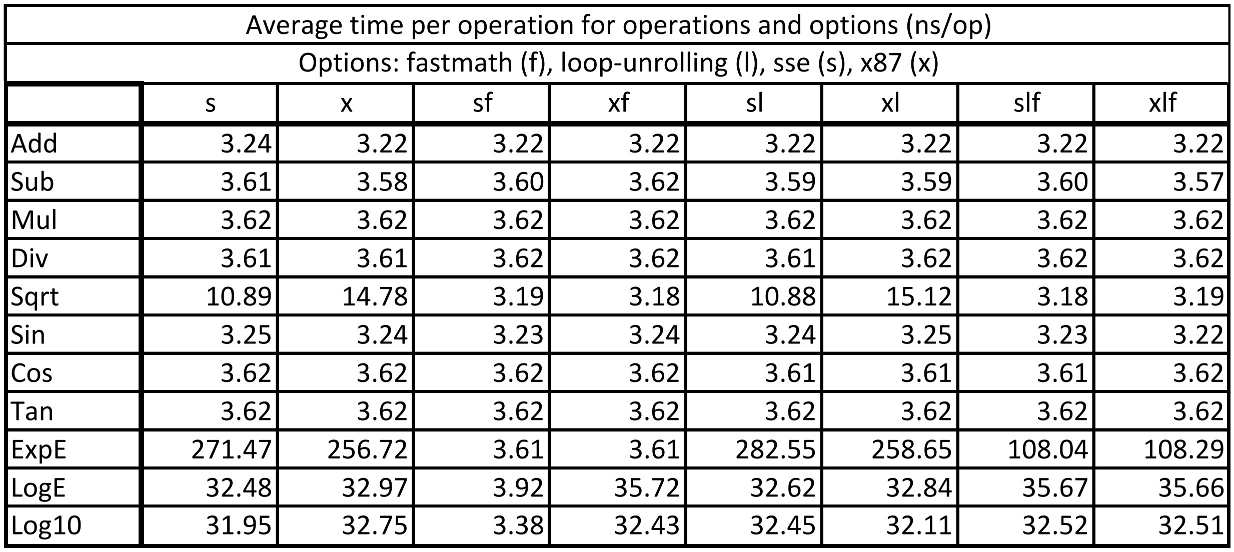
4.1.6 Summary

Getting the XD1000 system prepared for development required considerable time throughout the project duration. Numerous problems were encountered involving licensing, programming, and the immaturity of the currently available tools. Impulse C provided the communication link that would otherwise not have been available without deciphering hundreds of lines of undocumented C code drivers. This tool could be used effectively in the future when the tools have matured enough that they no longer require such a concerted effort to utilize. The majority of a developer's time must be spent on designing and improving code, not debugging the development tools themselves. Once these tools have matured, FPGA development may prove more viable.

4.2 CPU Benchmark Results

Writing a program to test the speed and accuracy of a CPU computing various arithmetic and transcendental functions proved a much easier task than programming an FPGA coprocessor. The programs were written using a simple text editor and were then compiled using the GNU C Compiler (GCC) on the XD1000. The following section outlines the results of these tests for both speed and accuracy.

The objective of the CPU benchmark was to develop a program that would test the CPU's ability to perform single precision floating point operations with minimal overhead. Certain additional processes, however, are necessary in the benchmarking process. For instance, during this process, an array of random floating point values was generated and stored on the heap using the "calloc" command in C. A pointer to the beginning of this array was then passed to a function that would read the start time, execute the requested operation on the data (incrementing the pointer to the next data location each time) and then return the end time. The overhead of incrementing the data pointer, incrementing the loop counter, and other low level processes (such as loading and storing data from/into registers) were also included in the timing. The results of 100 iterations of this benchmark using different compiler options for each operation is summarized below:



Average time per operation for operations and options (ns/op)								
Options: fastmath (f), loop-unrolling (l), sse (s), x87 (x)								
	s	x	sf	xf	sl	xl	slf	xlf
Add	3.24	3.22	3.22	3.22	3.22	3.22	3.22	3.22
Sub	3.61	3.58	3.60	3.62	3.59	3.59	3.60	3.57
Mul	3.62	3.62	3.62	3.62	3.62	3.62	3.62	3.62
Div	3.61	3.61	3.62	3.62	3.61	3.62	3.62	3.62
Sqrt	10.89	14.78	3.19	3.18	10.88	15.12	3.18	3.19
Sin	3.25	3.24	3.23	3.24	3.24	3.25	3.23	3.22
Cos	3.62	3.62	3.62	3.62	3.61	3.61	3.61	3.62
Tan	3.62	3.62	3.62	3.62	3.62	3.62	3.62	3.62
ExpE	271.47	256.72	3.61	3.61	282.55	258.65	108.04	108.29
LogE	32.48	32.97	3.92	35.72	32.62	32.84	35.67	35.66
Log10	31.95	32.75	3.38	32.43	32.45	32.11	32.52	32.51

Table 13: CPU Benchmark Results

Table 13 shows the average duration of each operation in nanoseconds. Simple arithmetic (i.e. addition, subtraction, multiplication and division) required nearly the same amount of time regardless of compiler options at around 3.6 nanoseconds. Addition required the least amount of time (about 3.2ns), while subtraction, multiplication and division were nearly the same. The square root function required considerably more time, requiring less for SSE compared to x87 instructions and increasing to the speed of addition when using the fast-math compiler option. Sine, cosine, and tangent required around the same amount of time as simple arithmetic, varying little with compiler options. The exponential function (e^x) required by far the greatest amount of time, between 257 and 272 nanoseconds, except when fast-math option was used and the loop unrolling was not, taking only as much time as simple arithmetic. Finally, the logarithm function base e and 10 followed each other closely, requiring between 32 and 33 nanoseconds, unless fast-math was turned on and loop-unrolling was off. In this case, they required only a few nanoseconds.

Although impressive speed improvements were noted using the fast-math option, accuracy was also evaluated. We developed another program to compute average and maximum relative and units in the last place (Ulp) error. After running a sample 10,000 results using every combination of our selected compiler options, the results of the simple arithmetic and square root functions indicated that no error was detected in any file. We then ran the same test with the remaining transcendental functions and found little error, except in a few notable situations. Using the x87 and fast-math options during the trigonometric benchmarks resulted in significant accuracy loss. The average error in these cases was about 24% for sine and cosine and 0.046% for tangent. These results do not take into account the large number of NaN calculations that were generated using the fast-math option. When taking this into account, the average units in the last place (Ulp) error was in the millions (as NaN is represented completely differently than the expected values). The exponential and logarithmic functions did much better, with no error detected while not using fast-math and errors around 10^{-10} detected using fast-math. The maximum Ulp error was 1 in all cases.

While fast-math considerably speeds up certain transcendental functions, this option can considerably detract from accuracy. Frequent NaN results and inaccurate results from trigonometric functions indicate a serious problem with using these functions on a broad range of random numbers. While these problems persist, the increased speed of trigonometric functions with fast-math was negligible. We conclude from our data that fast-math should never be used with trigonometric functions, but can be acceptable with other transcendental functions.

Unfortunately, separating the execution time of these operations from the overhead is not possible. Running a test on the CPU that only loops and increments an array without performing operations results in a similar average time of execution. Due to the pipeline nature of the CPU and the FPU architecture, many operations can be computed in parallel. For example, our research indicated that modern CPUs can complete an integer and floating point operation simultaneously. These issues are important considerations when interpreting the results outlined in this section.

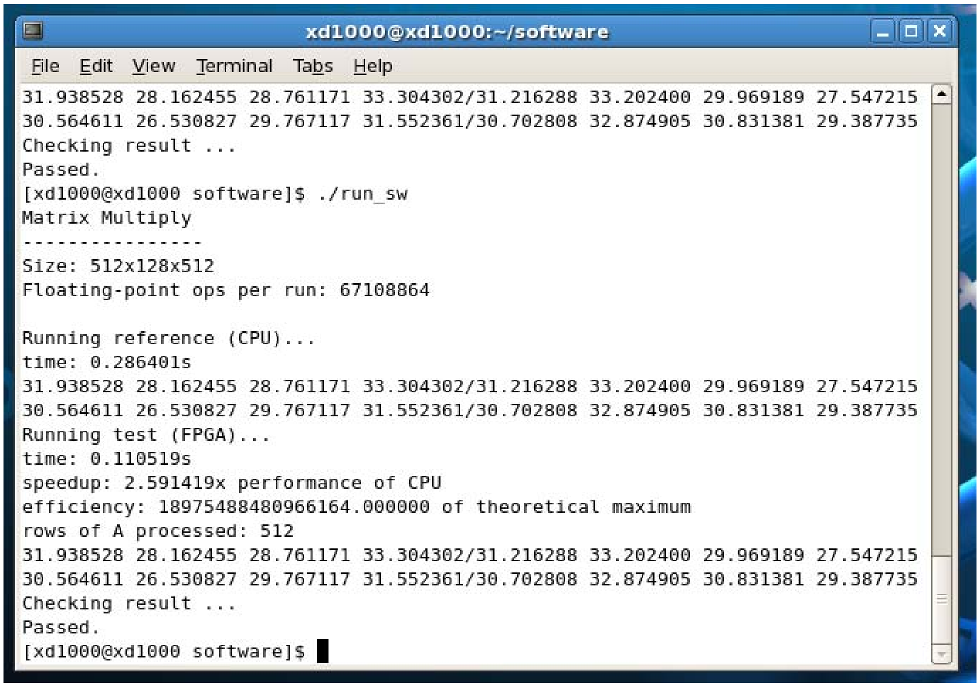
The data gathered during this section of the project provided a method with which we were able to compare FPGA implementations of mathematical functions. The ability of an Opteron CPU using GCC is considerable. Even including certain overhead, the CPU was able to do basic arithmetic in three to four nanoseconds. The next section describes the results of basic and transcendental math applications using an FPGA as a coprocessor.

4.3 FPGA Coprocessor Results

While programming the XD1000 using Impulse C was difficult in many ways, we did obtain some useful data. Several tests were run based on a matrix multiply example provided by Impulse Accelerated Technologies. This section will first outline the timing and accuracy results from the matrix multiply example and then outline the findings from our benchmark tests.

4.3.1 Matrix Multiplication (Double Precision)

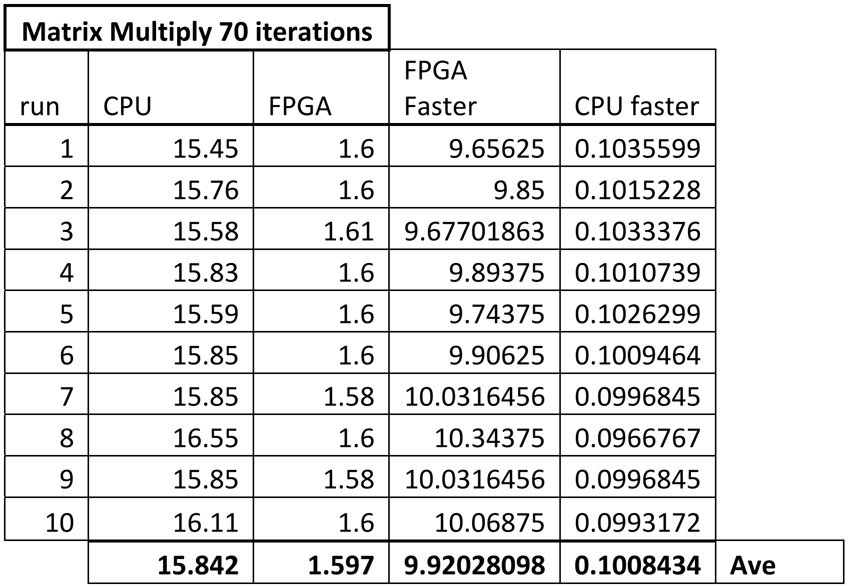
The matrix multiplication example provided by Impulse Accelerated Technologies illustrated the capability of an FPGA coprocessor. The program created three random matrices (512 X 128, 128 X 512, and 512 X 512 respectively) and then multiplied and added them, outputting the result to a 512 X 512 matrix, which corresponds to about 67 million floating point operations. The program timed the FPGA from the point where the processor calls the FPGA to begin until after the FPGA signals that it has finished. It then computed the difference in time and outputs an error if the relative error exceeds 0.000001. After running the example, the result was between three and 14 times speedup using the FPGA with no accuracy problems. Repeated running of the program resulted in varying results, which was likely due to the operating system scheduling. Altering the example to run 70 iterations decreased variability considerably and resulted in an average of about a ten times increase in computations. These results illustrated the capability of the FPGA coprocessor on a limited data set with repeated simple arithmetic. The output of the matrix multiply example on the XD1000 is shown in Figure 33.



```
31.938528 28.162455 28.761171 33.304302/31.216288 33.202400 29.969189 27.547215
30.564611 26.530827 29.767117 31.552361/30.702808 32.874905 30.831381 29.387735
Checking result ...
Passed.
[xd1000@xd1000 software]$ ./run_sw
Matrix Multiply
----------------
Size: 512x128x512
Floating-point ops per run: 67108864

Running reference (CPU)...
time: 0.286401s
31.938528 28.162455 28.761171 33.304302/31.216288 33.202400 29.969189 27.547215
30.564611 26.530827 29.767117 31.552361/30.702808 32.874905 30.831381 29.387735
Running test (FPGA)...
time: 0.110519s
speedup: 2.591419x performance of CPU
efficiency: 18975488480966164.000000 of theoretical maximum
rows of A processed: 512
31.938528 28.162455 28.761171 33.304302/31.216288 33.202400 29.969189 27.547215
30.564611 26.530827 29.767117 31.552361/30.702808 32.874905 30.831381 29.387735
Checking result ...
Passed.
[xd1000@xd1000 software]$ 
```

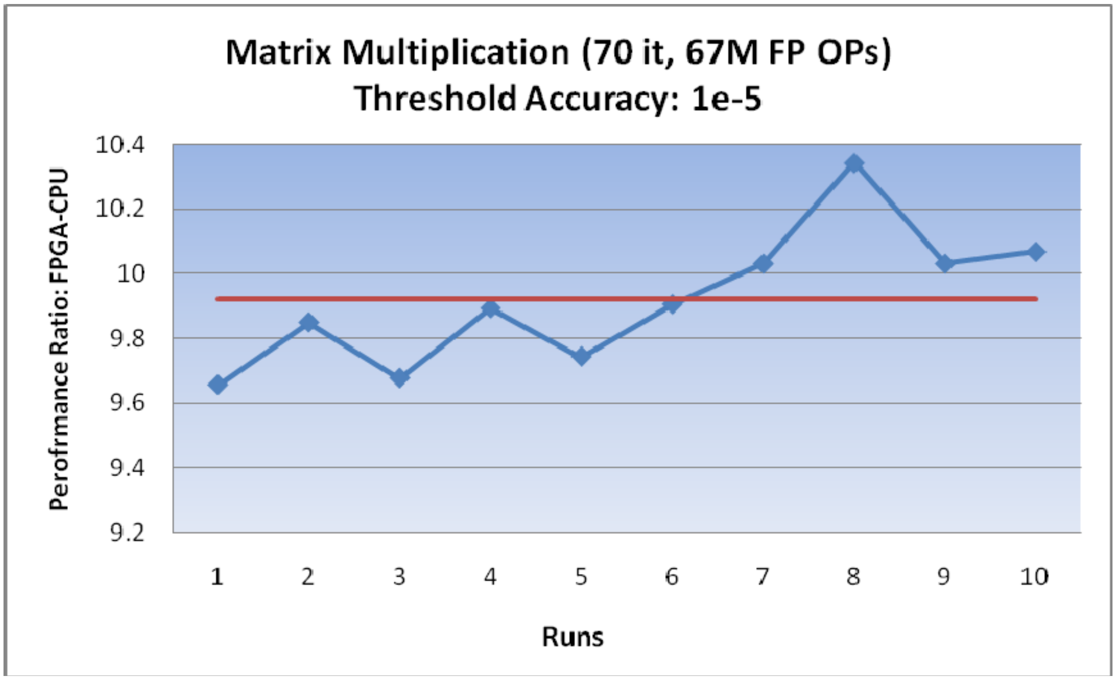
Figure 33: Matrix Multiply on XD1000 (replace!)

Table 14 below shows the results of ten runs of the matrix multiply example with 70 iterations. The FPGA was between about 9.7 and 10 times faster than the CPU during our tests, with the average at 9.9. Figure 34 graphically displays these results. The blue line indicates the number of times faster the FPGA calculated the results compared to the CPU for each run, while the red line indicates the average of these runs.



Matrix Multiply 70 iterations					
run	CPU	FPGA	FPGA Faster	CPU faster	
1	15.45	1.6	9.65625	0.1035599	
2	15.76	1.6	9.85	0.1015228	
3	15.58	1.61	9.67701863	0.1033376	
4	15.83	1.6	9.89375	0.1010739	
5	15.59	1.6	9.74375	0.1026299	
6	15.85	1.6	9.90625	0.1009464	
7	15.85	1.58	10.0316456	0.0996845	
8	16.55	1.6	10.34375	0.0966767	
9	15.85	1.58	10.0316456	0.0996845	
10	16.11	1.6	10.06875	0.0993172	
	15.842	**1.597**	**9.92028098**	**0.1008434**	**Ave**

Table 14: Matrix Multiply Results (10 runs)

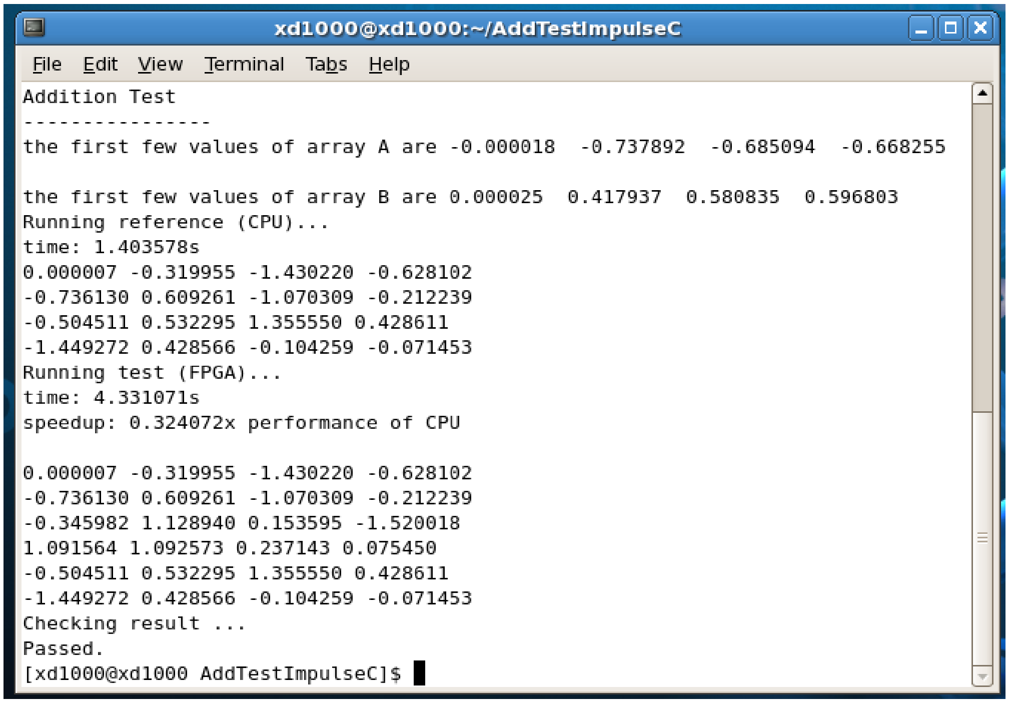


Matrix Multiplication (70 it, 67M FP OPs)
Threshold Accuracy: 1e-5

Figure 34: Matrix Multiplication showing runs (blue) and average (red)

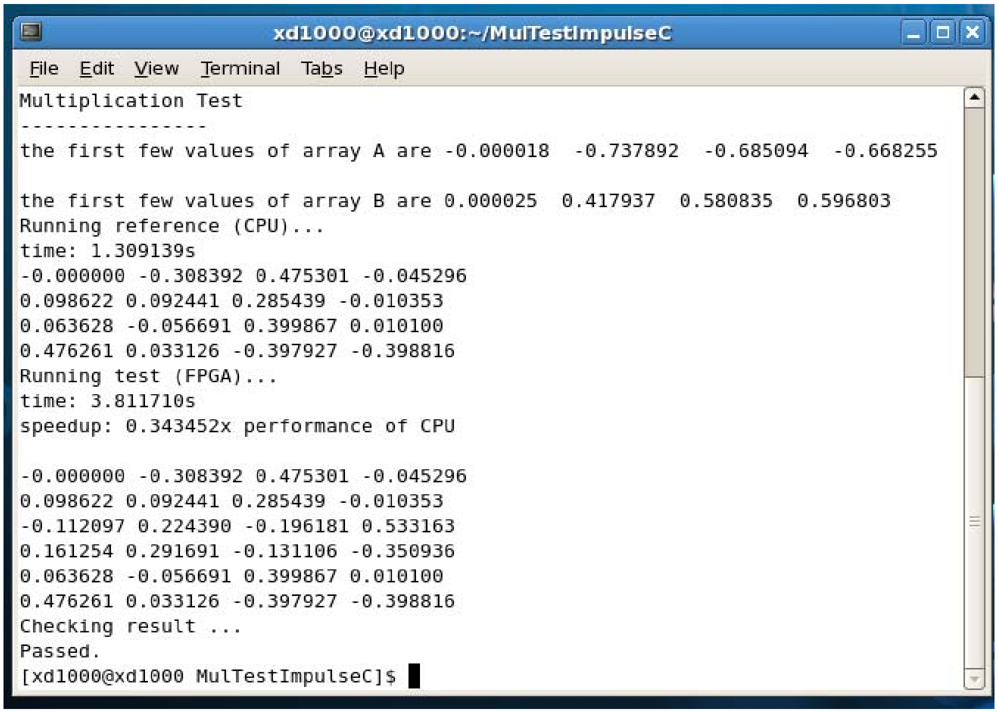
4.3.2 Addition and Multiplication

After running the provided example, we altered the code to do simple addition and multiplication. Using two arrays with 2^{17} double precision numbers, we measured the difference from the start signal to receiving the done signal. The addition program iterated the calculation one thousand times, while multiplication was repeated 900 times. For addition the CPU required about 1.2 seconds while the FPGA worked about 4.6 seconds to finish the same computation. Multiplication, on the other hand, required about 1.2 seconds on the CPU and 4.1 seconds on the FPGA. In both cases, the results were accurate to more than 1.0e-10 relative error, but resulted in about three times latency using the FPGA compared to the CPU. Figure 35 and Figure 36 are screenshots taken from the XD1000 system for the addition and multiplication benchmarks respectively. These findings indicate that using an FPGA on a large data set for such simple operations is not viable.



```
Addition Test
----------------
the first few values of array A are -0.000018  -0.737892  -0.685094  -0.668255

the first few values of array B are 0.000025  0.417937  0.580835  0.596803
Running reference (CPU)...
time: 1.403578s
0.000007 -0.319955 -1.430220 -0.628102
-0.736130 0.609261 -1.070309 -0.212239
-0.504511 0.532295 1.355550 0.428611
-1.449272 0.428566 -0.104259 -0.071453
Running test (FPGA)...
time: 4.331071s
speedup: 0.324072x performance of CPU

0.000007 -0.319955 -1.430220 -0.628102
-0.736130 0.609261 -1.070309 -0.212239
-0.345982 1.128940 0.153595 -1.520018
1.091564 1.092573 0.237143 0.075450
-0.504511 0.532295 1.355550 0.428611
-1.449272 0.428566 -0.104259 -0.071453
Checking result ...
Passed.
[xd1000@xd1000 AddTestImpulseC]$
```

Figure 35: Addition Benchmark on the XD1000


xd1000@xd1000:~/MulTestImpulseC
File Edit View Terminal Tabs Help

```
Multiplication Test
----------------
the first few values of array A are -0.000018  -0.737892  -0.685094  -0.668255

the first few values of array B are 0.000025  0.417937  0.580835  0.596803
Running reference (CPU)...
time: 1.309139s
-0.000000 -0.308392 0.475301 -0.045296
0.098622 0.092441 0.285439 -0.010353
0.063628 -0.056691 0.399867 0.010100
0.476261 0.033126 -0.397927 -0.398816
Running test (FPGA)...
time: 3.811710s
speedup: 0.343452x performance of CPU

-0.000000 -0.308392 0.475301 -0.045296
0.098622 0.092441 0.285439 -0.010353
-0.112097 0.224390 -0.196181 0.533163
0.161254 0.291691 -0.131106 -0.350936
0.063628 -0.056691 0.399867 0.010100
0.476261 0.033126 -0.397927 -0.398816
Checking result ...
Passed.
[xd1000@xd1000 MulTestImpulseC]$
```

Figure 36: Multiplication Benchmark on the XD1000

Table 15 shows the results of ten runs of the addition benchmark with one thousand iterations. On one occasion the CPU and FPGA required considerably more time to run the calculation; however, the results varied from 2.8 to 3.5 (CPU times faster than FPGA) without this outlier. Figure 37 displays these results graphically with each run represented by the blue line and the average shown in red.

Addition 2^17 numbers 1e-8					
run	CPU	FPGA	FPGA Faster	CPU faster	
1	1.41	4.1	0.344	2.9069767	
2	1.39	4.17	0.344	2.9069767	
3	5.38	8	0.673	1.4858841	
4	1.49	4.23	0.352	2.8409091	
5	1.34	4.38	0.305	3.2786885	
6	1.48	4.52	0.328	3.0487805	
7	1.48	4.19	0.352	2.8409091	
8	1.28	4.41	0.289	3.4602076	
9	1.2	4.18	0.288	3.4722222	
10	1.25	4.18	0.299	3.3444816	
	1.77	**4.636**	**0.3574**	**2.9586036**	**Ave**

Table 15: Addition Benchmark Test (10 runs)

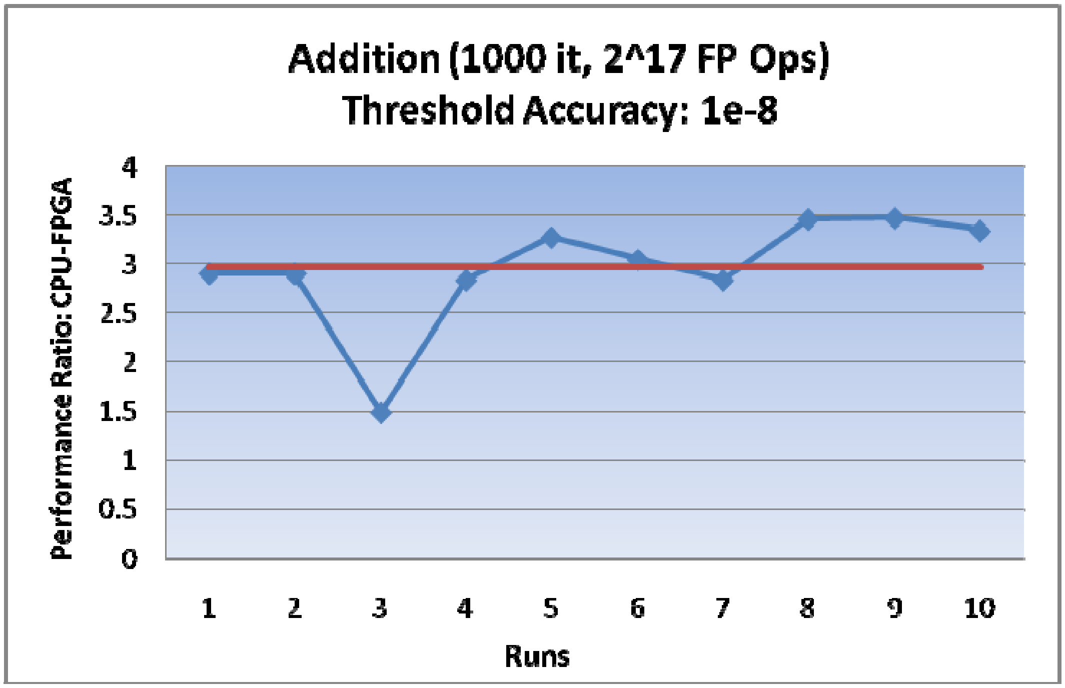


Figure 37: Addition Benchmark Results, each run (red) and average (blue)

Table 16 shows the results of ten runs of the multiplication benchmark, while Figure 38 graphically represents these data.

Multiplication 2^17 numbers 1e-8					
run	CPU	FPGA	FPGA Faster	CPU faster	
1	1.26	3.69	0.341	2.9325513	
2	1.33	6.81	0.195	5.1282051	
3	1.27	3.77	0.337	2.9673591	
4	1.24	3.79	0.327	3.058104	
5	1.21	3.78	0.321	3.1152648	
6	1.09	3.8	0.288	3.4722222	
7	1.22	3.78	0.323	3.0959752	
8	1.13	4.12	0.274	3.649635	
9	1.26	3.77	0.335	2.9850746	
10	1.1	3.81	0.289	3.4602076	
	1.211	**4.112**	**0.303**	**3.3864599**	**Ave**

Table 16: Multiplication Benchmark Results (10 runs)

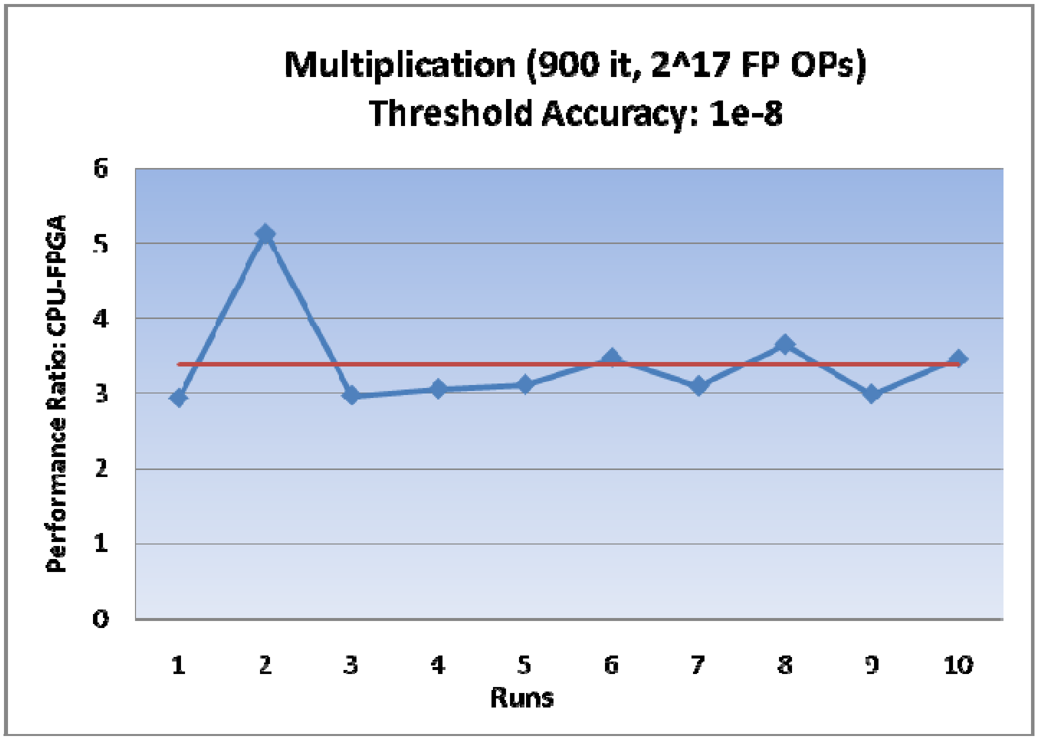
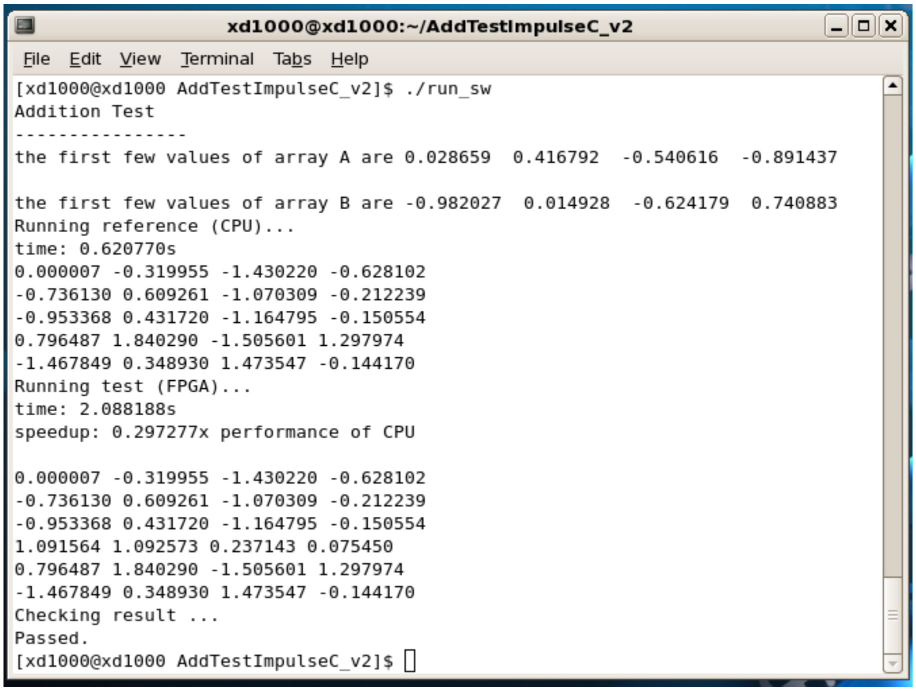


Figure 38: Multiplication Benchmark Results, runs (blue) and average (red)

Based on these findings, we altered the addition code in an attempt to increase speed. To do this, we halved the data arrays and read one entirely into the FPGA SRAM before calculations began. Unfortunately, due to a pipeline problem, the results were incorrect. After removing the pipeline, the resulting program ran no faster, but did produce accurate results. A screenshot of this program is provided in Figure 39.



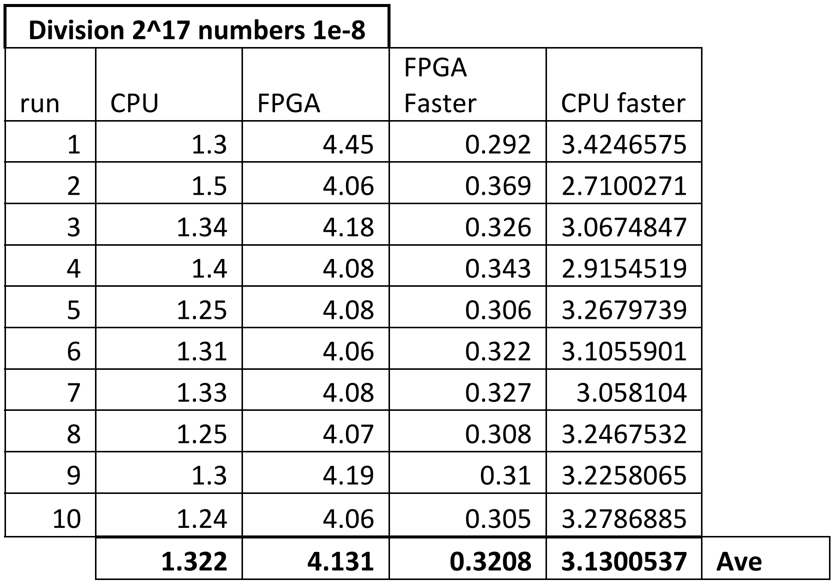
```
[xd1000@xd1000 AddTestImpulseC_v2]$ ./run_sw
Addition Test
----------------
the first few values of array A are 0.028659  0.416792  -0.540616  -0.891437

the first few values of array B are -0.982027  0.014928  -0.624179  0.740883
Running reference (CPU)...
time: 0.620770s
0.000007 -0.319955 -1.430220 -0.628102
-0.736130 0.609261 -1.070309 -0.212239
-0.953368 0.431720 -1.164795 -0.150554
0.796487 1.840290 -1.505601 1.297974
-1.467849 0.348930 1.473547 -0.144170
Running test (FPGA)...
time: 2.088188s
speedup: 0.297277x performance of CPU

0.000007 -0.319955 -1.430220 -0.628102
-0.736130 0.609261 -1.070309 -0.212239
-0.953368 0.431720 -1.164795 -0.150554
1.091564 1.092573 0.237143 0.075450
0.796487 1.840290 -1.505601 1.297974
-1.467849 0.348930 1.473547 -0.144170
Checking result ...
Passed.
[xd1000@xd1000 AddTestImpulseC_v2]$ 
```

Figure 39: Addition Test Version 2

It is possible that going into the program and counting clock cycles for operations would find a problem with the pipeline that could then be fixed. The pipeline option could then be reinserted, providing a performance boost. Overall, however, using the FPGA for simple arithmetic does not seem viable.

4.3.3 Division

Following the same procedure as in the multiplication and addition benchmarks, we created another program to test division. The division block was more complicated and required us to decrease the number of simultaneous operations in hardware from 16 to eight. This meant that the I/O had to be changed slightly as well. After making these changes and implementing the design, we found that division required more time on the FPGA than the CPU by about three times with an accuracy of up to 1e-8 threshold relative error. This is a similar finding to that of the addition and multiplication benchmarks.



Division 2^17 numbers 1e-8					
run	CPU	FPGA	FPGA Faster	CPU faster	
1	1.3	4.45	0.292	3.4246575	
2	1.5	4.06	0.369	2.7100271	
3	1.34	4.18	0.326	3.0674847	
4	1.4	4.08	0.343	2.9154519	
5	1.25	4.08	0.306	3.2679739	
6	1.31	4.06	0.322	3.1055901	
7	1.33	4.08	0.327	3.058104	
8	1.25	4.07	0.308	3.2467532	
9	1.3	4.19	0.31	3.2258065	
10	1.24	4.06	0.305	3.2786885	
	1.322	**4.131**	**0.3208**	**3.1300537**	**Ave**

Table 17: Division Benchmark Results (10 runs)

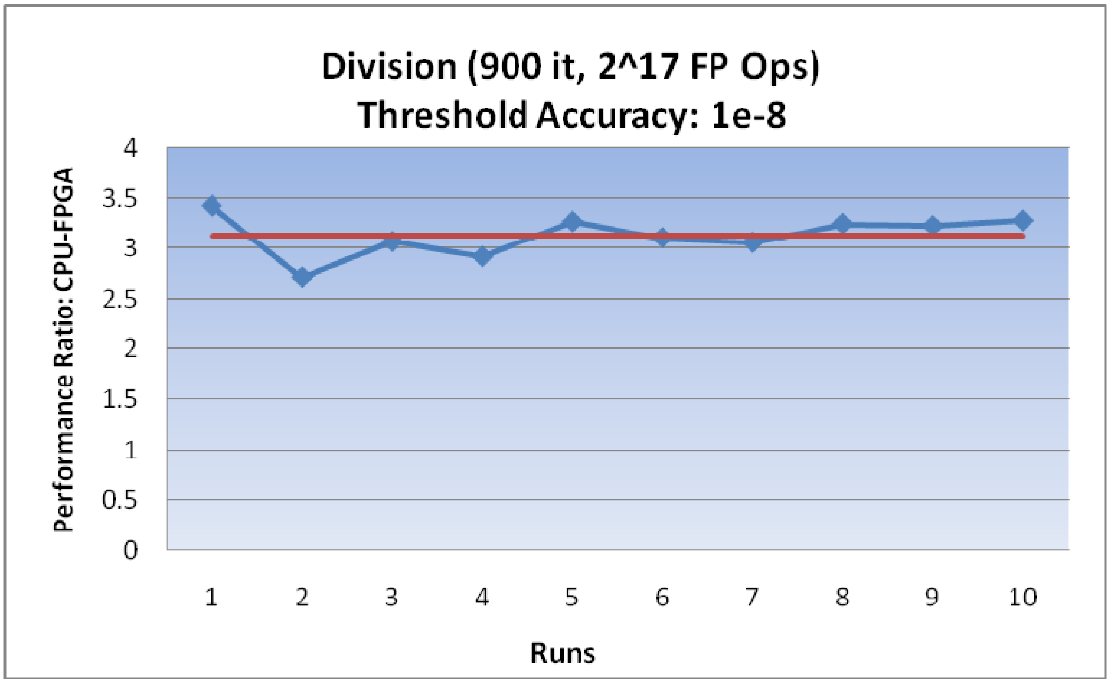


Figure 40: Division Benchmark Results, each run (blue) and average (red)

Table 17 shows the results from ten runs of the division benchmark and the averages of these runs. Figure 40 displays these results graphically, with the blue line representing each run and the red line being the average.

4.3.4 Square Root and Transcendental Functions

After completing the benchmark tests for basic arithmetic, we continued our research by designing several transcendental function benchmark tests. We decided to first implement a square root function, however we ran into several problems during its implementation. Developing trigonometric and logarithmic functions were difficult to implement as well.

Very few of the math functions listed in the math.h library exist in the hardware libraries of CoDeveloper. For example, changing the basic arithmetic benchmarks to use the sqrt() function would not work. In order to implement a square root, we decided to use VHDL code. After adding the VHDL code to the project and including a #PRAGMA CO

IMPLEMENTATION directive, we found that the code simulated properly. We ran into problems during the export process, however, because the VHDL code had to use a wrapper to interface with CoDeveloper. This wrapper would have taken considerable time to develop, so we moved on implement transcendental functions.

Next, we attempted to implement a sine function. Instead of implementing a VHDL design, we decided to use the #PRAGMA CO PRIMITIVE directive to implement a C code primitive, which CoDeveloper would then translate into HDL. We chose a simple Taylor expansion for this purpose. After adding a 5th order Taylor series to the program and altering the I/O slightly so that it would take in one array instead of two, we found that the simulation ran well and was able to accurately determine sine up to 1e-7 threshold relative error. The hardware generation, export, and programming continued without error, except that we needed to reduce simultaneous operations to two in order to fit it on the FPGA. When we tested the program, however, the resulting data from the FPGA was a series of zeros. After contacting Impulse to determine the problem, we were told that removing the #PRAGMA CO PIPELINE might fix the problem. Changing this and altering the design to use eight simultaneous sine blocks resulted in a working program in hardware; however, the timing on the sine function was still quite slow, averaging 6.2 times slower on the FPGA than the CPU.

Sine (2^17) 300 iterations					
run	CPU	FPGA	FPGA Faster	CPU faster	
1	1.95	12.08	0.162	6.1728395	
2	1.91	12.18	0.157	6.3694268	
3	1.91	12.06	0.158	6.3291139	
4	2	12.07	0.166	6.0240964	
5	1.94	12.3	0.158	6.3291139	
6	1.97	12.06	0.164	6.097561	
7	1.9	12.44	0.153	6.5359477	
8	1.99	12.05	0.165	6.0606061	
9	1.95	12.23	0.159	6.2893082	
10	2.04	12.14	0.168	5.952381	
	1.956	**12.161**	**0.161**	**6.2160394**	**Ave**

Table 18: Sine Benchmark Results

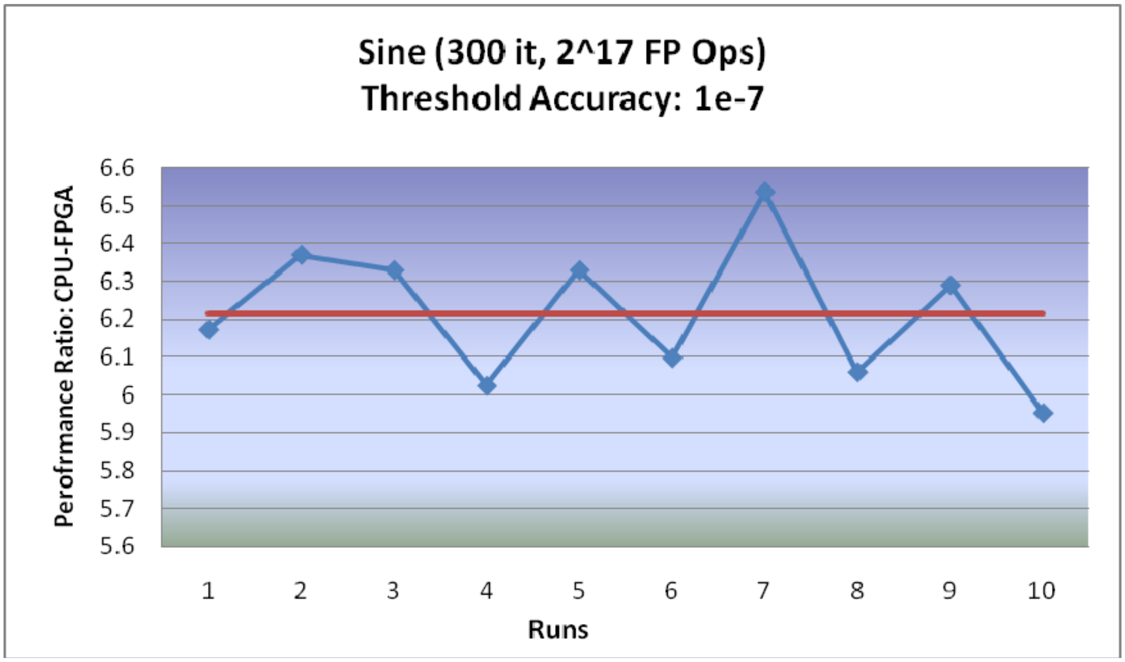


Figure 41: Sine Benchmark Results, Results per Run (blue) and Average (red)

Following the same procedure, we also implemented a cosine function, which required more terms for an accuracy of 1e-5 (two orders of magnitude lower than sine). The results of the cosine benchmark were worse than sine, averaging ten times slower on the FPGA than CPU. This result is likely due to the additional terms used. Figure 41 and Figure 42 illustrate the results of the sine and cosine benchmarks. After successfully implementing cosine without a pipeline, we added the pipeline directive to test for improvements. The project was successfully implemented, but did not result in any noticeable speedup.

Cosine (2^17) 300 iterations					
run	CPU	FPGA	FPGA Faster	CPU faster	
1	1.67	16.92	0.0988	10.121457	
2	1.76	16.8	0.105	9.5238095	
3	2.09	16.74	0.125	8	
4	1.68	16.85	0.0996	10.040161	
5	1.67	16.85	0.0991	10.090817	
6	1.7	16.8	0.101	9.9009901	
7	1.69	16.86	0.1	10	
8	1.72	16.79	0.102	9.8039216	
9	1.68	17.87	0.0997	10.03009	
10	1.69	21.97	0.077	12.987013	
	1.735	**17.445**	**0.10072**	**10.049826**	**Ave**

Table 19: Cosine Benchmark Results

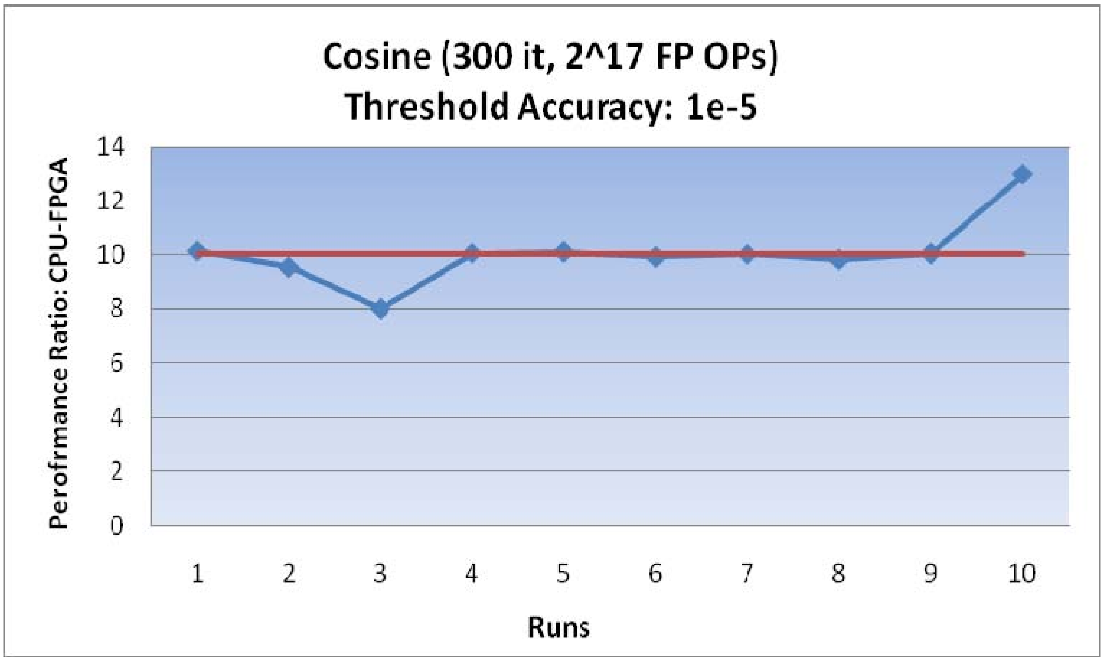


Figure 42: Cosine Function Benchmark, Times per Run (red) and Average (blue)

We also ran into problems when we developed the natural logarithm function. This function was considerably more complicated than the sine function, since it used three primitives: one to separate the exponent, one to separate the mantissa, and the other to calculate the logarithm given these inputs. After building the hardware and software, the XD1000 would hang up during the FPGA calculation. This problem was not able to be resolved by the end of the project, even after removing the pipeline and trying numerous other fixes.

Although we attempted to develop more complex programs that might be able to harness the capabilities of the XD1000, the problems during development did not allow us to make many alterations. Frequently just getting the FPGA to complete the calculation was considered a success. Perhaps with more experience we would be able to better optimize these functions; however, our findings indicate that only a limited number of mathematical operations fall into the viable niche of FPGAs.

4.3.5 Impulse Provided European Options Example

At the end of this project, Impulse Accelerated Technologies provided a European Options Monte Carlo example that boasted a ten times increase in speed over the CPU. Although the time to analyze this project was limited, we tested it with the XD1000 and determined some initial observations.

The initial version of the simulation did not work properly on our system. The result of the calculations was the FPGA determining the future value of the option at approximately $8e20 with a standard deviation and standard error of NaN. After reporting this problem, Impulse responded that the number of simulations needed to be increased to 4096. After making this change, the results seemed appropriate; however, this problem illustrates how sensitive development on the XD1000 can be.

After fixing this problem, the value of the option from the FPGA and CPU were reasonable; however, the time to compute varied widely on the FPGA. During the first ten iterations, the FPGA recorded times from 0.0086 to 0.551 seconds, a difference of 64 times. The processor, on the other hand, varied from 0.176 to 0.276 seconds. The overall result was a maximum increase of speed on the FPGA from 30 times to a speedup on the CPU of three times. The average of these runs was about an 11.5 times speedup using the FPGA. The wide variations of time were a concern and seemed to flatten out as the simulation ran, settling on much faster times for the FPGA. Table 20 lists the results per run of this model, while Figure 43 and Figure 44 illustrate the results per run and average in FPGA times faster and CPU times faster, respectively.

While this example did help illustrate the potential for FPGAs in financial HPC, it did raise a couple of concerns. First, the unpredictability of the example could easily be an issue in the financial world. If, for instance, using a cluster to perform financial analyses typically takes

12 hours, an FPGA accelerator with an expected speedup of between 30 and 0.32 times would yield a range of between 24 minutes and 37.5 hours. This difference could severely impact an investment bank. The other concern was accuracy. While investigating the example, we noted that during the simulation, it was necessary to find the exponential (e^x) of a floating point number. This process was repeated many times during the simulation; however, the hardware exponential function only calculated the integer portion of the exponential. This could very easily detract from the accuracy of the results, especially in exponential functions, which have such high rates of change. The larger the inputs become, the more the hardware function will underestimate this value. While this analysis is preliminary and needs more time to be done thoroughly, a couple areas of concern were addressed.

Euro Opt Sim, 4096 simulation					
run	CPU	FPGA	FPGA Faster	CPU faster	
1	0.193	0.231	0.83549784	1.1968912	
2	0.276	0.0686	4.02332362	0.2485507	
3	0.263	0.00861	30.5458769	0.0327376	
4	0.203	0.0727	2.79229711	0.3581281	
5	0.176	0.551	0.31941924	3.1306818	
6	0.193	0.0949	2.0337197	0.4917098	
7	0.191	0.216	0.88425926	1.1308901	
8	0.195	0.00862	22.6218097	0.0442051	
9	0.19	0.00861	22.0673635	0.0453158	
10	0.245	0.00862	28.4222738	0.0351837	
	0.2125	**0.126866**	**11.4545841**	**0.6714294**	**Ave**

Table 20: European Options Simulation Results

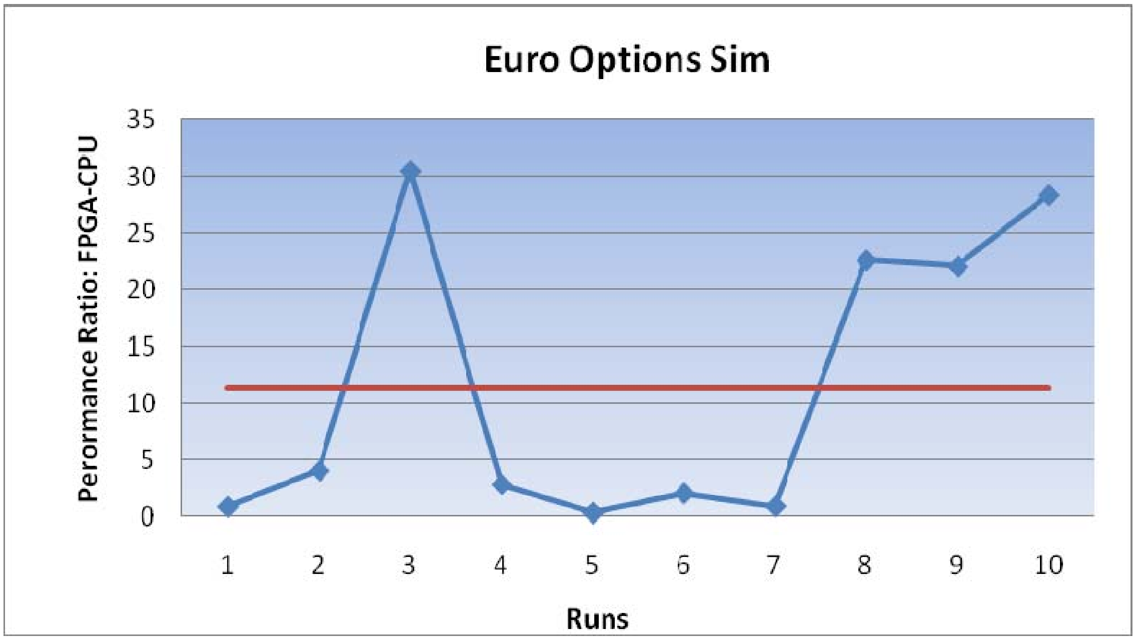


Figure 43: European Options Simulation (FPGA Times Faster), Ind Runs (blue) and Average (red)

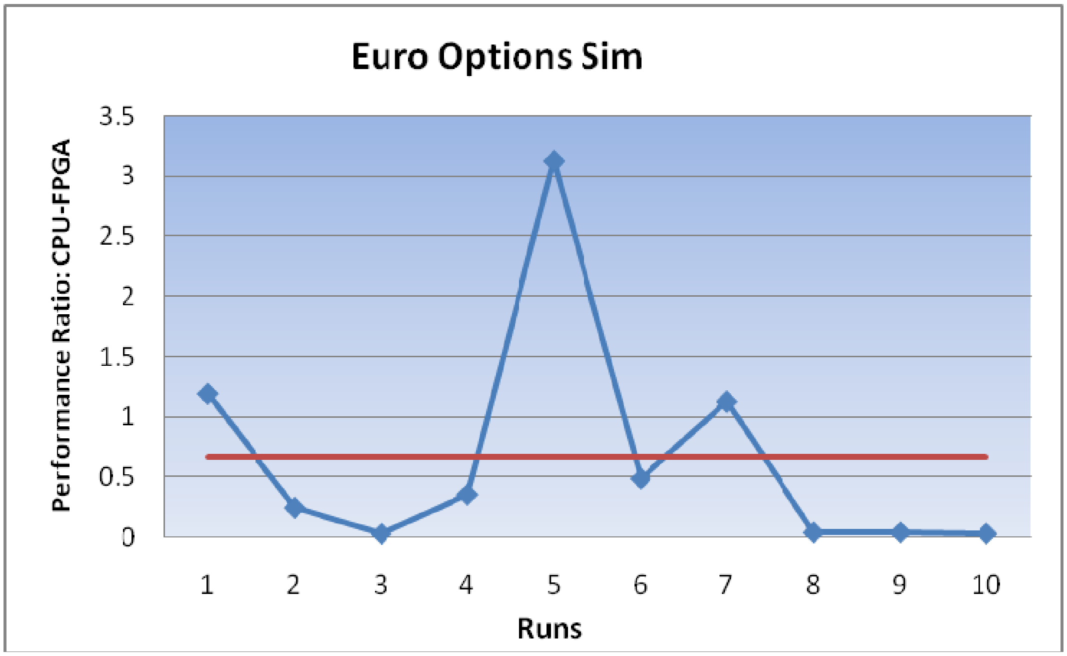


Figure 44: European Options Simulation (CPU Times Faster), Ind Runs (blue) and Average (red)

4.4 Conclusions

Current processors are very capable of computing floating point operations. Using an FPGA to increase throughput of these functions could be viable; however, the data gathered in this project illustrates the challenges of this process. Development on FPGAs is difficult, requiring considerably more time to program and debug. The currently available tools, while useful, require experience and support to utilize. Simple arithmetic on large data sets is impractical on an FPGA and typically requires three times the processing time of a CPU.

Certain applications, like matrix multiplication fall into the niche of FPGAs. Small data sets and easily parallelized operations can be effective. One of the main issues to consider is how many operations per datum are being executed. For instance, in the case of our addition benchmark, the FPGA loaded two 2^{17} number arrays to do 2^{17} operations, or one operation per two data. The matrix multiplication example, however, did 512*128*512*2 or about 67 million operations on two 512 by 128 matrices, which corresponds to 512 operations per datum. The nature of a matrix multiplication operation using the same data for many operations minimizes the I/O issues between the CPU and FPGA, making it much more viable. While investigating FPGA coprocessors, developers need to take into account the current limitations of the technology and development tools. The overall results of our benchmark tests are represented by Figure 45.

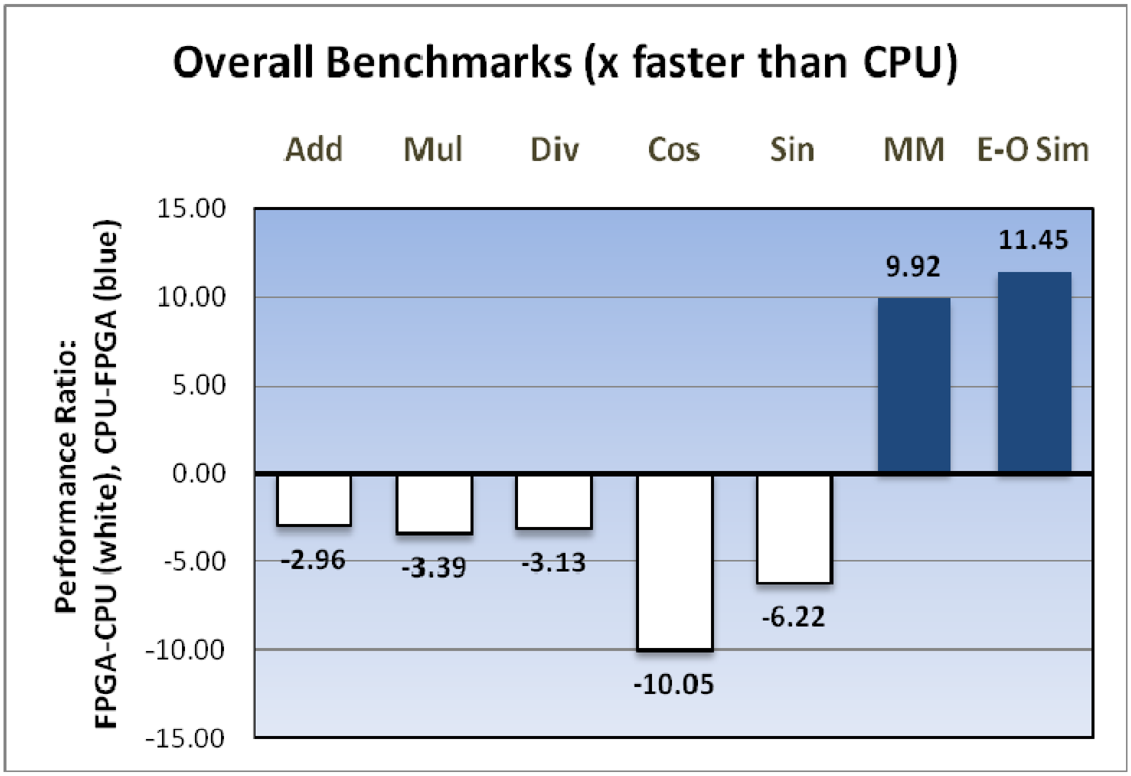


Figure 45: Overall Results (FPGA times faster than CPU)

5.0 RECOMMENDATIONS

Developing on the XD1000 system was time consuming even for simple applications. The communication and attention to detail that is currently required to implement an FPGA coprocessor application is considerable. Based on these experiences, we compiled a list of recommendations for the future research of this technology.

1. Developers should wait until a stable release of Impluse C CoDeveloper is available. Much of this project was built on alpha versions of CoDeveloper 3.0, which without the frequent need for support would be an excellent tool for development on the XD1000 system. Developers with a stable version of this software would be able to focus their time on researching FPGA capabilities, rather than the capabilities of the development software itself. Rather than waiting for a stable version of CoDeveloper, searching for an alternative for development on the system could yield a better approach. In our case, learning the drivers and developing custom VHDL and C code to interface with the XD1000 was not practical; however, future developments could make it easier to implement VHDL into the XD1000 without the need for Impulse C.
2. A subsequent project group should first familiarize themselves with Impulse C using our examples. Altering the benchmarks should provide a good idea of the limitations and advantages of Impulse C, while allowing the individual to begin to develop an understanding of optimization on FPGAs. Once familiar with the tools, we recommend a review of our benchmarks to examine possible improvements. The limited time of this project and problems with the development tools did not allow us to explore these optimizations in enough detail. Following this review, improving the transcendental benchmarks, including trigonometry, logarithm, and exponentiation as well as a

successful implementation of a financial model would be helpful in evaluating the capabilities of FPGA coprocessors. Other applications could also run well on the FPGA. Researching and testing other easily parallelized applications could yield excellent results. This research would serve as a continuation of the research completed during this project.

3. Future research should include an analysis of the Impulse C provided European Options model for accuracy and possible improvements. Since companies like Impulse Accelerated Technologies have vested interests in FPGA development, it is possible that their programs reflect this bias and while the model may be much faster, could sacrifice accuracy. A detailed evaluation of this model would be valuable to understand FPGA viability in financial applications.

4. FPGA technology has proven to be capable of improving mathematical operations; however, the currently available development tools and lack of libraries can make this process difficult. While attempting to implement a simple square root function in Impulse C, we discovered that there were no built in libraries for doing this on our system. In this case, one is forced to implement an external HDL file, or trust the C-to-HDL compiler to convert a C approximation into HDL. During this project, the compiler was not always able to do these conversions properly and so the only alternative was using an external HDL file; however, doing so requires developing a wrapper to interface with CoDeveloper, which can be a time-consuming process. Future researchers should prepare themselves for the lack of libraries by familiarizing themselves with the steps necessary do to this. This ability would allow the developers to implement virtually any VHDL or Verilog code, utilizing CoDeveloper to do the communication via the

HyperTransport bus. This would allow the developers to test a wide range of functions without the burden of the C-to-HDL compiler or library limitations.

5. FPGA technology is improving rapidly – in many ways faster than processors. It is very possible that FPGAs in a few years will be much larger and easier to program. Future researchers should familiarize themselves with the currently available technology in order to keep updated on the evolution of FPGAs. Even though the research provided in this project was inconclusive about the current viability of FPGA technology as coprocessors, improvements could make it the way of the future.

6.0 WORKS CITED

[1] J. Choy. (September 10, 2007). *conference call*

[2] XtremeData inc. (2007), XD1000 development system. [Online]. *2007(11/19),* pp. 1. Available: http://www.xtremedatainc.com/xd1000_devsys.html

[3] Intel inc. Moore's law, the future. [Online]. *2007(10/24),* pp. 1. Available: http://www.intel.com/technology/mooreslaw/index.htm

[4] E. Feretic. (2007), Trading in the fast lane. *AMD Accelerate* [Online]. *(Summer 2007),* pp. 54. Available: http://enterprise.amd.com/us-en/AMD-Business/Multimedia-Center/AMD-Accelerate/Trading-in-the-Fast-Lane.aspx

[5] Altera inc., "Accelerating high-performance computing with FPGAs," Altera inc, 2007.

[6] XtremeData inc. (2006), FPGA acceleration in HPC: A case study in financial analytics. XtremeData inc., [Online]. Available: http://www.xtremedatainc.com/pdf/FPGA_Acceleration_in_HPC.pdf

[7] Federal Deposit Insurance Corporation. FDIC mission, vision, and values. [Online]. *2007(10/24),* pp. 1. Available: http://www.fdic.gov/about/mission/index.html

[8] S. Maidanov. Monte carlo european options pricing implementation using various industry library solutions. Intel, inc., [Online]. Available: http://www.intel.com/cd/ids/developer/asmo-na/eng/columns/61383.htm

[9] K. Rubash. A study of option pricing models. [Online]. *2007(10/25),* pp. 8. Available: http://bradley.bradley.edu/~arr/bsm/pg07.html

[10] Merriam-Webster Inc. (2007), Merriam-webster's online dictionary. [Online]. *2007(10/25),* pp. 1. Available: http://www.merriam-webster.com/

[11] K. Yee. Information and stock prices: A simple introduction. Columbia University, [New York Public Library].

[12] P. D. Cretien. (2006), Comparing option pricing models. *Futures* [New York Public Library]. *(September 2006),* pp. 38.

[13] University of Southern California. Monte carlo basics. [Online]. Available: http://cacs.usc.edu/education/phys516/01-1MCbasics.pdf

[14] S. Savage. (2000, October 8, 2000). The flaw of averages. *San Jose Mercury News* [Online]. Available: http://www.stanford.edu/~savage/faculty/savage/Flaw%20of%20averages.pdf

[15] S. L. Graham, M. Snir and C. A. Patterson. (2005), *Getting Up to Speed, the Future of Supercomputing.* (1st ed.)

[16] B. Randell. (1980), The colossus. Academic Press Inc., [Online]. Available: http://www.cs.ncl.ac.uk/research/pubs/books/papers/133.pdf

[17] S. Gilheany. Evolution of intel microprocessors: 1971-2007. [Online]. Available: http://www.cs.rpi.edu/~chrisc/COURSES/CSCI-4250/SPRING-2004/slides/cpu.pdf

[18] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. [Online]. Available: http://www.cs.wisc.edu/multifacet/papers/tr1593_amdahl_multicore.pdf

[19] S. Browne, J. Dongarra and A. Trefethen. (2001), Numerical libraries and tools for scalable parallel cluster computing. *International Journal of High Performance Computing* [Online]. *(15),* pp. 175. Available: http://icl.cs.utk.edu/news_pub/submissions/libraries-tools1.pdf

[20] Top500.org. *2007*Available: http://www.top500.org/

[21] HP inc. (2007), Using infiniband for high performance computing. HP inc., [Online]. Available: h20000.www2.hp.com/bc/docs/support/SupportManual/c00593119/c00593119.pdf

[22] M. Zwolinski, *Digital System Design with VHDL.* ,2nd ed.Dorchester: Prentice Hall, 2004, pp. 368.

[23] A. Jakoby and C. Schindelhauer. (2001, December 15, 2001). Albert-ludwigs-universitat freiburg. [Online]. *2007(10/29),* pp. 1. Available: http://cone.informatik.uni-freiburg.de/pubs/fpga-add.pdf

[24] Xilinx. Our history. [Online]. *2007(10/29),* pp. 1. Available: http://www.xilinx.com/company/history.htm#begin

[25] J. Kaczynski. (2004), The challenges of modern FPGA design verification. *FPGA and Structured ASIC* [Online]. Available: http://www.fpgajournal.com/articles/20040727_aldec.htm

[26] B. Zeidman. (2006, March 21, 2006). All about FPGAs. [Online]. *2007(10/29),* pp. 1. Available: http://www.pldesignline.com/183701630;jsessionid=ETQ1VABSQDHD4QSNDLRSKH0CJUNN2JVN?pgno=1

[27] Xilinx inc. (2007, FPGA and CPLD solutions from xilinx. [Online]. *2007(10/30),* Available: http://www.xilinx.com/

[28] EASIC. FPGA flexibility meets ASIC performances. [Online]. *2007(10/29),* pp. 1. Available: http://www.easic.com/index.php?p=fpga_flexibility_meets_asic_performance

[29] B. H. Fletcher. FPGA embedded processors. Presented at FPGA Embedded Processors. [Online]. Available: http://www.xilinx.com/products/design_resources/proc_central/resource/ETP-367paper.pdf.

[30] Anonymous (2006), The promise of FPGA engineering. [Online]. *2007(October 30, 2007),* Available: http://www.timelogic.com/technology_fpga.html

[31] R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin and C. Kitchen. (2006, An overview of FPGAs and FPGA programming. Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Daresbury, Warrington, Cheshire, UK.

[32] Doulos Ltd. (2007), A brief history of VHDL. [Online]. *2007(October 30, 2007),* pp. 1. Available: http://www.doulos.com/knowhow/vhdl_designers_guide/

[33] D. J. Smith. (1996), HDL basic training: Top-down chip design using verilog and VHDL. [Online]. Available: http://www.edn.com/archives/1996/102496/df_04.htm

[34] Doulos Ltd. (2007), A brief history of verilog (2005-2007). [Online]. *2007(October 30, 2007),* pp. 1. Available: http://www.doulos.com/knowhow/verilog_designers_guide

[35] J. Evans. High level modeling tools. [Online]. Available: http://cas.web.cern.ch/cas/Sweden-2007/Lectures/Web-versions/Evans.pdf

[36] Impulse - Accelerated Technology. (2007), From C to FPGA. [Online]. *2007(October 30, 2007),* pp. 1. Available: http://www.impulse-support.com/C_to_fpga.htm

[37] Intel inc. (2006), Standards & initiatives. Intel inc., [Online]. Available: http://www.intel.com/standards/floatingpoint.pdf

[38] S. Hollasch. (2005, February 24, 2005). IEEE standard 754 floating point numbers. [Online]. Available: http://steve.hollasch.net/cgindex/coding/ieeefloat.html

[39] D. Goldberg, "What every computer scientist shoud know about floating point arithmetic," Association for Computing Machinery, Inc, 1991.

[40] K. S. Miller. (1999, 01/12/1999). Floating point arithmetic. [Online]. *2007(10/30),* pp. 1. Available: http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/arith.flpt.html

[41] J. Harrison, T. Kubaska, S. Story and P. Tang, "The Computation of Transcendental Functions on the IA-64," *Intel Technology Journal Q4,* pp. 1-1-2; 5-6, 1999.

[42] A. Binstock. (2001), Pentium 4 processor faster at math. DevX inc., [Online]. Available: http://www.intel.com/cd/ids/developer/asmo-na/eng/20214.htm?page=1

[43] S. Paschalakis and P. Lee. Double precision floating point arithmetic on FPGAs. Presented at 2003 IEEE International Conference. [IEEE Xplore]. Available: http://ieeexplore.ieee.org/

[44] N. Shirazi, A. Walters and P. Athanas, *Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines.* IEEE Symposium on FPGA's for Custom Computing Machines, 1995, pp. 155-162.

[45] AMD inc. (2007), Software optimization guide for AMD family 10h processors. AMD inc., [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf

[46] G. Torres. (2007, September 12, 2007). How the memory cache works. [Online]. *2007(11/3),* pp. 9. Available: http://www.hardwaresecrets.com/article/481/4

[47] J. Zelenski and N. Parlante. (2007), Computer architecture: Take 1. [Online]. Available: www.stanford.edu/class/cs107/handouts/13-Computer-Architecture.pdf

[48] N. Dodge. (September 2007). Lecture #8: Registers, counters, and other latch-based circuits. [Online]. Available: www.utd.edu/~dodge/EE2310/Lec8.pdf

[49] J. Stokes. (2000, March 21, 2000). SIMD architectures. [Online]. *2007(11/6),* pp. 6. Available: http://arstechnica.com/articles/paedia/cpu/simd.ars/1

[50] AMD inc. (2007, September 2007). AMD64 architecture programmer's manual, volume 1: Application programming. AMD inc., [Online]. Available: www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf

[51] Intel inc. (2007, Intel 64 and IA-32 architectures optimization reference manual. Intel, inc., [Online]. Available: http://www.intel.com/design/processor/manuals/248966.pdf

[52] Intel inc. (2007, August 2007). Intel 64 and IA-32 architectures software developer's manual, volume 1: Basic architecture. Intel, inc., [Online]. Available: www.intel.com/design/processor/manuals/253668.pdf

[53] Altera inc. Statix III - the world's fastest FPGAs. [Online]. *2007(11/7),* pp. 1. Available: http://www.altera.com/products/devices/stratix3/overview/architecture/performance/st3-performance.html

[54] R. Bodenner. Impulse C tutorial: DGEMM for the XtremeData XD1000.

[55] R. Bristow-Johnson. Digital signal processing using computers. [Online]. Available: http//groups.google.com/group/dsp/msg/8ba6bd6fe2474876

[56] E. Trexel, Personal communication Dec 4.

[57] D. Pellerin. Personal communication, Email, Dec 4.