Variable Precision Floating-Point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms

A Thesis Presented

by

Xiaojun Wang

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the field of

Computer Engineering

Northeastern University

Boston, Massachusetts

December 2007

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Variable Precision Floating-Point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms.

Author: Xiaojun Wang.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Doctor of Philosophy Degree

Thesis Advisor: Miriam Leeser	Date
Thesis Committee Member: Jennifer Dy	Date
Thesis Committee Member: Xinping Zhu	Date
Department Chair: Ali Abur	Date

Graduate School Notified of Acceptance:

Director of the Graduate School: Prof. Yaman Yener	Date

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Variable Precision Floating-Point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms.

Author: Xiaojun Wang.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Doctor of Philosophy Degree

Thesis Advisor: Miriam Leeser	Date
Thesis Committee Member: Jennifer Dy	Date
Thesis Committee Member: Xinping Zhu	Date
Department Chair: Ali Abur	Date

Graduate School Notified of Acceptance:

Director of the Graduate School: Prof. Yaman Yener	Date

Copy Deposited in Library:

Reference Librarian	Date

Abstract

Field Programmable Gate Arrays (FPGAs) are frequently used to accelerate signal and image processing algorithms due to their flexibility, relatively low cost, high performance and fast time to market. For those applications where the data has large dynamic range, floating-point arithmetic is desirable due to the inherent limitations of fixed-point arithmetic. Moreover, optimal reconfigurable hardware implementations may require the use of arbitrary floating-point formats that do not necessarily conform to IEEE specified sizes in order to make the best use of available hardware resources.

Division and square root are important operators in many digital signal processing (DSP) applications including matrix inversion, vector normalization, and Cholesky decomposition. We present variable precision floating-point divide and square root implementations on FPGAs. The floating-point divide and square root operators support many different floating-point formats including IEEE standard formats. Both modules demonstrate a good tradeoff between area, latency and throughput. They are also fully pipelined to aid the designer in implementing fast, complex, and pipelined designs.

To demonstrate the usefulness of the floating-point divide and square root operators, two applications are presented. First, we use floating-point divide to implement the mean updating step for K-means clustering in FPGA hardware, allowing the entire application to run on the FPGA. This frees the host to work on other tasks concurrently with K-means clustering, thus allowing the user to exploit the coarse-grained parallelism available in the system.

The second application is QR decomposition using Givens rotations. QR decomposition is a key step in many DSP applications including sonar beamforming, channel equalization, and 3G wireless communication. Our implementation uses a truly two dimensional systolic array architecture so latency scales well for larger matrices. Unlike previous works that avoid divide and square root operations by using special operations such as CORDIC (COordinate Rotation by DIgital Computer) or special number systems such as the logarithmic number system (LNS), we implement the Givens rotation algorithm using our floating-point divide and square root. The QR module is fully pipelined with a throughput of over 130MHz for the IEEE single-precision floating-point format.

Acknowledgements

I would like to start off by acknowledging the help and support from several people, who made this work possible. I am first and most grateful to my advisor, Dr. Miriam Leeser, for her precious guidance and support over the four years that we have worked together. I would like to express my sincere gratitude to her for providing me the right direction, constructive criticism, and supportive mentoring. I especially want to thank her for her frequent review of my work, for her encouragement when I was stuck, for her tolerance of my often forgotten ”spell check”, her patient editing of my English writing, and for providing us a nice kitchen so that I can eat anytime (which really stimulates my brain during my work). It is not only the knowledge that I have obtained from her, but more importantly her perceptive vision of research that I have benefited a lot from. All of these have been inspiring me throughout my doctoral study to become an independent researcher, an objective thinker, and a problem solver. It is a great honor for me to work with her.

I also want to thank Dr. Jennifer Dy and Dr. Xinping Zhu for agreeing to serve on my dissertation committee, for their inspirational suggestions, insightful comments, and careful review on this research. I want to thank Dr. Dy for her initial advice with the K-mean clustering algorithm. I also want to thank Dr. Zhu for his generous help with my

curriculum vitae. I am very thankful for their kindness, support and cooperation, especially for their flexibility for accommodating a tight dissertation proposal defense schedule. I want to express sincere thanks to my previous advisor, Dr. Brent Nelson, who enlightened me on reconfigurable computing. I am greatly indebted to Dr. Nelson for his patient guidance on every step I had made, for his encouragement during my first presentation at a conference when I was extremely nervous, and his very understanding of international students. Indeed, I respect him as a great mentor, an exemplary scholar, and a friend for lifetime.

Along with these mentors, I am fortunate to work with a group of great people at Northeastern University Reconfigurable Computing Laboratory. I want to thank them for sharing with me their invaluable research ideas, as well as their snacks during some drowsy afternoons and late nights in the lab. It is my great pleasure to work with them. I want to show my special thanks to Haiqian Yu for her help when I started in the lab, her friendship and her support. I am truly grateful to have such a wonderful friend.

Last but not least, I am most grateful to my parents for their unfailing support, encouragement and love in my education and my life. I want to thank them for always trying their best to support me both financially and emotionally, for believing in me when I am in doubts, and for bearing with me while I am a very busy student forever while they wondered when I am going to graduate eventually. No achievement in my life could ever have happened without their love. I dedicate this dissertation to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the applications of reconfigurable hardware is to accelerate algorithms which are initially implemented in software that runs on a general purpose processor. The prime candidates are those algorithms that are highly regular and parallel including image and signal processing applications as well as scientific applications. Such applications typically contain a high number of arithmetic operations. The acceleration is achieved via the fine-grained architecture of reconfigurable hardware that supports custom datapaths of fixed-point or floating-point arithmetic. For further control over the custom datapath, designers want to be able to optimize the bitwidth of each signal of each operation through the whole design for a specific application. Thus, support for arbitrary data formats within the same application is needed. Arbitrary fixed-point formats are not difficult to implement and are in common use. However, due to the inherent complexity of the floating-point representation, arbitrary floating-point formats are harder to implement, albeit no less desirable. Minimizing the bitwdith of each signal allows more parallelism, and makes possible fast circuitry with smaller area and latency as well as lower power dissipation.

1.1 Overview

Division and square root are among the most complicated floating-point operations. Compared to other operations, floating-point divide and square root hardware modules are much larger and have longer latency. As a result, algorithms that require floating-point divide or square root have frequently been avoided. Although some work has been done on floating-point division and square root implementations on FPGAs, most of them implement division and square root with slow or large components that are less suitable for real applications than the add and multiply components. As the performance gap widens between addition/subtraction/multiplication and division/square root, many signal and image processing algorithms involving floating-point operations have been rewritten to avoid the use of division and square root. Alternative implementations of division and square root are also frequently employed. One approach is to use logarithmic arithmetic to transform divide and square root into subtract and shift; another is to use CORIDC (Coordinate Rotation by Digital Computer). Alternatively, the computational part excluding floating-point divide and square root can be accelerated on an FPGA, but the floating-point division and square root is still implemented on the host processor. It is not the relatively infrequent use of division and square root that makes them less studied, but the complexity of their algorithms and the corresponding design challenges, as well as the limited resources of early FPGAs that make them difficult to implement. Now with the rapid increase of FPGA density and speed, as well as the emergence of the new architectures of modern FPGAs that include embedded multipliers and on-chip BlockRAMs, it is becoming possible to implement high performance floating-point division and square root and to use them in real-time hardware signal and image processing applications.

1.2 Contributions

We have extended the variable precision floating-point (VFloat) library from Northeastern University Reconfigurable Computing Laboratory [76] by implementing parameterized floating-point divide and square root. We have explored the applications that can take advantage of floating-point divide and square root to achieve significant performance improvement using reconfigurable computing. The divide and square root presented in this dissertation differ from previously published FPGA implementations. They are well suited to implementation on modern FPGAs because they use a mix of small table lookup and multipliers, which can allow for good utilization of the embedded multipliers and embedded RAMs of state-of-the-art FPGA architectures. They are based on non-iterative algorithms and can be easily pipelined. Compared to many other works that are limited to their own custom floating-point formats, our divider and square root can both be parameterized for optimal bitwidth just as all other hardware modules in the NU VFloat library can.

The contributions of this work include:

- Floating-point divide (`fp_div`) and floating-point square root (`fp_sqrt`) modules are based on Taylor Series Expansion algorithms and use the embedded multipliers and on-chip BlockRAMs of state-of-the-art FPGAs. These algorithms are particularly well-suited for implementation on FPGAs such as the Altera Stratix and Xilinx Virtex family devices. The results show the designs exhibit a tradeoff of area, latency and throughput.

- Both the floating-point divide and square root support general flexible formats including IEEE standard formats. They can be customized to allow optimization for bitwdith of both exponent and mantissa, providing the most area efficient, high performance and low power hardware solutions.

- Both the floating-point divide and square root use non-iterative algorithms as opposed to many other implementations that use iterative or bit serial division and square root algorithms. They are fully pipelined modules; thus they can be easily fit into a pipeline of a larger design without affecting the throughput of the whole design.

- We demonstrate the use of our variable precision floating-point divide for the K-means clustering algorithm applied to multispectral satellite images using a Mercury VantageRT Reconfigurable Computing Board [73] with a Xilinx Virtex-II XC2V6000 FPGA. It is the first implementation that has the complete K-means algorithm done in FPGA hardware, relieving the host to work on other tasks concurrently with K-means clustering. The results show that the hardware implementation achieves a speedup of over 2150x for core computation time and about 11x for total run-time including data transfer time.

- We demonstrate the use of our variable precision floating-point divide and square root for a systolic array based QR decomposition using Givens Rotations on a Xilinx XC5VLX220 FPGA. Compared to other implementations using special operations or special number systems, this straightforward implementation has the advantage of better portability to other systems. Using the parameterized floating-point divide and square root as well as other modules from the NU VFloat library, the input matrix size can be configured at compile-time to many different sizes. With the fully pipelined feature of both the floating-point divide and square root, a true fully pipelined two dimensional systolic array architecture was implemented. Therefore, the latency of our implementation only increases linearly with matrix size, making it scale well for larger matrices. The QR decomposition has high throughput, fast clock rate and high maximum frequency, making it suitable for high-speed FPGA implementation.

1.3 Dissertation Outline

This dissertation is organized as follows. The background related to this research is presented in Chapter 2. Chapter 3 reviews the related work in this area. The implementations of floating-point divide and square root are presented in Chapter 4. Chapter 5 and Chapter 6 discuss two digital signal and image processing applications based on our floating-point divide and square root. Finally, conclusions and future work are given in Chapter 7.

Chapter 2

Background

In this chapter, the background for variable precision floating-point arithmetic on FPGAs are presented. We discuss variable precision floating-point formats first, then introduce the variable precision floating-point (VFloat) library from Northeastern University Reconfigurable Computing Laboratory [76]. New features of state-of-the-art FPGA architectures will be presented. The specifics of the experiment setup is also introduced in this chapter.

2.1 Variable Precision Floating-Point Formats

One of the most widely used formats for representing numeric values is the fixed-point format, which derives its name from the fixed location of its radix point - the delimitation between the integer part and the fractional part. For integer values, the radix point is located to the right of the least significant digit. The simplicity and low cost of implementing the fixed-point format makes it popular for many designs. However, due to its limited range, it is unsuitable for many scientific and engineering computations which process both very large and very small numbers. Thus, floating-point number representation has been adopted to process numbers in a much wider range. Floating-point number representation [41] is the primary method used for real arithmetic in most digital systems.

A binary floating-point number is a bit string characterized by three components: a sign, a biased exponent and a mantissa, also known as the significand. The sign bit is simple: 0 denotes a positive number and 1 denotes a negative number. The biased exponent is the sum of the exponent and a constant (bias) chosen to make the biased exponent's range nonnegative. The mantissa represents the magnitude of the number. When a number is normalized, the mantissa is composed of a leading one bit to the left of its implied binary point and the fraction bits to the right. The numerical value of a normalized binary floating-point number is the signed product of its mantissa and two raised to the power of its exponent[1]. That is $(-1)^s \times 1.f \times 2^{e-BIAS}$, where s represents the sign, e represents the biased exponent, and f represents the fractional part of the mantissa. The most widely used floating-point format is ANSI/IEEE Std 754-1985 [50] for radix 2. Two basic IEEE floating-point formats are depicted in Figure 2.1. One is the 32-bit single-precision format, and the other is the 64-bit double-precision format.

There exists a large set of floating-point number formats in which the word width and precision differ from the standard. It is desirable to support variable precision floating-point operations where both exponent and mantissa bitwidths can vary. All hardware modules in the NU VFloat library are parameterized by word width and number of fractional bits.

2.2 FPGA Based Reconfigurable Hardware

Reconfigurable computing uses hardware that can adapt at the logic level to specific application requirements and system architectures. It is a nice compromise between Application

[1]We assume a base of 2

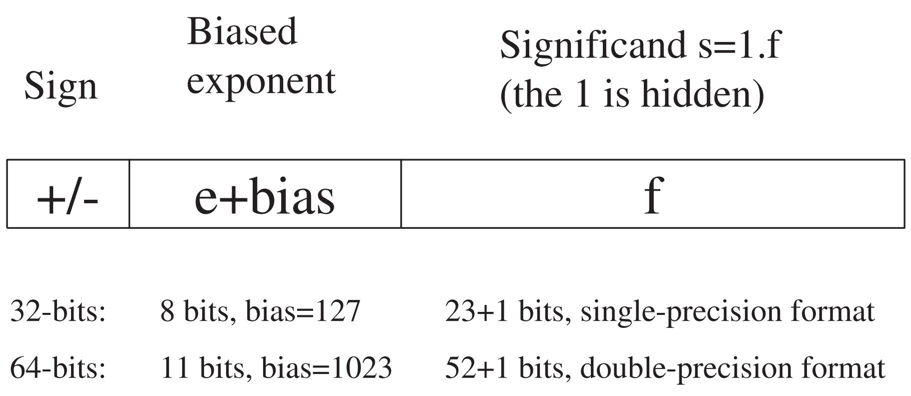

Sign
Biased exponent
Significand s=1.f (the 1 is hidden)
+/-
e+bias
f
32-bits:
8 bits, bias=127
23+1 bits, single-precision format
64-bits:
11 bits, bias=1023
52+1 bits, double-precision format

Figure 2.1: The ANSI/IEEE Standard Floating-Point Number Representation Formats

Specific Integrated Circuits (ASICs) and general purpose processors. General Purpose processors are very flexible but not tuned to the application and therefore usually have poor performance for the specific application. ASICs solve this problem with very high performance for the specific application. However they are inflexible, have high cost per chip, high Non-Recurring Engineering (NRE) cost, and a long design cycle. Reconfigurable hardware can greatly reduce the NRE cost, has fast time to market and pretty high performance. The cost of a single reconfigurable chip is much cheaper than for an ASIC, and so is the cost of tools for FPGAs.

Field Programmable Gate Arrays (FPGAs) are one of the most popular types of reconfigurable hardware. FPGAs are integrated circuits which contain large, two-dimensional arrays of homogeneous logical elements (LE). The logic element connects to a switch matrix to access the configurable distributed routing network. The LEs support both combinational logic and memory storage. Each FPGA logic element (LE) contains a look-up table (LUT) (usually 4-input) and a one-bit flip-flop (FF). They can be programmed to perform any user-defined Boolean function and their functionality varies from one FPGA architecture to another. The output of each LE can be selected by an output multiplexer to be either registered or unregistered. In addition to LEs, FPGAs also have interconnect and input/output blocks.

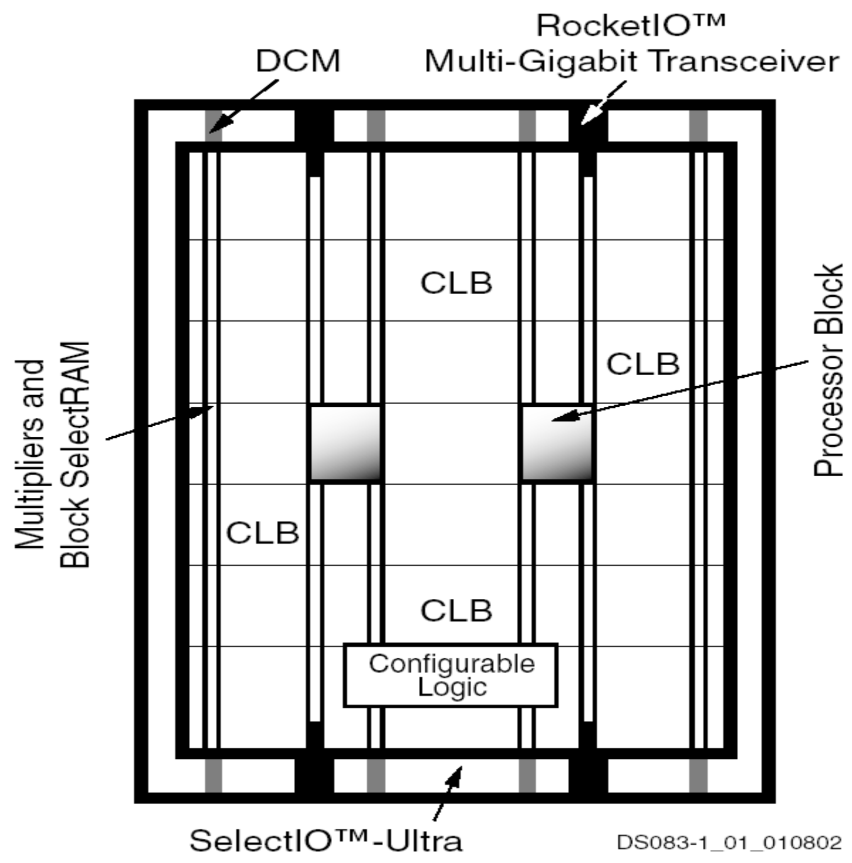


Figure 2.2: Xilinx Virtex-II Pro FPGA (Figure Courtesy of Xilinx [106])

Modern FPGAs are getting denser and faster. A single FPGA chip has millions of gate equivalents and a clock speed greater than 100MHz. Other than general purpose programmable resources like I/O, logic blocks and interconnect, FPGAs also include larger embedded logic blocks like memories, multipliers, DSPs, and processor cores. Figure 2.2 and Figure 2.3 show two modern FPGA architectures. The first is the Xilinx Virtex-II Pro FPGA shown in Figure 2.2. It has two embedded processors in the middle, embedded

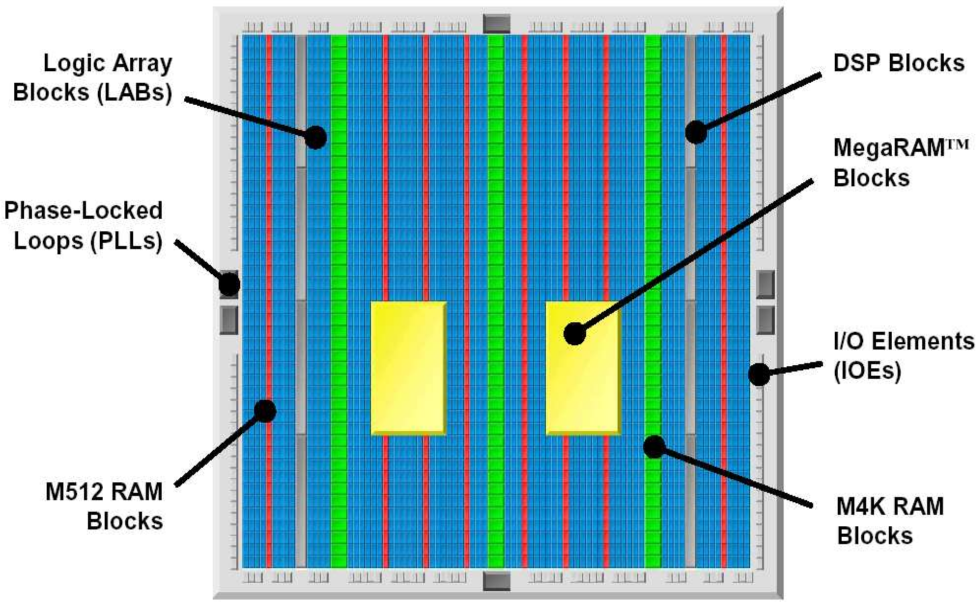


Figure 2.3: Altera Stratix FPGA (Figure Courtesy of Altera [6])

multipliers and Block SelectRAMs organized in vertical columns of blocks on the left and right of configurable logic block (CLB) arrays. The other is the Altera Stratix FPGA shown in Figure 2.3. It has three types of embedded RAMs: MegaRAM blocks in the middle, M512 RAM blocks, and M4K RAM blocks organized in vertical columns of blocks on the left and right of logic Array Blocks (LABs). There are also DSP blocks on the Stratix FPGA.

2.2.1 Embedded Multiplier Blocks and Embedded DSP Blocks

The Xilinx Virtex-II family has embedded 18-bit signed multipliers. They are really 17x17 bit multipliers with one bit of sign logic. Figure 2.4 is a diagram of the Xilinx Virtex family embedded multipliers. Altera Stratix devices have embedded unsigned 9x9 bit multipliers implemented in embedded DSP blocks. The DSP blocks also contain adders, subtracters,

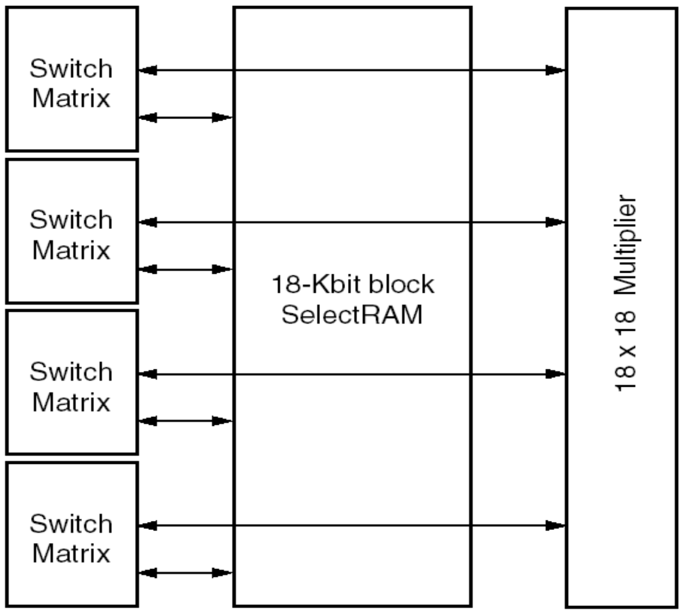


Figure 2.4: Xilinx Embedded Multipliers (Figure Courtesy of Xilinx [106])

accumulators, and registered inputs that can become shift registers.

2.2.2 Embedded RAM Blocks

Xilinx Virtex-II family has embedded Block SelectRAMs. They are fully synchronous, 18Kb dual-port RAM arranged in columns of CLB fabric. Altera Straix has three types of TriMatrix Dual-Port RAMs of different sizes - M512 RAM blocks, M4K RAM blocks, and MegaRAM blocks. Figure 2.5 shows the Altera TriMatrix Dual-Prot M512 RAM block.

2.2.3 Embedded Processor Cores

Modern FPGAs also support embedded processor cores. Some are softcore processors such as the Nios soft processor from Altera and the Microblaze from Xilinx; others are hardcore processors. The Xilinx Virtex-II Pro FPGA has up to 4 IBM PowerPCs on one

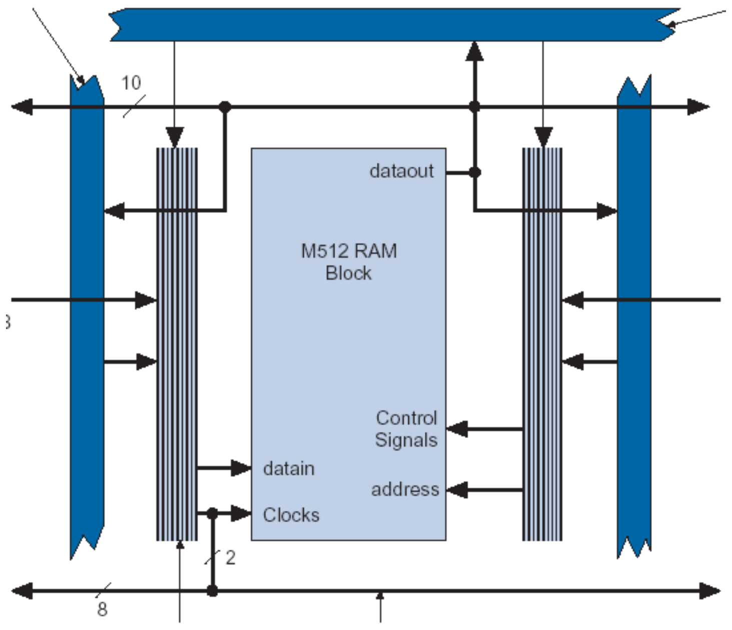


Figure 2.5: Altera TriMatrix Dual-Port RAM: M512 (Figure Courtesy of Altera [6])

chip.

All these large embedded logic blocks make more efficient use of on-chip FPGA resources. For example, embedded RAMs makes much more efficient use of area for memory. However, they can also waste on-chip resources if they are not being used. Making good use of these embedded on-chip resources now becomes an important design consideration. In this work, we will explore the utilization of embedded multipliers and embedded BlockRAMs on Xilinx Virtex FPGAs in implementing floating-point operations.

2.3 VFloat Library

Variable precision divide and square root [104] in this work are incorporated into the previous work by Pavle Belanović on the original Northeastern University parameterized floating-point library [16, 15]. Table 2.1 shows the name, function and latency (in clock cycles) of all hardware modules in the NU VFloat library. The components `fp_div` and `fp_sqrt` have latencies varying with the bitwidth of the floating-point format. We will see later that they have much longer latencies for wider bitwidth format than the other library modules due to their intrinsic complexity. The `fp_acc` component has a latency varying with the number of operands. The longer latency of these three components is due to the attempt to make all library modules have as similar a clock period as possible in order to achieve the highest throughput pipeline.

Table 2.1: Floating-Point Hardware Modules and Their Latency

Module	**Function**	**Latency**
FORMAT CONTROL		
`denorm`	Introduction of implied integer digit	0
`rnd_norm`	Normalizing and rounding	2
ARITHMETIC		
`fp_add/fp_sub`	Addition/Subtraction	4
`fp_mul`	Multiplication	3
`fp_div`	Division	Variable
`fp_sqrt`	Square Root	Variable
`fp_acc`	Accumulation	Variable
FORMAT CONVERSION		
`fix2float`	Unsigned fixed-point to floating-point conversion	4
`fix2float_signed`	Signed fixed-point to floating-point conversion	5
`float2fixf`	Floating-point to unsigned fixed-point conversion	4
`loat2fix_signed`	Floating-point to signed fixed-point conversion	5

All designs in the NU VFloat library are written in VHDL. The VHDL description of each module accepts the parameters *exp_bits* and *man_bits*. The *exp_bits* represents the bitwidth of the exponent and *man_bits* represents the bitwidth of the mantissa. These parameters are compile-time variables; they allow the bitwidth of each unit in the datapath to be parameterized, thus optimizing the amount and the structure of logic needed to implement a circuit. In other words, the designer has the flexibility to build a custom datapath by specifying the exact bitwidth of each datapath unit.

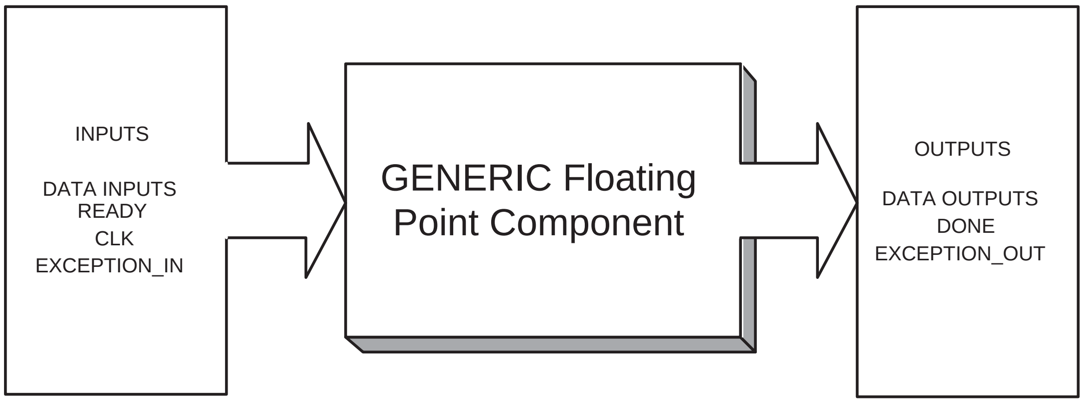


Figure 2.6: A Generic Library Component

Figure 2.6 shows a generic library component. Each component in the NU VFloat library has a READY signal and a DONE signal. When the READY signal is high, the component starts to process the input data. Upon completion, the component makes the output values available on the data outputs and sets the DONE signal to high. These handshaking signals allow the library components to be easily assembled into a pipelined datapath. The NU VFloat library also supports some error handling. Each module in the library has an EXCEPTION_IN signal and an EXCEPTION_OUT signal. An error fed into one component with the EXCEPTION_IN signal high or detected inside this component, is propagated to the EXCEPTION_OUT signal. In this manner, error signals are propagated through a pipeline along with their corresponding results.

The IEEE standard specifies four rounding modes: round to zero; round to nearest;

round to positive infinity ($+\infty$); and round to negative infinity ($-\infty$). The first two modes of rounding are supported in this work. “Round to zero” is simply truncation. It is the simplest rounding method but it may introduce significant rounding error. “Round to nearest” overcomes this problem and minimizes rounding error by assigning each value to its closest rounded value. Some values are equally close to both their higher rounded values and lower rounded values. To simplify the hardware design, we assign those values to their higher rounded values.

The key elements of the NU VFloat library include:

- The NU VFloat library supports general flexible formats which can be parameterized to accept any width operands including IEEE standard formats while most earlier work only supports either IEEE single-precision format or other non-general application-specific formats.

- Each component in the NU VFloat library has a READY signal and a DONE signal to support pipelining. All the hardware modules in the library are fully pipelined.

- The NU VFloat library has a separate denormalization unit, and a separate normalization and rounding unit supporting either ”round to zero” or ”round to nearest”. Few other works have these features. Moreover, our design also supports some error handling and exception detection.

- The NU VFloat library has components to convert from fixed-point format to floating-point format and vice versa, which supports the implementation of hybrid designs with both fixed-point and floating-point formats in a single domain.

In summary, the NU VFloat library is a superset of many previously published floating-point formats, and is general and flexible. The library is also more complete than earlier work with support for separate normalization, rounding and some error handling. In this research, we focus particularly on the implementation and use of floating-point divide and square root.

2.4 Experiment Setup

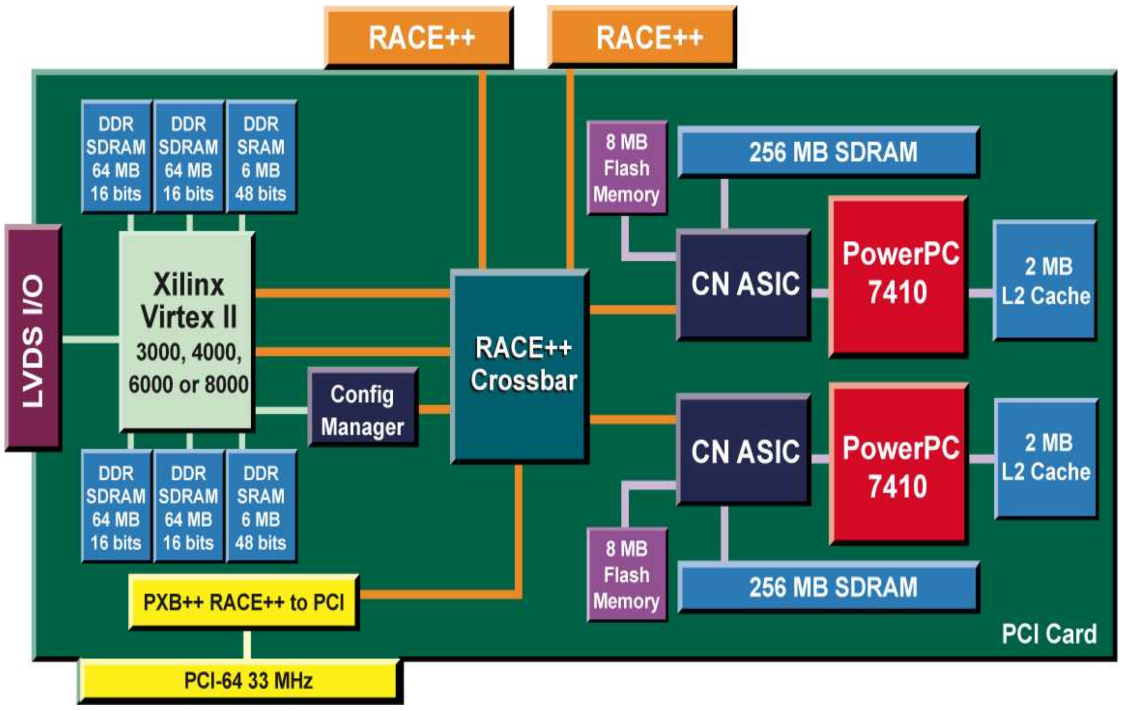


Figure 2.7: Mercury VantageRT FCN Board (Figure Courtesy of Mercury [73])

Reconfigurable computing is characterized by the use of hardware elements that have reconfigurable architectures, as opposed to general purpose computing which uses hardware elements with fixed architectures. Many reconfigurable computing systems are based on one or more FPGAs connected to a number of memory banks. All designs presented in this chapter are implemented on a VantageRT FCN board from Mercury Computer Systems [73]. Figure 2.7 shows the structure of this board, which integrates an FPGA with

PowerPC G4 microprocessors in the Mercury RACEway switch fabric architecture. Some of the main features of the VantageRT FCN board are: one Xilinx Virtex-II XC2V6000 FPGA compute node (speed grade -5); 12MB of DDR SRAM and 256MB of DDR SDRAM both running at 133MHz; two 500MHz 7410 microprocessor PowerPCs.

2.5 Summary

This chapter presented several fundamental aspects of this dissertation. After a brief introduction on the basics of floating-point formats, we focused on the variable precision features of the hardware modules in the NU VFloat library. Designs with smaller bitwidths than those IEEE standard formats require fewer hardware resources thus allow for more parallelism. Another obvious tradeoff is that given a certain total bitwidth, there are various combinations of bitwidths for exponent and mantissa. Larger exponent results in wider range while larger mantissa results in higher precision. All modules in the NU VFloat library are parameterized with variable word width, which gives the users flexibility and full control over the optimal formats for their specific application requirements.

The architectures of several embedded blocks of two modern FPGAs – Xilinx Virtex-II FPGA and Altera Stratix FPGA – were presented. The first are embedded multiplier blocks on Xilinx Virtex-II FPGAs and embedded DSP blocks on Altera Stratix FPGAs; the second are embedded RAM blocks; and the third are embedded processor core blocks. The rest of this dissertation focused on how to fully utilize these embedded blocks for the floating-point divide and square root implementations. The experiment setup with Mercury VantageRT FCN board is also discussed in this chapter. The VantageRT board has one Xilinx Virtex-II XC2V6000 FPGA with on-chip embedded blockRAMs and on-chip embedded multipliers. This setup is used in our experiment in Chapter 4 and Chapter 5. In the next chapter, related work is discussed.

Chapter 3

Related Work

This chapter covers a variety of topics related to our research. Studies from different floating-point division and square root algorithms, as well as different floating-point formats are discussed as many of these studies influenced our work. There are also several design automation techniques for bitwidth optimization that we studied. To compare with our NU VFloat library, other work focusing on floating-point libraries, as well as several commercial floating-point IP cores are also discussed.

3.1 Early Work on Floating-Point Arithmetic on FPGAs

One of the earliest studies of floating-point arithmetic on FPGAs was done in 1994 by Fagin et al. [28]. The authors implemented floating-point addition and multiplication including rounding and underflow over four Actel A1280 FPGAs [4]. Their results showed that implementing IEEE single-precision add and multiply was feasible but impractical on FPGA technology at that time. Therefore, smaller, custom floating-point formats other than IEEE standard formats were suggested. Many papers and publications were presented shortly after that expanding on this original proposal of custom floating-point formats. One work [92] built floating-point adder/subtractors, multipliers and dividers for two application-specific formats - 16 bit (1-6-9: 1 bit sign, 6 bit exponent, 9 bit mantissa) and 18 bit (1-7-10) rather than the 32 bit (1-8-23) IEEE 754 standard single-precision format [50]. Their work was mapped to Xilinx XC4010 FPGAs [108], which consists of only a 20x20 grid of configurable logic blocks (CLBs). They applied both of their special formats to 2-D FFT and FIR filter applications. The ultimate goal of these application-specific formats is to meet the high performance requirement of real-time signal processing applications on custom computing machines.

While small custom floating-point formats enable significant speedup for some applications, they can not always meet the high accuracy that many scientific algorithms require. Many works were published to study implementing IEEE single-precision floating-point units on FPGAs. The earliest single-precision floating-point division and square root on FPGAs was implemented by Louie and Ercegovac in 1993 [65, 67, 66]. They explored single-precision (24 bit mantissa) divider and square root using digit recurrence algorithms on FPGAs with a special focus on the exploration of suitable variations of the algorithms for the target FPGA characteristics and efficient mapping to the lookup table based Xilinx XC4010 FPGA [108].

Another work [64] implemented an IEEE single-precision adder and multiplier on the Altera FLEX 8000 series [7] and applied them to the gravitational n-body problem. To conserve area, the authors separated normalization from the arithmetic operations. IEEE single-precision adder and multiplier circuits supporting "round to nearest" and exception handling were presented in [62]. The floating-point adders and multipliers were used to construct multiply-accumulate units (MACs), and the multiply-accumulate units were then used to build a matrix multiplication application. Similar work on single-precision adder/subtraction, multiplication and accumulation in floating-point with 40×40 and 96×96 matrix multiplication was done by Sahin et al. [87, 39]. All the hardware modules in their work are highly pipelined and synchronization signals were introduced to synchronize the flow of data through pipelines. Similar work by Stamoulis et al. [94] presented pipelined floating-point units for addition/subtraction, multiplication and division that were used for 3D computer graphics applications in FPGAs.

These papers show that it is feasible to implement single-precision floating-point unit in a FPGA. However, for earlier FPGA chips with low densities, the floating-point units are slow and take a large amount of FPGA resources. Supporting single-precision floating-point format was costly in the earlier FPGAs. Moreover, it is impractical to make use of the floating-point units in real signal processing applications with reasonable performance.

3.2 Floating-Point Division

Floating-point division is hard to implement due to the complexity of the algorithms and the limited density of early FPGAs. After the earliest exploration of floating-point division algorithms for FPGAs by Louie and Ercegovac [65, 67], a number of studies have emerged [92, 20, 85, 54, 105, 79, 78, 100, 44, 63, 98, 22, 42]. The division algorithms in these works fall into three categories: digit recurrence algorithms such as SRT algorithm, lookup table and multiplier based algorithms, and iterative algorithms such as Newton-Rapson.

Digital recurrence is the most popular class of algorithms due to its simplicity. The most well-known digit recurrence division algorithm is the SRT algorithm named by Freiman [32] and originated by Sweeney [97], Robertson [84], and Tocher [99], who suggested the algorithm independently at approximately the same time. Many variations of the original SRT division algorithm were thoroughly studied by Ercegovac and Lang [25]. Oberman and Flynn [77] analyzed overall system performance of division and other floating-point operations. Leeser [93] discussed the design decisions of floating-point division and square root in several microprocessor architectures. The earliest implementation of single-precision floating-point divider on a Xilinx XC4010 [65, 67] used a variation of the radix-2 SRT algorithm. The first IEEE double-precision floating-point divider implemented on a Xilinx XCV1000 FPGA [79] also used the simple radix-2 digit-recurrence algorithm. This is a non-pipelined design so it is very small in area but very low in throughput with a very high latency of 60 clock cycles for double-precision division. Based on this sequential design, a pipelined design [98] was presented, which unrolls the iterations of the digit recurrence computations and optimizes the operations within each unrolled iteration. The maximum frequency of this pipelined design for double-precision floating-point divide is over 100MHz on a Xilinx XC2V6000 FPGA while consumes a modest 8% of the chip area. Another work to improve the throughput of the division via a simple digit-recurrence algorithm is based on a scalable pipeline insertion method [78]. This is also a heavily pipelined design so it has the maximum frequency of over 200MHz for a single-precision floating-point divider on a Xilinx Virtex-II XC2V1000 FPGA. Comparison of their results with two commercial floating-point IP cores [81, 75] shows that it outperforms the commercial IP cores by about half the latency while maintaining similar maximum frequency and resource utilization. Recently, Underwood [100] discussed the feasibility of supporting IEEE double precision floating-point on FPGAs. He presents add, multiply and divide components. The divider is bit serial and exhibits long latency. All the above digit recurrence algorithms are radix 2 based. Floating-point dividers based on a high radix SRT division algorithm for radix 2, 4, and 8 were compared [105]. Three implementations - a low cost iterative implementation, a low latency array implementation, and a high throughput pipelined implementation were studied. Another high radix SRT divider (radix 4) of both pipelined and non-pipelined structure [54] were implemented on Xilinx Virtex-II XC2V1000 and XC2V6000 FPGAs. The single-precision pipelined implementation takes 15 clock cycles with a clock rate of $9.6ns$ while the double-precision pipelined implementation takes 29 clock cycles with a clock rate of $12.2ns$. The double-precision divide [44] supporting most features of the IEEE Std 754 [50] is also based on digit recurrence algorithms. Extending this work, more recent work [22] support both 64-bit double-precision divide and 128-bit quad-precision divide on a Xilinx Virtex-II Pro FPGA.

Lookup table based algorithms are another class of division algorithm. A popular approach to carry out a division is to calculate the inverse of the denominator based on a lookup table, followed by multiplying that inverse by the numerator. However, the size of the table to do the inversion increases exponentially when the size of divider increases. So this inverse lookup table method is only feasible for small floating-point dividers. One work [92] built custom floating-point formats divider - 16 bits (1-6-9) and 18 bits (1-7-10) for 2-D FFT and FIR filter applications. Another work [20] built the floating-point divide with 8 bit output accuracy for a video processing application in a Xilinx VirtexE FPGA. As an extension of [92], the authors coded the inverse to reduce the required bitwidth of the following multiplication.

The third approach for division is iterative algorithms. One work using iterative algorithms to implement a floating-point divider in FPGAs was presented by Roesler and Nelson [85]. Based on the Newton-Rapson algorithm and using repeated multiplication to approximate the result, this algorithm requires only multipliers. However, a small lookup table can be added to compute the result of the first few iterations to reduce the total number of iterations. Unlike the lookup table based algorithms, larger floating-point formats beyond IEEE single-precision formats can be built. However, this approach has long latency due to the iterative nature of the algorithm. Designs that take advantage of new features of Xilinx Virtex-II FPGA - SRL16 and 18×18 embedded multipliers were explored. The results of various floating-point formats with and without using these new features were presented. Another work [42] is based on Goldschmidt's algorithm and has very low latency. The latency of their double precision divide on Altera Stratix II EP2S30 FPGA is only $82ns$. However, the algorithm is still iterative so their divide cannot be fully pipelined. For double precision, a new operation starts every 8 clock cycles instead of every clock cycle.

One recent work [63] published on a division algorithm for FPGAs employs a novel hybrid algorithm which employs Prescaling, Series expansion and Taylor expansion (PST). It is a combination of a lookup table based algorithm and an iterative algorithm using lookup tables and multipliers. This hybrid algorithm proves to be a very good balance of table lookup and arithmetic circuitry. The results on an Altera Stratix-II EP2S15F484C3 FPGA show a significant reduction in delay and power compared to a division IP core generated by Altera MegaWizard for 32-bit division. Results with and without using Stratix-II FPGA embedded DSP blocks were compared as well. However, they only did the integer division with one fixed bitwdith. It is still an iterative algorithm and the clock frequency of their 32-bit divider is pretty slow as well.

In summary, both the digit recurrence and iterative algorithms lead to long latency due to the iterative nature of the algorithm while previous lookup table and multiplier based algorithms require a huge table - the table size increases exponentially with the size of divider. To get good performance with a reasonable amount of hardware resources, a better algorithm is desired.

3.3 Floating-Point Square Root

Similar to floating-point division, floating-point square root is also very difficult to implement on FPGAs. Square root and division share a lot of common features and use similar algorithms including the non-restoring subtractive algorithm, the SRT algorithm, and the Newton-Rapson iterative algorithm. The most popular square root algorithm is also digit recurrence. After the earliest study of floating-point square root on Xilinx XC4010 FPGAs [65, 67], there have been a few other works focused on IEEE single-precision floating-point square root [59, 54, 105, 78] and double-precision square root [54, 79, 44, 98].

In 1997, Li and Chu presented an IEEE single-precision floating-point square root [59] on Xilinx XC4000 FPGA based on a non-restoring subtractive algorithm. They presented two implementations - one is a low-cost iterative implementation that uses one traditional adder/subtractor; the other is a high-throughput pipelined implementation that uses multiple adder/subtractors. In addition to the non-restoring subtractive algorithm, another digit recurrence algorithm is the SRT algorithm. Similar to divide, various floating-point formats up to IEEE single-precision square root were presented with radix 2, radix 4, and radix 8 SRT algorithm [105]. Tradeoffs of area, latency, and throughput were studied. Again, similar to divide, a scalable pipeline insertion method [78] is explored to improve the throughput of floating-point square root. The study shows results close to two commercial floating-point IP cores [81, 75] on a Xilinx XC2V1000 FPGA.

The first published 64-bit IEEE double-precision floating-point square root implementation [54] was based on a radix-4 SRT algorithm and targets Xilinx Virtex-II XC2V1000 and XC2V6000 FPGAs. The double-precision pipelined square root takes 28 clock cycles with a clock rate of $13.8ns$. Another double-precision floating-point square root using the radix-2 digit recurrence algorithm was implemented on a Xilinx XCV1000 FPGA [79]. Due to its non-pipelined nature, it has a very high latency of 59 clock cycles as well as a very low throughput. However, it takes a very small amount of FPGA resources even though it is double-precision format. In comparison, its pipelined counterpart [98] reaches the maximum frequency of about 133MHz for double-precision floating-point square root on a Xilinx XC2V6000 FPGA while consumes a modest 8% of the chip area. The double-precision square root supporting most features of IEEE Std 754 [50] was discussed [44]. The authors use the same non-restoring algorithm as the work of Li and Chu [59].

So far, all published work on floating-point square root uses digit recurrence algorithms. Therefore, they all have comparatively long latency.

3.4 General Floating-Point Format

The first application-specific floating-point format was suggested by Shirazi in 1995 [92]. Their target FPGA is a very small Xilinx XC4010 [108] with only 20x20 CLBs, so neither an IEEE single-precision multiply nor a divide can fit in one chip. A floating-point format with bitwidth less than 32 bits provides enough dynamic range and precision for their 2-D FFT and FIR filter applications. For the above two reasons, they suggested two smaller formats. The first format is 16 bits (1-6-9) for the FIR filter. They chose 16 bit data since local, 16-bit wide memories were used in their pipelined calculations. The second format is 18 bits (1-7-10) for the 2-D FFT application. Since the Xilinx 4010 processors on the Splash-2 are 36 bits wide, two 18-bit input operands can be clocked in one cycle. Satisfactory results were showed with their small custom floating-point formats. However, their floating-point format only supports these two specific formats.

To allow further fine-grained datapath configuration, a flexible floating-point format for optimizing each operation in the datapath was proposed [20]. For further optimization, conversion between fixed-point and floating-point was included to assist embedding the floating-point units in a DSP chain with fixed-point input and output data. Their format works very well for their HDTV application since many image processing algorithms including their video processing application require only 8 bits data. However, their flexible floating-point formats are not general enough to represent IEEE standard formats or other arbitrary floating-point formats. Moreover, both works [92, 20] only support floating-point formats that are smaller than 32-bit single-precision and may not be enough for the numerial stability of other scientific applications.

This line of custom floating-point format study was expanded by a lot of recent work [51, 85, 105] with the rapid development of CAD tools and hardware description language such as Handel-C [13] and JHDL [49]. Handel-C from Celoxica is a software-style language based on ANSI-C along with a compiler that compiles algorithms directly into EDIF netlists optimized for the target FPGA device. This approach aims at algorithmic level optimization. In 2001, Jaenicke and Luk [51] presented a floating-point adder and multiplier that allowed compile-time parameterization of both range and precision of the floating-point operands using the Handel-C language. Many optional features such as overflow protection, gradual underflow and rounding modes are also selectable at compile-time via the Handel-C compiler. They implemented multipliers using three floating-point formats – 16 bits, 32 bits and 40 bits on a Xilinx XCV1000 FPGA in an RC1000-PP system. A 2D Fast Hartley Transform (FHT) implemented using their parameterized FPU that showed an effective tradeoff between performance and required resources. JHDL [49] from the Configurable Computing Laboratory at Brigham Young University is a "structurally based Hardware Description Language (HDL) implemented with JAVA". A module generator coded in JHDL can take both exponent width and mantissa width as compile-time parameters. Using these module generators in JHDL, two works [85, 105] support arbitrary floating-point formats including but not limited to IEEE 754 standard format. In the first work [85], various floating-point formats up to 41 bits (1-8-32) were presented for floating-point division; while in the second work [105] various floating-point formats up to 32 bits

IEEE single-precision were presented for both floating-point division and square root.

Further expanding this line of study, various parameterized floating-point libraries have emerged [16, 54, 61, 70, 19, 44] as we will discuss in Section 3.6. Custom floating-point formats allow the bitwidth of each unit in the datapath to be optimized. Therefore, designers have fine-grained control over the datapath bitwidth based on their specific application and the value that each signal carries. This customized datapath with optimal bitwidth of each datapath unit makes area utilization more efficient and increases potential parallelism, thus achieving high performance and reducing power dissipation.

3.5 Design Automation of Bitwidth Optimization

Several CAD tools have been developed to automate bitwidth optimization of floating-point formats. One automatic tool called fp translates a floating-point algorithm to a hardware implementable fixed-point representation [58]. This software system consists of a fixed point C++ class, a profiler to determine the required precision of each signal, an optimiser to find the minimum bitwidth within required accuracy, and a compiler that outputs synthesisable VHDL code. A post-rendering 3D image warping application was successfully implemented with the VHDL code automatically generated by the fp tool from a floating-point description written in C.

A variety of other work [35, 33, 34] focuses on automatic tools for bitwidth optimization of a floating-point format. To minimize the area and maximize the speed of FPGA implementations, we want to reduce the width of each floating-point signal within the required error range. One technique suggested by Gaffar et al. [33] is based on iteratively reducing the bitwidth of the operations until nothing can be reduced any more without violating the accuracy criteria. This tool is based on a C++ operator overloaded library, and a collection of plug-in modules which interact with each other via interfaces. The tool produces the optimal size for both the mantissa and the exponent for each operation, and generates customized hardware that is smaller and faster compared with a non-optimized design. This is particularly useful for many scientific applications involving large dynamic ranges. However, the search space for optimal sizes is enormous with this method.

Another technique [35, 34] is an automatic differentiation (AD) mathematical method to analyze the sensitivity of one output variable with respect to other output variables. The required precision of the mantissa part of that floating-point output is determined by its sensitivity to the changes of other nodes. Automatic differentiation is much faster than the simulation-based iterative method discussed above with huge search space [33]. In the first work [35], the AD tool is based on a C++ library with user defined types and overloaded operators. Two applications, a DFT and a FIR filter, were built using bitwidth optimization. The area and speed of a series of implementations with different bitwidth and corresponding error range were compared. This approach only supports floating-point operations and only considers the optimization of mantissa bitwidth. As an extension to this work, a unified bitwidth optimization method for fixed-point and floating-point was developed [34]. Mantissa size is determined via a tool called BitSize that is implemented as a C++ object library similar to that in previous work [35]. In addition to precision analysis, range analysis is performed to determine the exponent size for floating-point operations or the integer bitwidth for fixed-point operations. This BitSize technique is demonstrated with four applications: ray-tracing, function approximation, finite-impulse response (FIR) filter and Discrete Cosine Transform (DCT).

In addition to bitwidth optimization of floating-point operations, Liang developed a floating-point unit generation tool [60] to automate parameter selection including but not limited to optimal bitwidth. To ease design implementation, this tool is fully integrated into ASC [72], a C++ based stream compiler tool for FPGAs developed at Bell Laboratories, and mapped to the PAM-Blox II module generation environment [71]. In addition to custom bitwidth operations, many other implementation parameters can be automatically selected by this tool. First, different FPGA features such as carry chains, LUTs, tri-state buffers and flip flops can be tuned to obtain different throughput, latency, and area values, leading to parallel and serial versions of units. Second, tradeoffs of latency, throughput and area can be explored through the selection of different algorithms. Last, optional floating-point operation features such as sign mode, normalization, rounding, and exception handling can be selected to further optimize area and performance. A wavelet filter was implemented based on the floating-point units generated via this tool and two modules, one optimized for latency and the other optimized for throughput, outperformed two commercial floating-point IP cores [75, 21].

Users of the NU VFloat library can make use of the tools presented in this section to determine the optimal bitwidth of each floating-point operation in their specific application automatically.

3.6 Other Floating-Point Libraries

In addition to the NU VFloat library [76] presented in Section 2.3, there are a few similar libraries [54, 61, 70, 19, 44]. Among all these other libraries, only two libraries [70, 19] are available on the web.

One of these libraries [61] includes floating-point adder, multiplier, divider and square root. All floating-point operations use a generalized floating-point representation with generic exponent and mantissa bit. With a fixed 7 bit exponent, various mantissa floating-point formats up to 24 bits, were implemented on a Xilinx XC2V3000 FPGA. The floating-point library was used to accelerate the smoothed particle hydrodynamics (SPH) algorithms for N-Body simulations. The parameterisable floating-point cores that first supported IEEE double-precision format [54] were implemented on Xilinx Virtex-II XC2V1000 and

XC2V6000 FPGAs. Four operations: addition, multiplication, division and square root, were implemented with a maximum frequency of about 100MHz for single-precision operations and about 75MHz for double-precision operations.

Instead of floating-point arithmetic, others [70] [19] use Logarithmic Number System (LNS) arithmetic. One work [70] is by Matousek et al. Their library includes addition/subtaction, multiplication, division, square and square-root operations, and supports both 32-bit and 20-bit precision. Detrey and Dinechin [19] developed a parameterized library of arithmetic operations – addition/subtraction, multiplication, division and square root – using LNS arithmetic. This Lyon Library [14] supports both floating-point arithmetic and Logarithmic Number System (LNS) arithmetic. Multiplication and division are as simple as addition and subtraction respectively in LNS; square root is simpler in LNS as well. But addition and subtraction are much more complicated in LNS. Comparing the Lyon library and the NU VFloat library, we see that both libraries provide a variety of variable precision floating-point components that are easily parameterizable. Both support conversion from different formats. All arithmetic operations in the Lyon library are implemented in both pure combinational and pipelined versions while the NU VFloat library provides more support for pipelining with handshaking signals as well as support for exception handling. In addition, the NU VFloat library includes an accumulator component. The divide and square root components in the Lyon library are SRT radix 4 and radix 2 respectively. These designs exhibit longer latencies than the components in the NU VFloat library.

The most recent floating-point library [44] supports most features of IEEE Std 754 including all types of numbers representable, all types of exceptions, all rounding modes, and the NaN diagnostics. Different from all the libraries described above that are parameterized for the bitwidth of exponent and/or mantissa, all hardware cores in this library including addition/subtraction, multiplication, division and square root are only IEEE double-precision.

These hardware cores are parameterized by the degree of pipelining and the level of compliance with IEEE Std 754. All the floating-point cores in this library were applied to the force and potential calculation kernels in a Molecular dynamics (MD) simulation.

3.7 Commercial Floating-Point IP Cores

There are also several commercial floating-point IP cores available [81, 21, 75, 107, 8]. One is the floating-point cores from Digital Core Design, Inc. [21]. These are IEEE single and double precision formats with floating-point add, multiply, divide, square root, compare, and IEEE-754 floating-point/integer conversion. All the cores are pipelined so that results are available at every clock for all these cores. Overflow, underflow and invalid operation detection are supported. Another library is the IEEE 754 Floating-Point Cores from Nallatech, Inc. [75]. It supports both IEEE single-precision and double-precision floating-point cores for Xilinx Virtex family FPGA products. Available functions include signed addition/subtraction, multiply, divide, square root, and IEEE-754/Integer conversion functions. Only the floating-point multiply operation uses Xilinx Virtex-II 18×18 embedded multipliers. The Nallatech floating-point cores employ their own floating-point format for internal use, with conversion blocks provided for IEEE-754 compatibility. These cores are also pre-placed using Relationally Placed Macros (RPMs). All these optimizations result in less logic and faster speed. The Nallatech floating-point cores have a fully pipelined architecture, with high throughput and long latency. For example, both the IEEE double precision divider and square root have latency of 58 clock cycles. Divide and square root run at a maximum frequency of 168MHz and 200MHz respectively.

Neither the IP cores from Digital Core Design, Inc. [21] nor those from Nallatech [75] are parameterisable for any bitwidth exponent or mantissa. Variable wordlength floating-point format is very useful to save area, increase performance and reduce power. Quixilica has variable wordlength floating-point arithmetic cores [81] including floating-point adder, multiplier, divider and square root for Xilinx Virtex, Virtex E and Virtex 2 FPGA families. All the floating-point cores are variable wordlength including IEEE single and double precision formats. Their IP cores support normalization and rounding as "round to even", exception handling including underflow detection, overflow detection,and other invalid operations such as Not a Number(NaN) and divide by zero. They are all fully pipelined to achieve the highest throughput so that a new floating-point calculation starts on each clock cycle. The latency is pretty high: 24 bit mantissa IEEE single-precision divider has a latency of 27 clock cycles with a maximum clock frequency of 142-189MHz.

Xilinx also has variable wordlength floating-point IP cores [107] that include five floating-point operations: multiply, add/subtract, divide, square root, compare and fixed-point and floating-point conversion. This Xilinx core is based on IP originally licensed from QinetiQ Ltd. [81]. They support normalization and the rounding mode of "round to nearest". All these cores have a common interface with the operation type provided. Both exponent width and fraction width can be set. Other than wordlength optimization, these cores can be customized to allow optimization for latency, rate, interface and other options. First, they allow the type and level of embedded multiplier usage to be specified. This gives the designer the flexibility to best utilize embedded multipliers for their specific application. Second, the latency between an operand input and result output for all operators (apart from the logic-assisted, double precision multipliers) can be set between zero and a maximum value that depends on the parameters chosen. In addition, the rate at which operands can be applied to the square-root and divider cores can also be specified. This allows the selection of a fully pipelined circuit (where one input can be applied on every clock cycle), or a partially pipelined circuit, thus allowing for a tradeoff between area and throughput. The maximum frequency of these cores is very high at about 200-300MHz for most of the cores up to 32-bit IEEE single-precision format. Even for 64-bit IEEE double-precision format, most of the cores have the maximum frequency close to 200MHz. However, many of them have long latency. For example, the IEEE double-precision divide and square root have a latency of 56 clock cycles. Last but not least, Xilinx floating-point IP also supports the optional inclusion of many exception handling signals such as underflow, overflow, invalid operation, divide by zero and control signals such as operation ready (RDY), chip enable (CE), etc.

3.8 Summary

In this chapter, related research has been presented. Our research is related to the fields of floating-point division and square root algorithms, custom floating-point formats, and efficient exploitation of modern FPGA resources for high performance floating-point operations. Several projects from each of these fields were examined to show how these other projects influenced our own research. While other floating-point divide and square implementations use bit serial or iterative algorithms and have long latency, non-iterative algorithms are employed in this work. Our floating-point divide and square root implementations differ from previous research in that they exhibit a good tradeoff between area, latency and throughput. They are easily pipelined to fit into a larger design. They also support the creation of custom format floating-point datapaths. In the next chapter, the implementations of our variable precision floating-point divide and square root are introduced.

Chapter 4

Floating-Point Divide and Square Root

Divide and square root are important operations in many high performance signal processing applications. In this chapter, we introduce the hardware implementation of floating-point divide and square root on a Xilinx Virtex-II FPGA. The algorithms and hardware implementations of floating-point divide and square root are presented first. Both the division [48] and square root [26] algorithms are based on Taylor series expansion, and use a combination of small table lookup and small multipliers to obtain the first few terms of the Taylor series. These algorithms are particularly well-suited for implementation on an FPGA with embedded RAM and embedded multipliers such as the Altera Stratix and Xilinx Virtex family devices. They are also non-iterative algorithms which makes them easy to fit into the pipeline of a large design along with other modules of the NU VFloat library without affecting the throughput of the whole design. Finally, we discuss experimental results for the divide and square root components. The experiment section includes two parts: the first part is the analysis of the area, latency and throughput of the floating-point divide and square root on a Xilinx XC2V6000 FPGA; the second part is the comparison of our results with that of the Xilinx LogiCORE Floating-Point Operators v3.0 [107]. In

Chapter 5 and Chapter 6, we present using these components in a larger design.

4.1 Floating-Point Divider

The divider we built follows a previously published algorithm [48]. Assume a dividend X and a divisor Y, both $2m$ bit fixed-point numbers in the range of [1,2). They can be expanded in the form:

$$X = 1 + 2^{-1}x_1 + 2^{-2}x_2 + \cdots + 2^{-(2m-1)}x_{2m-1} \tag{4.1}$$

$$Y = 1 + 2^{-1}y_1 + 2^{-2}y_2 + \cdots + 2^{-(2m-1)}y_{2m-1} \tag{4.2}$$

where $x_i, y_i \in \{0, 1\}$. The divisor Y can be further decomposed into a higher order bit part Y_h and a lower order bit part Y_l, which are defined as:

$$\begin{aligned} Y_h &= 1 + 2^{-1}y_1 + 2^{-2}y_2 + \cdots + 2^{-m}y_m && (4.3) \\ Y_l &= 2^{-(m+1)}y_{m+1} + 2^{-(m+2)}y_{(m+2)} + \\ &\quad \cdots + 2^{-(2m-1)}y_{2m-1} && (4.4) \end{aligned}$$

The range of Y_h is between 1 and $Y_{hmax}(= 2 - 2^{-m})$, and the range of Y_l is between 0 and $Y_{lmax}(= 2^{-m} - 2^{-(2m-1)})$. To calculate X/Y using the Taylor series:

$$\begin{aligned} \frac{X}{Y} &= \frac{X}{Y_h + Y_l} = \frac{X}{Y_h}(1 - \frac{Y_l}{Y_h} + \frac{Y_l^2}{Y_h^2} - \cdots) \\ &\approx X \times (Y_h - Y_l) \times \frac{1}{Y_h^2} && (4.5) \end{aligned}$$

Since $Y_h > 2^m Y_l$, the maximum fractional error in equation (4.5) is less than 2^{-2m}, or 1/2 ulp. [1] Division based on equation (4.5) is very simple, requiring two multiplications and one table-lookup for $\frac{1}{Y_h^2}$. Figure 4.1 shows a schematic representation of equation (4.5).

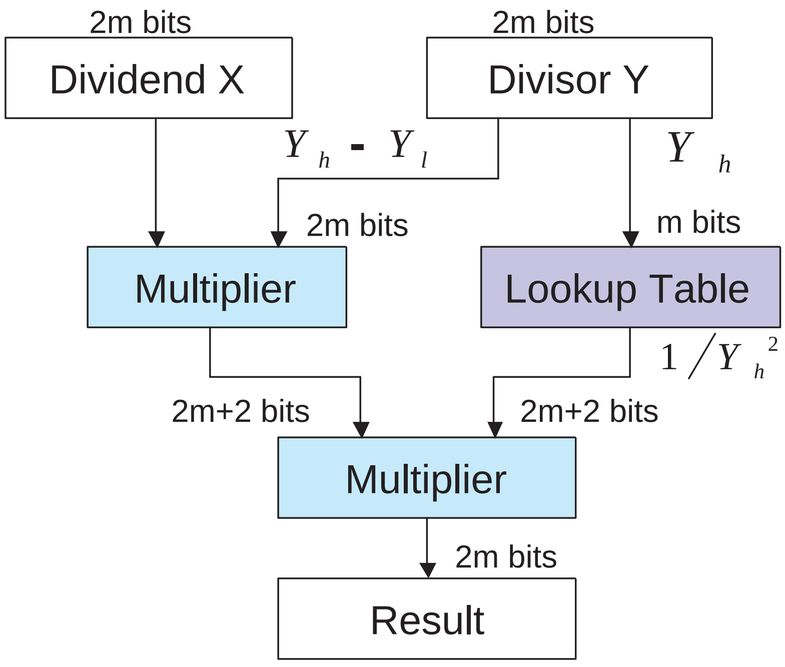


Figure 4.1: Table-Lookup Based Divider

The BlockRAMs and embedded multipliers that the Xilinx Virtex-II FPGA provides are used in our implementation for table lookup and multiplication. Table lookup via Block-RAM takes one clock cycle. The multipliers are pipelined multipliers and their latency varies with the size of the multiplier. Thus, the overall latency of the divider also varies with its size.

[1] ulp is an acronym for unit in the last place. The least significant bit of the fraction of a number in its standard representation is the last place.

Using this table-lookup based divider as the core, a floating-point divider can be implemented as:

$$\frac{(-1)^{s_1} \times m_1 \times 2^{e_1}}{(-1)^{s_2} \times m_2 \times 2^{e_2}} = (-1)^{s_1 \oplus s_2} \times (m_1/m_2) \times 2^{(e_1 - e_2)} \tag{4.6}$$

The sign of the quotient is the exclusive OR (XOR) of the signs of the input operands, the exponent of the quotient is the difference between the exponents of the input operands, and the mantissa of the quotient is the quotient of the mantissas of the input operands. The quotient is obtained using the table-lookup based divider core shown in Figure 4.1. These three parts are independent and can be implemented in parallel. Exception detection is implemented in parallel with the computation of the quotient mantissa.

4.2 Floating-Point Square Root

The square root we built is also based on a previously published algorithm [26]. It also uses lookup tables and multipliers based on Taylor series expansion, but is much more complicated than the divider. Figure 4.2 shows the three steps to complete a square root: reduction, evaluation, and post processing. Let Y be an n bit fixed-point number in the range of [1,2) and let k be an integer such that

$$\boldsymbol{k = \lceil n/4 \rceil}$$

The first step is to reduce Y to a k bit number A ($-2^{-k} < A < 2^{-k}$) such that the square root of Y can be easily obtained from a function of $f(A)$. This $f(A)$ will be evaluated in the next step. A can be obtained as:

$$\begin{aligned} A &= Y \times \hat{R} - 1 && (4.7) \\ \hat{R} &= 1/Y^{(k)} && (4.8) \end{aligned}$$

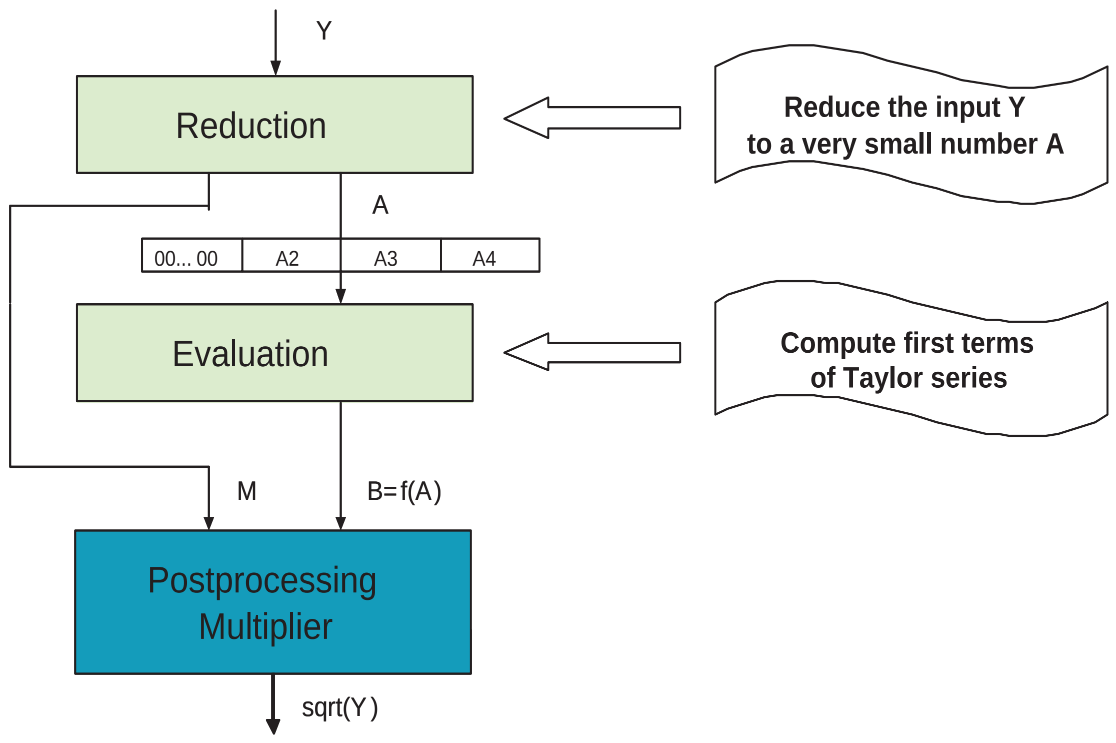

Y
Reduction
Reduce the input Y to a very small number A
A
00... 00
A2
A3
A4
Evaluation
Compute first terms of Taylor series
M
B=f(A)
Postprocessing Multiplier
sqrt(Y)

Figure 4.2: Table-Lookup Based Square Root

where $Y^{(k)}$ is Y truncated to its k^{th} bit. A $4k$ bit number M is also generated:

$$M = 1/\sqrt{\hat{R}} \tag{4.9}$$

This M will be used in the post processing step to correct the final result due to the reduction step. Figure 4.3 shows the data flow of this reduction step. Two table-lookups with k address bits each are required for $\hat{R}$ and M respectively and one $(k + 1) \times 4k$ multiplier is required for A.

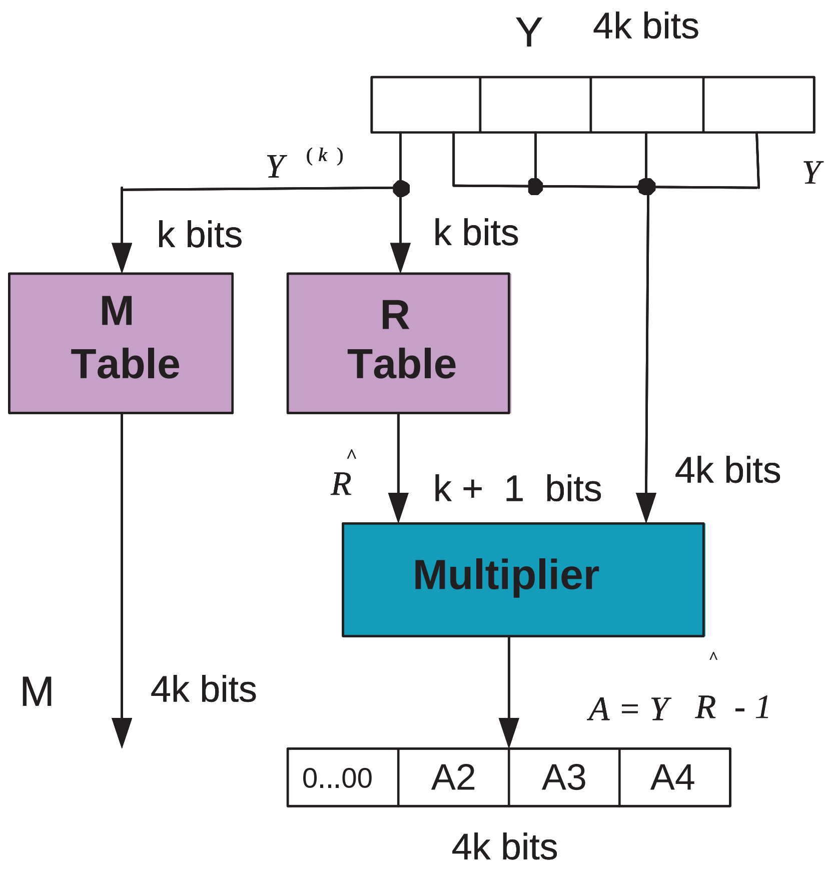


Figure 4.3: Reduction Step of Square Root

The second step is evaluation. Since $-2^{-k} < A < 2^{-k}$, A has the form:

$$A = A_2 z^2 + A_3 z^3 + A_4 z^4 + \cdots \tag{4.10}$$

where $z = 2^{-k}$ and $|A_i| = 2^k - 1$. Our goal is to compute the approximation $B = \sqrt{1+A}$. Using Taylor series expansion:

$$B = \sqrt{1+A} \approx 1 + \frac{A}{2} - \frac{1}{8}A_2^2 z^4 - \frac{1}{4}A_2 A_3 z^5 + \frac{1}{16}A_2^3 z^6 \tag{4.11}$$

This requires only two $k \times k$ multipliers for A_2^2 and A_2A_3, and one $2k \times k$ multiplier for A_2^3 as shown in Figure 4.4.

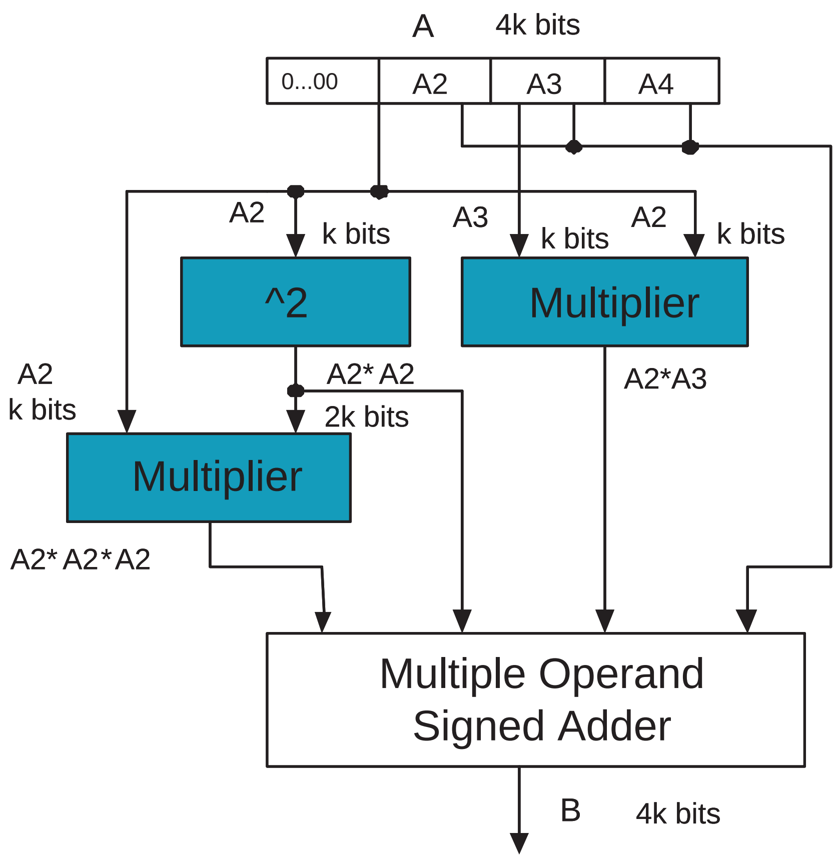


Figure 4.4: Evaluation Step of Square Root

At the end, we need to correct the result B by multiplying it by M from the reduction step. This requires one multiplier and is done in the post processing step. The overall error of all three steps is analyzed as 2.39×2^{-4k} [26].

The implementation of square root is similar to that of division. Lookup tables are built via BlockRAMs with one clock cycle latency. Multipliers are built via embedded multipliers which are pipelined with various latencies depending on their size. Thus the overall latency of the square root also varies with its size.

Using this table-lookup based square root as the core, a floating-point square root can be built. The sign of the input operand should always be 0. Otherwise, an error is raised. The exponent of the square root is half the exponent of the input operand if the input exponent is even. If the input exponent is odd, the exponent of the square root is half the input exponent minus one and the mantissa of the square root needs to be multiplied by a constant factor $\sqrt{2}$. We combine this constant multiplication in the computation of M in the reduction step. In other words, we have one extra table for $\sqrt{2}M$. Whether M or $\sqrt{2}M$ is selected depends on whether the input exponent is even or odd. The mantissa of the square root is the square root of the mantissa of the input operand, which can be obtained using the table-lookup based square root core outlined in Figure 4.2. Similar to other components, the exponent and mantissa are computed in parallel since they are independent. Exception handling is implemented in parallel as well.

4.3 Experimental Results

This section presents results for both floating-point divide (*fp_div*) and floating-point square root (*fp_sqrt*) for a wide range of floating-point formats including IEEE single-precision and double-precision formats. All our designs are written in VHDL and synthesized using Synplify Pro 8.0. The bitstream is generated using Xilinx ISE 6.3i and downloaded to the Virtex-II XC2V6000 FPGA on the Mercury Computer Systems VantageRT FCN board as shown in Figure 2.7 (see section 2.4). The floating-point divide and square root only uses the FPGA on this board. All the hardware modules have been tested both in simulation and in hardware. For simulation, we use a testbench that tests a combination of random numbers and corner cases for various bitwdith floating-point formats. For testing on the reconfigurable board, we send the same set of numbers through the PCI bus to the FPGA board, do the computation on the board, then read the results back to the host PC.

4.3.1 Experimental Results for Floating-Point Divider and Square Root

Table 4.1: Area, Latency and Throughput for Floating-Point Division

Floating Point Format	8(2,5)	16(4,11)	24(6,17)	32(8,23)	40(10,29)
slices	66 (1%)	115 (1%)	281 (1%)	361 (1%)	617 (1%)
BlockRAMs	1 (1%)	1 (1%)	1 (1%)	7 (4%)	62 (43%)
embedded multipliers	2 (1%)	2 (1%)	8 (5%)	8 (5%)	8 (5%)
clock period (ns)	4.9	6.8	7.8	7.7	8.0
maximum frequency (MHz)	202	146	129	129	125
latency (clock cycle)	10	10	14	14	14
latency (ns)	49	68	109	108	112
throughput (MFLOPS)	202	146	129	129	125

Table 4.2: Area, Latency and Throughput for Floating-Point Square Root

Floating Point Format	8(2,5)	16(4,11)	24(6,17)	32(8,23)	48(9,38)	64(11,52)
slices	85 (1%)	172 (1%)	308 (1%)	351 (1%)	779 (2%)	1572 (4%)
BlockRAMs	3 (2%)	3 (2%)	3 (2%)	3 (2%)	13 (9%)	116 (80%)
embedded multipliers	4 (2%)	7 (4%)	9 (6%)	9 (6%)	16 (11%)	24 (16%)
clock period (ns)	6.1	7.2	7.8	8.0	8.8	9.7
maximum frequency (MHz)	165	139	129	125	114	103
latency (clock cycle)	9	12	13	13	16	17
latency (ns)	55	86	101	104	140	165
throughput (MFLOPS)	165	139	129	125	114	103

Tables 4.1 and 4.2 show the area, latency and throughput of several different floating-point formats including IEEE single-precision format for division and both IEEE single-precision and double-precision format for square root. The format is represented by the total number of bits; the number of exponent bits followed by the number of mantissa

bits are shown in parentheses. The sign bit is assumed. The fifth column of both tables represents IEEE single-precision format $32(8, 23)$. The last column of Table 4.2 is the IEEE double-precision format $64(11, 52)$. The quantities for the area of each design in both tables are evaluated using three metrics - number of slices, number of on-chip BlockRAMs and number of 18×18 embedded multipliers. The Xilinx Virtex-II XC2V6000 FPGA has a total number of 33792 slices, 144 on-board BlockRAMS, and 144 embedded multipliers.

It is obvious from both tables that the wider the floating-point bitwidth, the larger and slower the circuit. For wider designs, not only does the clock period increase, but also the number of clock cycles to generate the final result. The largest floating-point component that we can implement is limited by the number of on-board BlockRAMs. As we can see from both tables, the number of BlockRAMs required increases at a much faster rate than the number of slices and the number of embedded multipliers as bitwidth increases. This is because the sizes of the tables for the divider (in Figure 4.1) and for the square root (in Figure 4.3) increase exponentially while other parts of the design increase polynomially with the increase in bitwidth. Figure 4.1 shows that the divider requires one table with a size of $2^m \times (2m + 2)$. The address of this table is m-bits wide, where m is half of the mantissa bitwidth. Figure 4.3 shows that the square root uses more tables but the address of each table is k-bits wide, which is $1/4$ of the mantissa bitwidth. Therefore, the size of tables for the square root increases much more slowly than that of the divider. For small bitwidths, square root requires more BlockRAMs than the divider, while for large bitwidths, it requires fewer. The largest floating-point square root we can implement on a Virtex-II 6000 FPGA is IEEE double-precision 64(11,52), which requires about 80% (116 out of 144) of on-board BlockRAMs for this chip. The largest floating-point divider we can implement on the Virtex-II 6000 is the 40(10,29) format. This format requires about 43% (62 out of 144) of on-board BlockRAMs. Since the table size for the divider more than doubles when the width of the address increases just by one bit, IEEE double-precision divider cannot be implemented on this FPGA chip due to insufficient on-chip BlockRAMs.

Note that non-iterative algorithms are employed for both our divider and square root, while many other divider and square root implementations use iterative approaches. The non-iterative nature of the algorithms allow them be fully pipelined and thus the throughput is one result per clock cycle. This allows these units to easily fit into the pipeline of a large design with other library modules without decreasing the throughput of the whole design. The throughput of our IEEE single-precision format divider is as high as 129 million results per second. Meanwhile, this design only takes 1% of the slices, 4% of the BlockRAMs, and 5% of the embedded multipliers on the FPGA chip. Similarly, our floating-point square root shows a good tradeoff between area, latency and throughput.

4.3.2 Comparison of our Floating-Point Divide and Square Root and Xilinx Floating-Point IP Cores

The algorithms we use are not digit-recurrence as are most other divider and square root implementations. As a result, the latency does not increase linearly as the data bitwidth grows. Our goal is to keep the clock period relatively constant over a wide range of bitwidths and formats. Therefore we have to add more pipeline stages for wider bitwidth formats as a compromise. This results in a slightly increasing latency with the increasing bitwidth. Thus, the latency of our floating-point divide and square root is very short compared to most other divider and square root implementations.

Table 4.3 and Table 4.4 show the comparison of our implementation and Xilinx LogiCORE Floating-Point Operators v3.0 [107]. For both divide and square root, two implementations of Xilinx LogiCORE modules are experimented with: Xilinx 1 is the implementation that has the same latency in clock cycles as ours; Xilinx 2 is their most deeply pipelined implementation that has the highest maximum frequency and longest latency.

Table 4.3: Comparison of Our Floating-Point Divide and Xilinx LogiCORE Floating-Point Divide v3.0

Format	slices	BlockRAMs	$MULT18X18$s	latency	freq.	latency(ns)	throughput(MHz)
VFloat (8,23)	361	3	8	14	129	108	129
Xilinx 1 (8,23)	453	-	-	14	109	128	109
Xilinx 2 (8,23)	686	-	-	28	186	151	186
VFloat (10,29)	617	116	8	14	125	112	125
Xilinx 1 (10,29)	680	-	-	14	70	200	70
Xilinx 2 (10,29)	1031	-	-	34	168	203	168

Table 4.4: Comparison of Our Floating-Point Square Root and Xilinx LogiCORE Floating-Point Square Root v3.0

Format	slices	BlockRAMs	$MULT18X18$s	latency	freq.	latency(ns)	throughput(MHz)
VFloat (8,23)	351	7	9	13	125	104	125
Xilinx 1 (8,23)	327	-	-	13	89	146	89
Xilinx 2 (8,23)	407	-	-	28	213	132	213
VFloat (11,52)	1572	62	24	17	103	165	103
Xilinx 1 (11,52)	1003	-	-	17	45	381	45
Xilinx 2 (11,52)	1645	-	-	57	149	381	149

Our IEEE single-precision square root takes 13 clock cycles to generate the final result with an $8ns$ clock period and 125MHz frequency, resulting in a total latency of $104ns$. The Xilinx 1 single-precision square root with the same 13 clock cycles latency runs at a much slower clock of only 89MHz, resulting in a longer latency of $146ns$. The Xilinx 2 single-precision square root runs at a very high clock rate of 213MHz at the cost of many more clock cycles in latency, also resulting in a longer latency of $132ns$. The advantage of the short latency of our implementation is more obvious for larger bitwidth. For double precision square root, the latency of both Xilinx implementations is $381ns$, more than twice our latency of $165ns$. This is because the Xilinx square root uses a digit recurrence algorithm and the latency grows linearly with the bitwdith increase, while the bitwidth increase has little affect on the latency of our square root. Our divide also has much shorter latency compared to both Xilinx implementations, especially for large bitwidth.

4.4 Summary

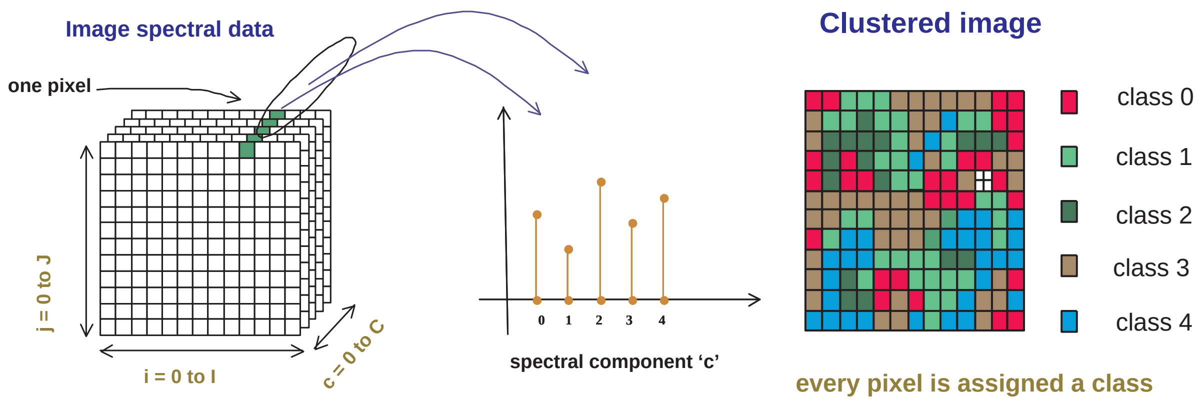
In summary, this chapter presented the hardware implementations of floating-point divide and square root. They are both fully parameterized and fully pipelined. In common with the other modules in the NU VFloat library, they support the creation of custom format floating-point datapaths as well. The area, latency, maximum frequency, and throughput of several different floating-point formats, including IEEE single-precision and double-precision formats for these two components are analyzed. Our results show that they exhibit a good tradeoff between area, latency and throughput, and are easily pipelined. Compared to divide and square root from the Xilinx LogiCORE Floating-point operators v3.0, our implementations have very small latency, especially for large bitwidth floating-point format. In the next two chapters we use our floating-point divide and square root in two real signal and image processing applications.

Chapter 5

First Application: K-means Clustering

To demonstrate the division implementation, we implemented a hybrid K-means clustering algorithm for multispectral satellite images. The K-means clustering algorithm is commonly used for segmentation of multi-dimensional data as shown in Figure 5.1. Figure 5.1(a) represents the input image data where each data pixel x_{ij} has C spectral components. Figure 5.1(b) is the output clustered image. In this example, the input image spectral data is classified into 5 different classes.

In this chapter, we present the hardware implementation of K-means clustering on the Mercury VantageRT Reconfigurable Computing Board [73] with one Xilinx Virtex-II XC2V6000 FPGA and onboard SRAMs and DRAMs (see section 2.4). First, we discuss the previous work that relates to our research. Then we introduce the K-means clustering algorithm. The high level design structure as well as the detailed hardware implementation are presented. Finally, we present the comparison of run-time between hardware and software implementations.



(a) Input Image Pixels (b) Output Clustered Image

Figure 5.1: K-means Clustering for Multispectral Satellite Images

5.1 K-means Related Work

Unsupervised clustering is a classic technique for segmentation of multispectral and hyperspectral images into K clusters. K-means is a simple iterative algorithm that generates successive clustering using a computationally expensive approach. Leeser et al. [57] first proposed a hardware implementation of the K-means clustering algorithm. Manhattan norm is proposed for distance calculation to reduce the number of multipliers without sacrificing the clustering quality. The authors also proved that the K-means algorithm can tolerate considerable bitwidth truncation of input data with little degradation in clustering quality [56]. This leads to a more compact and faster implementation. Expanding on their original ideas, they implemented K-means clustering of multispectral image on an Annapolis Wildstar board with three Xilinx Virtex1000 FPGAs and 40MB SRAM [27]. All input image data is pre-loaded to the onboard memory. The most computationally intensive part, distance calculation, is done in the FPGA. To accommodate arbitrary input data sets and faster generation of a hardware solution, a parameterized K-means implementation was also presented [55, 16].

Other early research was carried out by Lavenier [53], who implemented K-means clustering for hyperspectral images on various reconfigurable computing engines such as the Wildstar, SLAAC-1V and Spyder boards. In his work, only the distance calculation is done in the FPGA. The input image data is stored on the host and the data is streamed to the FPGA, so arbitrary size input images can be processed. This approach scales well to large input data sets but has more communication overhead between processor and hardware. Filho et al. [29] used a more accurate and expensive Euclidian distance in their hardware/software codesign K-means algorithm. The distance calculation is carried out in the Xilinx Spartan-II FPGA on an XESS PCI board. However, the hardware/software codesign only outperforms the software approach by about 2 times. To overcome the significant communication overhead between host processor and reconfigurable board due to slow I/O busses, K-means clustering on a hybrid processor was experimented with [40]. Two Altera Excalibur boards were used – one with a soft IP core 32-bit NIOS processor and one with a hard IP core ARM processor. A maximum speedup of 11.8x is achieved using the dual-port memory of the Excalibur ARM hybrid processor compared to a software implementation.

The most recent study on image data [69, 86] uses a variation of the K-means algorithm. First, a simplified filtering algorithm is used so that no more than 24 cluster centers need to be compared no matter how large the number of clusters K. Second, to handle large input datasets, a simplified FEKM (Fast and Exact K-Means) clustering algorithm is used to reduce the number of pixels that need to be scanned in every repetition. These techniques allow K-means clustering to consume much less time for large size images with large number of clusters. An average of 20-30 fps(frame per second) is achieved for a 756x512 pixel image divided into 256 clusters when implemented on an ADM-XRC-II board with one Xilinx XC2V6000 FPGA and 32MB SRAM.

In addition to segmentation of hyperspectral images, clustering has been used in many areas such as data mining, network security, and document clustering [18].

All previously published work that implements the K-means clustering algorithm in hardware for image and video processing uses software/hardware co-design approach. Only the distance calculation part of the K-means algorithm is done in FPGA hardware while the center updating in each iteration of the K-means is done on the host computer or embedded processor. The significant communication overhead between the host/embedded processor and the FPGA hardware is the primary impediment for high performance. To boost the performance of K-means clustering, this data transfer overhead has to be reduced as much as possible.

Our research is an extension of the previous work [16] applying K-means algorithm to multispectral satellite images using reconfigurable hardware. Due to the lack of a floating-point divide implementation, the mean updating step in each iteration of the K-means algorithm had to be moved to the host computer for calculation. The new means calculated on the host then had to be moved back to the FPGA board for the next iteration of the algorithm. This adds data transfer overhead between host and FPGA board. In our implementation, the mean updating step is done in FPGA hardware using the floating-point divide `fp_div` module presented in Chapter 4 of this dissertation. This greatly reduces the communication overhead between host and FPGA board and further accelerates the run-time. This example illustrates the use of our `fp_div` module, as well as conversions between fixed-point and floating-point (`fix2float` and `float2fix` modules in the NU VFloat library) seamlessly assembled together in a real application. Our K-means clustering implementation is parameterizable, highly parallel, and fully pipelined on a Xilinx Virtex-II XC2V6000 FPGA. With our fully-pipelined `fp_div` module, all computation in each iteration is now done on the FPGA, except for the mean initialization at the beginning of the algorithm. This resolves the data transfer overhead between the host and the FPGA during each iteration. An efficient memory hierarchy is deployed in this implementation. All input images are pre-loaded from the host at the start of computation and stored in onboard DRAM during the many iterations of the computation; the clustering result during computation is stored in onboard SRAMs and sent back to the host at the end; and the new means of each iteration are stored in on-chip memory.

5.2 K-means Clustering Algorithm

K-means is an unsupervised clustering algorithm where the total number of clusters K is given a *priori*. It works by assigning multidimensional vectors to one of K clusters. The aim of the algorithm is to minimize the variance of the vectors assigned to each cluster. The algorithm is iterative: after each vector is assigned to one of the clusters, the cluster centers are recalculated and the vectors are re-assigned using the new cluster centers. Pseudo-code describing the K-means algorithm is given in Figure 5.2. For more details see Duda and Hart [24]. In the pseudo-code, a total of N pixels are processed and assigned to one of the K clusters. The variable `centers` is an array of K multidimensional vectors which represent the center point or mean for each cluster. Each cluster has a counter and an accumulator for pixel values assigned to it.

The algorithm works as follows. First, means are initialized; we use a sub-sampling method. Next the while loop iterates until a termination condition is met. During each iteration, the distance to each cluster center is calculated for each pixel in the input image. The shortest distance is kept and the pixel is assigned to the closest cluster based on the distance between each pixel and each of the K cluster centers. Each pixel is then added to the accumulator of the cluster it is assigned to and the corresponding counter is incremented by one. After accumulation, the new mean of each cluster is calculated and is used for the next iteration. The new mean is obtained by dividing the accumulator value by the

```
kmeans(image) {
    //initialize cluster centers, accumulators and counters
    centers=initialize_cluster_centers();
    for(cluster=0;cluster<K;cluster++) {
        accumulators[cluster]=0;
        counters[cluster]=0;
    }
    //main loop; until terminating condition is met
    while(done!=TRUE) {
        //find pixel assignment for each pixel
        for(pixel=0;pixel<N;pixel++) {
            min_distance=infinity;
            for(cluster=0;cluster<K;cluster++) {
                //calculate the distance between pixel and cluster center
                distance=vector_distance(image[pixel],centers[cluster]);
                //assign the pixel to its nearest cluster
                if(distance<min_distance) {
                    cluster_image[pixel]=cluster;
                    min_distance=distance;
                }
            }
            //accumulate pixels and increment counter for each cluster
            accumulator(cluster_image[pixel])+=image[pixel];
            counter(cluster_image[pixel])++;
        }
        //update mean(cluster center) for each cluster
        for(cluster=0;cluster<K;cluster++)
            centers[cluster]=accumulator[cluster]/counter[cluster];
        done=terminating_condition();
    }
    //return clustered image and final positions of cluster centers
    return cluster_image, centers;
}
```

Figure 5.2: Pseudo-code of K-means Clustering Algorithm Based on [24]

counter value. In previous work [16] this mean updating step was done on the host because it requires floating-point divide. With our `fp_div` module, we are able to implement mean updating in FPGA hardware. This eliminates the overhead of moving the counter and accumulator values for each cluster from FPGA to host and moving the new means – the result of accumulator divided by counter for each cluster – from host back to the FPGA for each iteration. The use of the floating-point divide in FPGA hardware further accelerates the run-time of the K-means algorithm. At the end of each iteration, the termination condition is evaluated and, if not met, the process is repeated. Possible termination conditions are convergence or processing for a fixed number of iterations. When done, the algorithm returns the clustered image as well as the final mean values of all clusters.

5.3 High Level Design Structure

We implemented the K-means clustering algorithm from Figure 5.2 on the Mercury VantageRT FCN board described in section 2.4. In this work, we use the Xilinx Virtex-II XC2V6000 FPGA, DRAMs and SRAMs on the board. The PowerPCs on the board are not used. The implementation is partitioned between the host computer and the reconfigurable hardware. Cluster means are initialized on the host while the rest of the algorithm is done in reconfigurable hardware. Figure 5.3 shows the interaction between the host and reconfigurable hardware. All communication from host to reconfigurable board is over the PCI bus.

The operation of K-means clustering proceeds as follows. First, all pixels of the input spectral image are moved from the host to the DRAMs on the FPGA board. Then, initial cluster means are moved from host to the FPGA on the reconfigurable board. The input data pixels are stored in DRAMs on the board during the entire computation and are not changed during the iterations. The new means after each iteration are stored in the on-chip

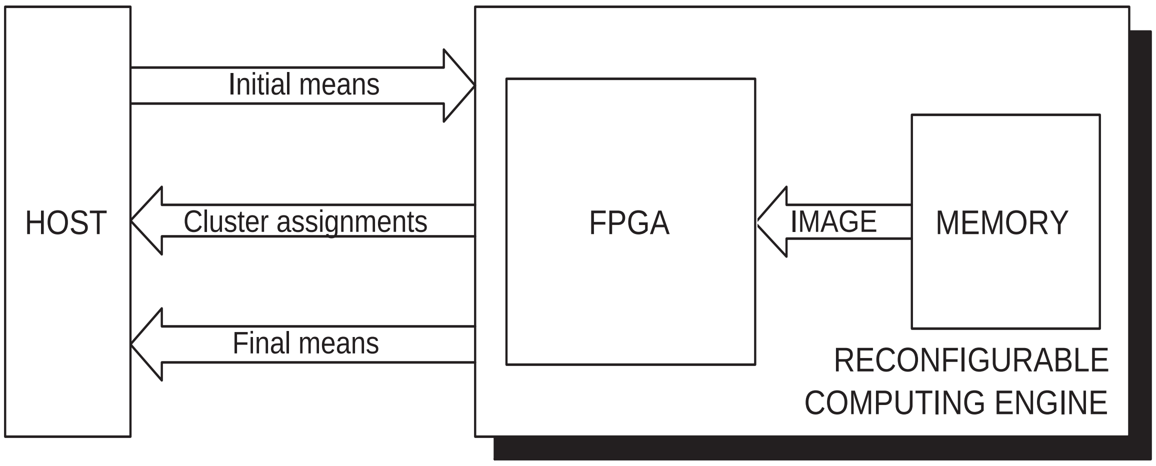


Figure 5.3: Interaction Between the Host and the Reconfigurable Hardware

memory of the FPGA. This is efficient because reading/writing of FPGA on-chip memory only requires one clock cycle while reading/writing of onboard memory can take many clock cycles. Due to the limited on-chip memory, the cluster assignment of each pixel is stored in onboard SRAM during computation. After the algorithm terminates, the final means stored in on-chip memory are moved back from the FPGA to the host, and the final cluster assignments are moved back to the host from onboard SRAM.

The reconfigurable hardware portion of the K-means clustering algorithm consists of three basic functions: assigning each pixel in the input image to one of the K clusters, accumulating the values of all pixels that belong to every cluster, and updating the mean for each cluster. These operations are performed on each pixel in the input image serially. The pixels are streamed from onboard memory and processed one per clock cycle until all pixels are processed.

Figure 5.4 shows the overall K-means clustering circuit. It is composed of three functional units as well as two shift-register units, validity and pixel shift that synchronize the pixel data with the computation. Inputs to the circuit are all pixels in the input image and

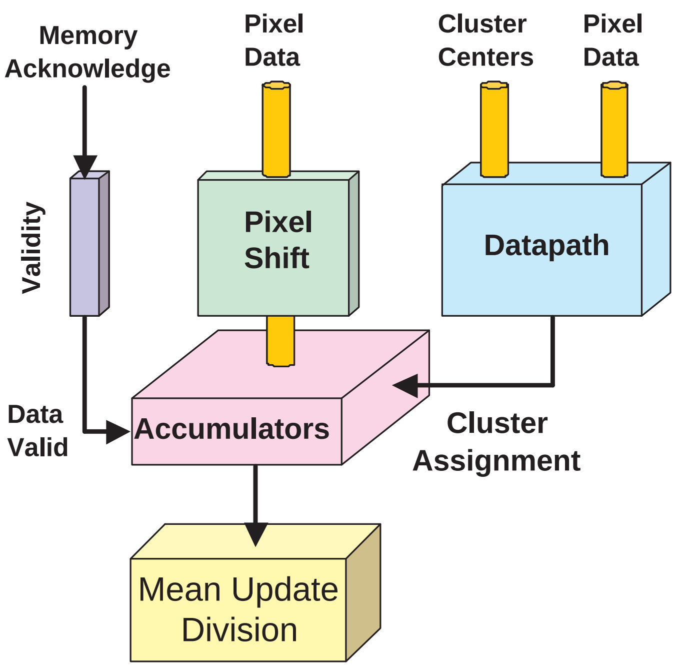


Figure 5.4: High Level Structure of K-means

the initial means of all K clusters; outputs are the cluster assignment for each pixel and the final value of the means of all K clusters. One functional unit is the datapath unit. It takes one pixel datum at each clock cycle along with all the cluster centers as inputs, and outputs the cluster assignment for that pixel. Each pixel is assigned to its nearest cluster, which is obtained by comparing the Manhattan(1-norm) distance between that pixel and every cluster center. This is the most computationally intensive part of the implementation. Another functional unit is the accumulator. Each accumulator is affiliated with one cluster and accumulates the total value of all pixels that belong to that cluster. There is also a counter associated with each cluster used to count the total number of pixels that belong to the cluster. The third functional unit is the floating-point divider for mean updating. Two shift units are used to delay signals from memory to the accumulator, thus synchronizing them with the cluster assignment which is delayed due to the latency of the datapath unit.

The pixel shift unit is a set of shift registers used to pass pixel values from memory to the accumulator in parallel with their operations in the datapath unit. The validity unit sets a `Data Valid` signal that indicates the pixel data is valid and enables the accumulator.

In our implementation, both datapath and accumulator units are implemented in fixed-point arithmetic while the divide unit is floating-point for improved precision. Converters from fixed to floating-point representation are placed before the floating-point dividers while floating to fixed-point converters are placed after the dividers. This hybrid implementation demonstrates that the NU VFloat library can support hybrid fixed and floating-point implementations.

5.3.1 Datapath Unit Complexity

Figure 5.5 shows the data flow of the datapath unit. The datapath unit determines the cluster assignment for each pixel. For each pixel, the cluster it belongs to is determined by first computing the distance of that pixel to every cluster center, using the Manhattan or 1-norm distance. The shortest distance is then found by comparing all distances. The pixel is assigned to its nearest cluster to minimize the variance within each cluster. For multispectal images, each pixel has several dimensions (or channels). Thus, the distance calculation needs to be done for each channel and the result is added for each pixel and cluster center pair. Assume that a C dimensional input image is segmented into K clusters. The Manhattan distance between one input pixel $pixel_{ij}$ and one cluster center $cluster_k$ is obtained by:

$$\sum_{c=1}^{C} |PIXEL_{ij}(c) - CLUSTER_k(c)| \tag{5.1}$$

We compute the Manhattan distance calculation for all K clusters in parallel. This calculation requires $K \times C$ subtractions, $K \times C$ absolute values, and $K \times (C-1)$ additions.

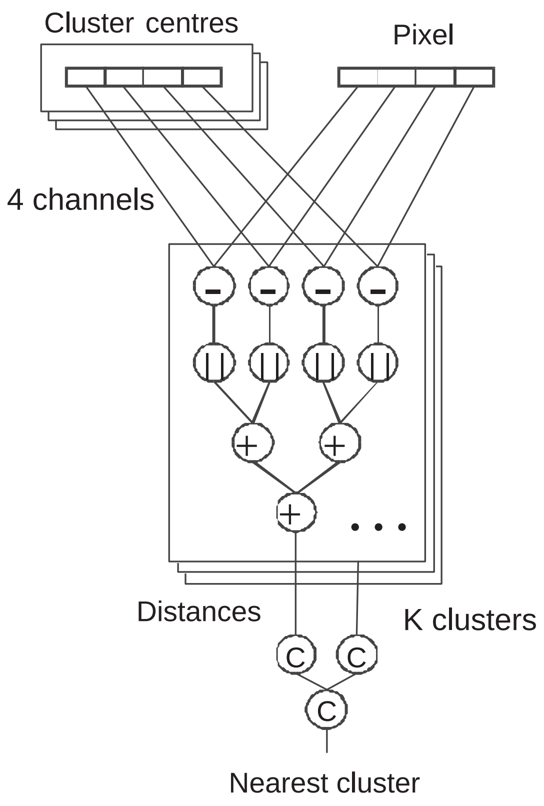


Figure 5.5: Datapath Unit of K-means

For an image with 8 channels and 8 clusters, 64 subtractors, 64 absolute values, and 56 adders are required. An adder tree with $\sum_{i=1}^{log_2 C} C/2^i$ adders is used to sum up absolute distances of all C channels for each cluster. Altogether 8 adder trees with a total of 7 adders each are required. Once the distances between one pixel and all K cluster centers are known, K comparators are required to find the shortest distance. Again, a comparison tree is adopted to exploit more parallelism. In this example, we need one comparison tree of 7 comparators. In summary, a total of 191 operations are needed.

5.3.2 Mean Update Unit

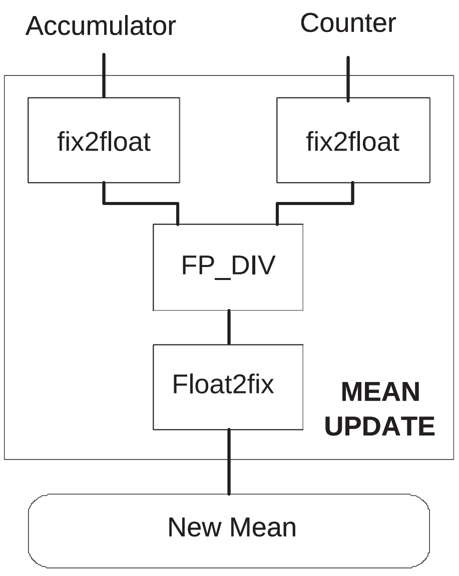


Figure 5.6: Mean Update Unit of K-means

After the datapath unit computes the cluster assignment of each pixel, the accumulator unit collects the total number of pixels assigned to this cluster, stored in register `counter`; and the total value of all pixels associated with this cluster, stored in register `accumulator`. New mean values for each cluster are calculated by dividing the `accumulator` value by the `counter` value. Note that there are K counters and K accumulators, one pair for each cluster. In this work, the mean update division is done in FPGA hardware. For arithmetic accuracy, divide is implemented in floating-point using the `fp_div` module.

Figure 5.6 shows the data flow of the mean update unit. Since both the `counter` and `accumulator` are in fixed-point format, `fix2float` conversion has to be done first. After the floating-point divide, the new mean in floating-point format needs to be converted back to fixed-point format using the `float2fix` module. For an image with 8 clusters, a total of 8 `fp_div` modules are in parallel, one for each cluster. Also, we need 8 `float2fix` modules, and 16 `fix2float` modules (8 for `counter` and 8 for `accumulator`).

5.4 Experimental Results

We implemented K-means clustering on the Mercury VantageRT FCN board for a multi-spectral satellite image with 614×512 pixels. Each pixel has 8 channels, with 8 bits of data per channel. The input pixels are segmented into 8 clusters. One resulting clustered image, shown in Figure 5.7, uses eight different colors to represent the different clusters. Note that this is a "pseudo-color image" with colors that carry no information randomly assigned to clusters.

Our design is written in VHDL and synthesized using Synplify Pro 8.0. The bitstream is generated using Xilinx ISE 6.3i and downloaded to the XC2V6000 FPGA on the board. All the results that we present in this section are based on hardware running on the VantageRT FCN board. To verify the correctness of the K-means clustering algorithm implemented in the FPGA, we compared it against a pure software implementation on a 3.2GHz Intel Pentium 4 processor. Both implementations generate the same results of final mean

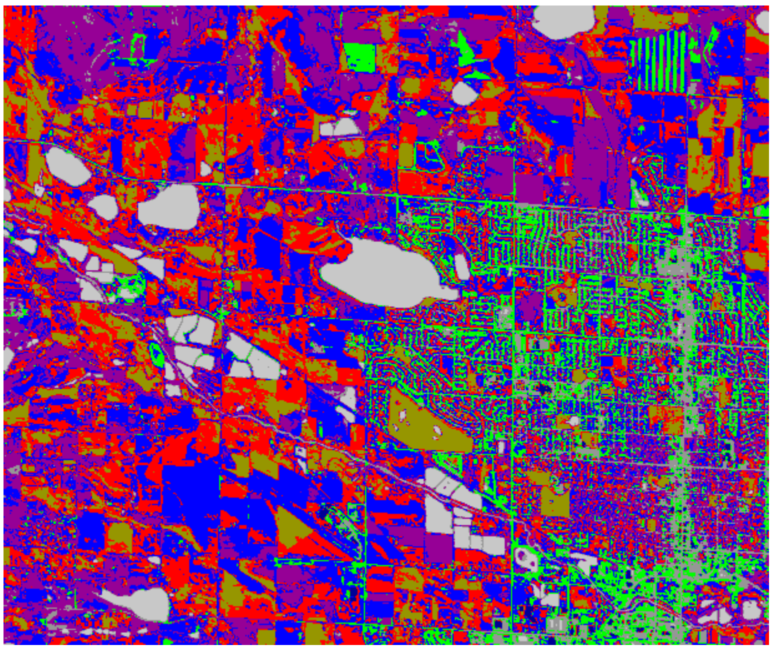


Figure 5.7: Output Clustered Image: the City of Denver

and cluster assignment for each pixel. To see the advantage of having floating-point divide in FPGA hardware instead of running mean update on the host, we experimented with two implementations, one with divide in the FPGA and the other with divide on the host. In the following discussion, we compare the results of these three implementations: pure software, hardware implementation with divide in FPGA hardware, and hardware implementation with divide on the host.

5.4.1 Hardware vs. Software

Table 5.1 shows the run-time of the pure software implementation and the hardware implementation with divide in the FPGA for different numbers of iterations. The first column is the number of iterations that the algorithm runs. The second and third columns are the

Table 5.1: Run-time of K-means Clustering Algorithm: Hardware vs. Software

iteration	**Software**		**Hardware (divide in FPGA)**			
	comp(s)	total(s)	comp(s)	**comp speedup**	total(s)	**total speedup**
1	5.203	38.047	0.0024	**2168**	27	**1.4**
20	104	137	0.048	**2167**	27	**5.1**
50	258	291	0.12	**2150**	27	**10.8**
1000	5188	5220	2.4	**2162**	30	**174**

core computation time and total run-time (both in seconds) of the pure software implementation. The fourth and sixth columns are the core computation time and total run-time of the hardware implementation. For both hardware and software implementations, core computation time is the run-time just for K-means algorithm; while the total run-time includes core computation time, as well as data transfer and file I/O time. The speedup of the hardware implementation compared to software are listed in the fifth and seventh columns. First, from Table 5.1 we see that the core hardware implementation achieves a huge speedup of over 2150 times. This proves that our FPGA implementation greatly improves the performance of the K-means clustering algorithm.

A second observation is that, while the speedup of the core computation time is three orders of magnitude, the speedup for the total run-time is much less, especially when the number of iterations is small. For most images, 50 iterations is sufficient to achieve convergence. In this case, the speedup of the total run-time is about 11x. This is because the portion of the run-time spent in transferring the input image to the board and transferring the output cluster assignment back to the host for a 614×512 pixel image is significant. The data transfer time in the hardware implementation is about 27 seconds, which includes reading the 614×512 pixel (where each pixel is 64 bits) input image from an external image file, then moving input pixel data from the host to the DRAM on the FPGA board at the start of computation. At the end, the cluster assignment of every pixel must be moved from the SRAM on the board and the final means must be moved from the on-chip memory on the board back to the host, then cluster assignments are written to an output data file. In the software implementation, the similar input/output data streaming takes about 33 seconds, which includes reading the input data file, storing it to local variables, and at the end reading the output from local variables and writing to an output data file. We can see that file I/O takes most of the time in both implementations. Note that the input image data file format is different for the two implementations so the file I/O times are slightly different. We can see that for a small number of iterations, this data transfer time dominates the run-time for both software and hardware implementations. In the hardware implementation where the core computation is greatly accelerated via FPGA hardware, this data transfer overhead becomes the bottleneck. Table 5.1 shows that in the hardware implementation, for number of iterations up to 50, the FPGA core computation time is negligible compared to the data transfer time.

5.4.2 Divide in the FPGA vs. Divide on the Host

Table 5.2: Run-time of Two Hardware Implementations of K-means Clustering Algorithm

iteration	**Hardware(divide on host)**			**Hardware(divide in FPGA)**		
	comp(s)	divide(s)	total(s)	comp(s)	divide(s)	total(s)
1	0.0025	0.0001	27	0.0024	0.000001	27
20	0.05	0.002	27	0.048	0.0002	27
50	0.125	0.005	27	0.12	0.0005	27
1000	2.5	0.1	30	2.4	0.001	30

The run-time comparison of the two hardware implementations, with divide in the FPGA and with divide on the host, is described in Table 5.2. Similar to Table 5.1, it lists the core computation time and total run-time for different numbers of iterations. Table 5.2 also shows the time for the *divide* operations, which are part of the core computation.

One interesting observation is that the difference between these two implementations is fairly small. This is not hard to explain with Amdahl's Law. We can see from Table 5.2 that *divide* takes a very small fraction of the core computation time, and even a smaller portion of the total run-time. The run-time is dominated by the fixed point datapath. For the first implementation with divide on the host, in one iteration *divide* takes $0.1ms$, which is about 4% of the core computation time while the core computation time is only about 0.009% of the total run-time. After moving *divide* from the host to the FPGA, the time spent for divide is accelerated 100 times, to about $1us$. Now the fraction of time spent on divide in the core computation is even smaller – only 0.04%. Since the *divide* is such a small portion of the whole computation, no mater how significant the improvement is, the effect on the whole design is modest.

The main advantage of having the *divide* in the FPGA is to free the host to work on other tasks concurrently with K-means clustering being implemented on the FPGA board. In scenarios where an image analyst is using K-means as a preprocessing step, further processing can be done on the host while clustering is performed on another image. The ability to exploit this coarse-grained parallelism will provide significant speedup for the image analyst.

5.5 Summary

This chapter presents an implementation of the K-means clustering algorithm for multi-spectral satellite image processing using our variable precision floating-point divide. Speedup is shown for moving the mean updating step from host PC to FPGA hardware with the `fp_div` module, eliminating the data transfer overhead between the host and the FPGA for each iteration. We also see that the data transfer time in all implementations dominates the run-time. This includes the time for reading the image data from an external input file, moving input pixel data from the host to the DRAM on the board, and the time for moving final cluster assignments stored in SRAM on the board, as well as the final means stored in on-chip memory on the FPGA back to the host and writing results to an output file. Since divide is a small percentage of the K-means algorithm, Amdahl's Law tells us that speeding up division will not have a great impact on overall run-time. The great benefit comes from being able to exploit the coarse-grained parallelism of running different applications on the host and the FPGA concurrently. To further demonstrate our variable precision floating-point divide and square root, both of them are used together in another application (QR decomposition), which is discussed in the next chapter.

Chapter 6

Second Application: QR Decomposition

The second application of our floating-point divide and square root is QR decomposition (QRD) [31]. In QR decomposition, an $m \times n$ real matrix A is written as:

$$A = QR \tag{6.1}$$

where Q is an $m \times m$ orthogonal matrix such that $QQ^T = I$ (I is the identity matrix), and R is an $m \times n$ upper triangular matrix. An orthogonal transform of a vector $X = [x_1, \ldots, x_m]^T$ can be defined by the orthogonal matrix Q as:

$$Y = Q^T X \tag{6.2}$$

The orthogonal transform does not change the vector's length: $\|Y\| = \|X\|$.

QR decomposition is well known for its computational stability and fast convergence. It is used in many applications including adaptive beamforming, phased-array radar & sonar, 3G wireless communication, channel equalization, echo cancellation, de-noising, smart antennas, and WiMAX (Worldwide Interoperability for Microwave Access). There are many algorithms for solving the QR decomposition problem including the Gram-Schmidt [83] and modified Gram-Schmidt algorithms, Householder transformations [47], and Givens rotations [38].

In this chapter, we use the Givens rotations algorithm for QR decomposition. Using Givens rotations, QR decomposition can be represented as a highly parallel array of computational processors. This leads to a systolic array architecture [37, 68]. The massive parallelism of this algorithm is very well suited to FPGA architectures and can greatly reduce the computation time. We start off by introducing the previous work on QR decomposition using Givens rotations. Then we detail the design architecture and hardware implementation of QR decomposition based on a two dimensional (2D) systolic array architecture. Finally, we present experimental results of our 2D systolic array QR implementation, as well as the comparison between our 2D systolic array implementation and a 1D systolic array implementation.

6.1 QR Related Work

There has been a lot of previous work on systolic array implementations of QR decomposition using Givens rotations on FPGAs [102, 103, 52, 90, 89, 46, 95, 96, 91, 30, 17, 36]. The implementation of the Givens rotations algorithm in these works fall into three categories. One class of implementation is square root free using the Squared Givens Rotation (SGR) algorithm. Another type of implementation is based on Logarithmic Number System (LNS) arithmetic. The third category of implementation uses the CORDIC algorithm.

The earliest published work on QR decomposition using Givens rotations in an FPGA [102] does weight calculation in an adaptive beamforming application for radar systems. QR decomposition is a very computationally intensive step, so a highly parallel solution is desired. In this work, a linear QR array architecture with discrete mapping is implemented on a Xilinx Virtex XCV1000 FPGA. A Squared Givens Rotation (SGR) algorithm [23] is used to avoid the square root operation. To achieve 70dB interference suppression, a floating-point format with 6-bit exponent and 14-bit mantissa is used. FPGA and ASIC implementations are compared. The throughput of the FPGA implementation is about 5GFLOPS, making it a viable alternative to an ASIC implementation. Follow up work [103] using the SGR algorithm and a linear array targets a Xilinx Virtex-E XCV3200E FPGA. A maximum of 9 processors can fit in this FPGA, achieving a clock rate of 150MHz and a throughput of 20GFLOPS. The throughput of the FPGA implementation is about 100 times higher than that of an implementation on a TI TMS320C6701 DSP processor. More recent work on QR decomposition based Recursive Least Square (QRD-RLS) for matrix inversion [52] also uses the SGR algorithm. Their floating-point operations are implemented with the Xilinx LogiCORE Floating-Point Operators v3.0 [107] using Xilinx System Generator. The input is a 4×4 matrix of complex floating-point data with 6-bit exponent and 14-bit mantissa. A partial systolic array architecture is implemented on a Xilinx Virtex4 XC4VLX200 FPGA. It achieves a throughput of 0.13M updates per second with a clock rate of 115MHz. The latency for generating the upper triangular matrix is 777 cycles.

The second type of QR decomposition using the Givens rotations algorithm is based on logarithmic number system (LNS) arithmetic. In the LNS system, multiplication and division operations become addition and subtraction respectively, and the square root operation becomes a simple bit-shift. As a result, the Givens rotations algorithm is divide and square root free in the LNS system. Matousek et al. implemented the QRD algorithm using their high-speed logarithmic arithmetic library (HSLA) [70]. Schier et al. continued this line of research using the same HSLA library to implement floating-point operations for Givens rotations [90] and Recursive Least Squares (RLS) [89]. Their hardware was designed using Handel-C from Celoxica [13] and targets a Xilinx Virtex-E XCV2000E FPGA. Two LNS data formats are implemented: one is a 19-bit LNS format and the other is a 32-bit LNS format. Both formats have an integer part of 8 bits. In their work, only a single row was implemented, but not the full 2D systolic array. Since addition and subtraction become the most computationally complex modules in an LNS system, for 32-bit arithmetic, only one logarithmic addition/subtraction block can fit on the Xilinx XCV2000E FPGA. Based on their QR processor, a finite interval constant modulus (CMA) algorithm was implemented on a Xilinx XCV2000E FPGA [46]. To improve the CMA performance of their original work, more recent work [95, 96] focuses on the high-level synthesis scheduling algorithm. In addition to using the same Xilinx XCV2000E FPGA, the CMA algorithm is also implemented on a newer Xilinx Virtex-II XC2V1000 FPGA. The throughput is 300MFLOPS on the Virtex-II and 210MFLOPS on the Virtex-E for an input matrix size of 24×20. A 46% speedup is achieved using their new scheduling algorithm compared to their first CMA implementation [46].

The third class of QR decomposition using Givens rotations is based on the CORDIC (COordinate Rotation by DIgital Computer) algorithm [101] to avoid divide and square root operations. Walke et al. [102] first implemented QR decomposition using the CORDIC algorithm. A linear QR array architecture with mixed mapping is implemented on a Xilinx XCV1000 FPGA, achieving a throughput of 6.4GFLOPS. With mixed mapping, each cell in systolic array has dual functionality of both boundary and internal cells. A 26-bit fixed-point data format is used to guarantee sufficient interference suppression. A lot of recent work continues this original line of research. One work [91] focuses on a new method for mapping the systolic array design to improve throughput. The throughput for a 9×9 matrix with 32-bit integer data is 340MFLOPS on a Xilinx Virtex XCV800 FPGA.

There are also many commercial QR IP cores that use the CORDIC algorithm. Altera has published a QRD-RLS design [17, 30] based on their CORDIC IP core [11] that supports applications such as WiMAX [12], smart antenna-beamforming [10], channel estimation and equalization of 3G wireless communications [9]. Altera's CORDIC IP block has a deeply pipelined parallel architecture enabling speed over 250MHz on their Stratix FPGAs. Their first published work [17] explored three mapping techniques for the systolic array – direct mapping, discrete mapping and mixed mapping – applied to a 64×9 matrix with 16-bit data on an Altera Stratix FPGA. Their second published work [30] implemented QRD-RLS for a Software Defined Radio (SDR) application where the FPGA is configured as a number of custom processors. Each processor is called an Application Specific Integrated Processor (ASIP). The ASIP is desired with high-level synthesis and a software-oriented approach compared to conventional hardware design.

QinetiQ Ltd. also has a CORDIC based floating-point QR Processor IP core [82]. It is fully pipelined with a throughput of 137MHz on Xilinx Virtex-II FPGAs. It can be implemented over multiple FPGAs to provide very high throughput on large problems. For example, the throughput is 20GFLOPS with 13 processors on Xilinx XC2V6000-5 FPGAs. Using their QR processor core for weight calculation as well as their floating-point IP cores [81], adaptive beamforming was implemented [36]. Two floating-point formats are implemented: the first is IEEE single-precision format; the second is a smaller format with 6-bit exponent and 14-bit mantissa.

Xilinx has a similar CORDIC IP core [110]. Using the CORDIC core, they implement adaptive beamforming based on QR decomposition [109]. Xilinx also has QR decomposition, QR inverse and QRD-RLS spatial filtering IP cores available in the AccelWare DSP IP Toolkits [2] originally from AccelChip, Inc. AccelWare is a library of floating-point MATLAB model generators that can be synthesized by AccelDSP into efficient fixed-point hardware. These AccelWare DSP cores are used in beamforming [1] and WiMAX baseband Multiple-Input Multiple-Output (MIMO) [3] applications.

All previous work on QR decomposition using Givens rotations avoids the divide and square root steps in the algorithm by using the CORDIC algorithm [102, 91, 17, 30, 36], the Squared Givens Rotation (SGR) algorithm [102, 103, 52], or the Logarithmic Number System (LNS) [90, 89, 46, 95, 96]. Due to the limited dynamic range of the fixed-point representation of CORDIC, the wordlength requirement is much greater than other floating-point implementations for the same accuracy. In addition, there are larger errors due to many sub-rotations of the CORDIC algorithm. The SGR algorithm is square root free at the cost of worse numerical accuracy compared with the standard Givens Rotation algorithm with square root. Using a logarithmic number system makes multiply, divide and square root operations less expensive. However, the simple add/subtract operation becomes much more costly. Furthermore, fitting a QR module based on the LNS system into a large design that uses a conventional number system requires extra conversion overhead.

In this work, we implement QR decomposition with a well-known systolic array architecture based on the Givens rotations algorithm on a Xilinx Virtex5 FPGA. The input/output data as well as all arithmetic operations of our QR decomposition are in floating-point format. All floating-point operations including floating-point divide and square root are implemented using modules from the NU VFloat library. Since floating-point cores are standard in many design libraries, this approach is more portable for different systems. Also, compared to the special operations in CORDIC or LNS, this implementation is better established and understood, and is easier to fit into a larger system design based on conventional operations.

6.2 QR Decomposition Algorithms - Givens Rotations

Givens rotations [38] is a popular method for implementing QR decomposition. More detail of Givens rotations can be found in many matrix algebra textbooks [43, 74]. In this section, we briefly introduce this algorithm so that we can explain our hardware implementation in the next section. To obtain the upper triangular matrix R, a series of rotations are applied to the rows of the original matrix A, zeroing the elements of the lower triangular sub-matrix one at a time. It works as follows: an $m \times n$ matrix A is zeroed out one element at a time using a 2×2 rotation matrix $Q_{i,k}$:

$$Q_{i,k} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{6.3}$$

where $i = [2 \ldots m]$, and $k = [1 \ldots n]$. Here c and s stands for cosine and sine parameters, where $|c|^2 + |s|^2 = 1$. Each rotation matrix $Q_{i,k}$ is orthogonal and will zero out the element a_{ik} in matrix A, starting at the bottom of the first column and working up the columns, then move to the second column and so on. A series of rotation matrices $Q_{i,k}$ are applied to the original matrix A until it becomes an upper triangular matrix R. Figure 6.1 shows the structure of a sequence of matrices produced by successive operations of Givens rotations. Here, the rotation is shown above the arrow, and the matrix elements affected by the preceding transformation are shown in boldface.

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{G_{4,1}} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \mathbf{0} & \times & \times \end{bmatrix} \xrightarrow{G_{3,1}} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \mathbf{0} & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{G_{2,1}} \begin{bmatrix} \times & \times & \times \\ \mathbf{0} & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{G_{4,2}}$$

$$\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \mathbf{0} & \times \end{bmatrix} \xrightarrow{G_{3,2}} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \mathbf{0} & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{G_{4,3}} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \mathbf{0} \end{bmatrix}$$

Figure 6.1: Sequence of Matrices via Successive Givens Rotations

Equation (6.4) shows one Givens rotation involving two rows of the input matrix using one 2×2 rotation matrix Q (Eq. (6.3)):

$$\begin{bmatrix} 0 & \dots & 0 & r & r'_{11} & \dots & r'_{1i} & \dots \\ 0 & \dots & 0 & 0 & r'_{21} & \dots & r'_{2i} & \dots \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} 0 & \dots & 0 & x & r_{11} & \dots & r_{1i} & \dots \\ 0 & \dots & 0 & y & r_{21} & \dots & r_{2i} & \dots \end{bmatrix} \tag{6.4}$$

On the right side of equation (6.4), an orthogonal transform is applied to the two rows of the input matrix via the orthogonal rotation matrix Q. The first row vector is $\mathbf{R_1}$:

$$\mathbf{R_1} = \begin{bmatrix} 0 & \dots & 0 & x & r_{11} & \dots & r_{1i} & \dots \end{bmatrix}$$

where x is the first non-zero element of row vector R_1. The second row vector is $\mathbf{R_2}$:

$$\mathbf{R_2} = \begin{bmatrix} 0 & \dots & 0 & y & r_{21} & \dots & r_{2i} & \dots \end{bmatrix}$$

where y is the first non-zero element of row vector R_2. After one Givens rotation, the first non-zero element y of the second row vector R_2 is zeroed out. The remaining elements of the two rows are updated to two new row vectors as shown on the left side of equation (6.4):

$$\mathbf{R'_1} = \begin{bmatrix} 0 & \dots & 0 & r & r'_{11} & \dots & r'_{1i} & \dots \end{bmatrix}$$
$$\mathbf{R'_2} = \begin{bmatrix} 0 & \dots & 0 & 0 & r'_{21} & \dots & r'_{2i} & \dots \end{bmatrix}$$

The 2×2 rotation matrix Q is formed from two sine and two cosine parameters. The sine s and cosine c parameters can be computed using the following formulae:

$$c = \frac{x}{\sqrt{x^2 + y^2}} \tag{6.5}$$

$$s = -\frac{y}{\sqrt{x^2 + y^2}} \tag{6.6}$$

where x and y are the first non-zero elements of row vectors R_1 and R_2, on the right side of equation (6.4). The updated elements of the two new row vectors, as shown on the left side of equation (6.4), can be computed as:

$$r_{1i}^{'} = c \times r_{1i} + s \times r_{2i} \tag{6.7}$$

$$r_{2i}^{'} = c \times r_{2i} - s \times r_{1i} \tag{6.8}$$

Since one rotation involves only two matrix rows and zeros one element, multiple rotations can be computed in parallel if they operate on different rows, leading to the potential for massively parallel computation.

6.3 Design Architecture

A two dimensional (2D) systolic array of computational processing elements (PEs) are implemented in this research. These parallel PEs perform rotations on each element in the input matrix to zero out the elements in the lower triangular part of the input matrix, thus transforming the input matrix to upper triangular form. In this work we explore fitting as many computational PEs as possible on an FPGA to achieve maximum parallelism. The number of computational PEs is limited by the available FPGA resources. In addition to the computational PEs, local communication between these PEs has to be carefully designed. Many other factors such as the complexity of the inter-processor connections, and the delay of the inter-processor communication need to be fine tuned. All these design decisions impact the final run time of the algorithm.

6.3.1 Systolic Array Structure of QR Decomposition

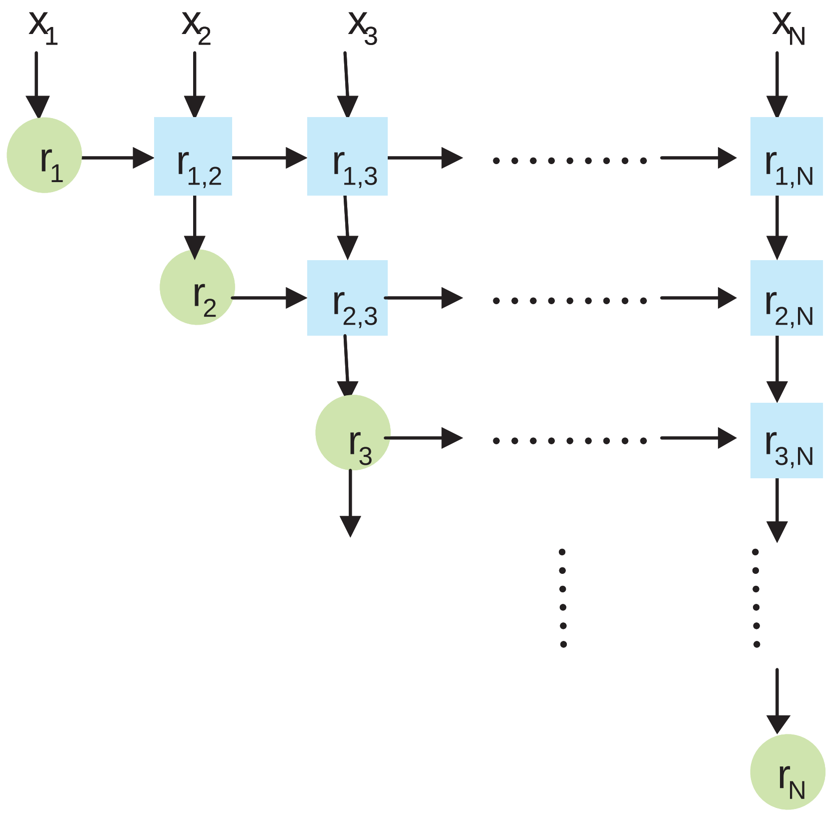


Figure 6.2: Systolic Array for the QR Algorithm

Figure 6.2 shows the highly parallel 2D systolic array architecture for QR decomposition that we implement. This systolic array consists of two types of computational processing elements (PE): diagonal processing elements and off-diagonal processing elements. The diagonal PEs (shown as circles in Figure 6.2) are used to calculate the Givens rotation that is applied across a particular row in the input matrix. An orthogonal rotation matrix (Eq. (6.3)) that can annihilate one lower triangular element of the decomposed matrix is calculated, and the rotation parameters c and s (Eq. (6.5) and (6.6)) are output to the off-diagonal PEs. The rotation angles generated by the diagonal PE are broadcast at the same time to all off-diagonal PEs in the same row as the diagonal PE in the systolic array. The off-diagonal PEs (shown as squares in Figure 6.2) apply the orthogonal transform (Givens rotation) received from the diagonal PE in each row as input values, to calculate new values as outputs. Specifically, using the rotation angles c and s from the diagonal PE as inputs, the off-diagonal PEs update the remaining elements in the two rows of the matrix involved in one rotation as specified in Equations 6.7 and 6.8.

The systolic array works as follows: the rows of the input matrix are fed as inputs to the systolic array from the top. After receiving the first input value, the diagonal PE r_1 (in the first row in Figure 6.2) starts processing. The off-diagonal PEs $r_{1,2},\ r_{1,3},\ \cdots,\ r_{1,N}$ in the first row update the elements 2 to n of the first row of the input matrix. All these off-diagonal PEs start simultaneously upon receiving the c and s from the diagonal processing element r_1. The updated values are then sent to the PEs in the second row of the systolic array. Upon update completion, the new values of the first off-diagonal processing element $r_{1,2}$ are sent to the diagonal PE below, stimulating the computation of the diagonal processing element r_2 immediately. The new values of the rest of the off-diagonal PEs in the first row $r_{1,3},\ \cdots,\ r_{1,N}$ have to be stored in local memory before being passed to the off-diagonal PEs in the second row. This is because the off-diagonal PEs in the second row cannot start computation until they receive c and s from the diagonal processing element r_2 of their row. In the same fashion, the diagonal element r_3 will start processing upon completion of the off-diagonal processing element $r_{2,3}$ above it. It then stimulates all the off-diagonal processing elements in the same row and so on. Data flows from top to bottom and from left to right in this structure; the calculation of the decomposed matrix R propagates through the systolic array on a diagonal wavefront. Once all inputs have passed through the systolic array, the values held in each of the PEs are the output values of the decomposed R matrix.

For an $n \times n$ square input matrix, the systolic array is in a triangular form with n rows. Each row has one diagonal PE and a set of off-diagonal PEs. Every off-diagonal PE except the first one of a row has one local memory of 8 words. The number of off-diagonal PEs is $n-1$ in the first row, $n-2$ in the second row, etc. In total, n diagonal PEs, $\frac{n\times(n-1)}{2}$ off-diagonal PEs, and $\frac{(n-1)\times(n-2)}{2}$ local memories are needed. In addition to processing a square matrix, our implementation works for any size input matrix. For a $m \times n$ tall matrix with $m > n$, the systolic array structure is exactly the same as an $n \times n$ square matrix. For a $m \times n$ short matrix with $m < n$, the systolic array structure and the number of PEs are slightly different. A short matrix systolic array has a short form with a total of m rows and m diagonal PEs. The first row has $n-1$ off-diagonal PEs, the second row has $n-2$ off-diagonal PEs, $\cdots$, and the last row has $n-m$ off-diagonal PEs. In total, m diagonal PEs, $n\times m - \frac{m\times(m+1)}{2}$ off-diagonal PEs, and $n\times m - \frac{m\times(m+3)}{2}$ local memories are needed.

6.3.2 Diagonal Processing Element

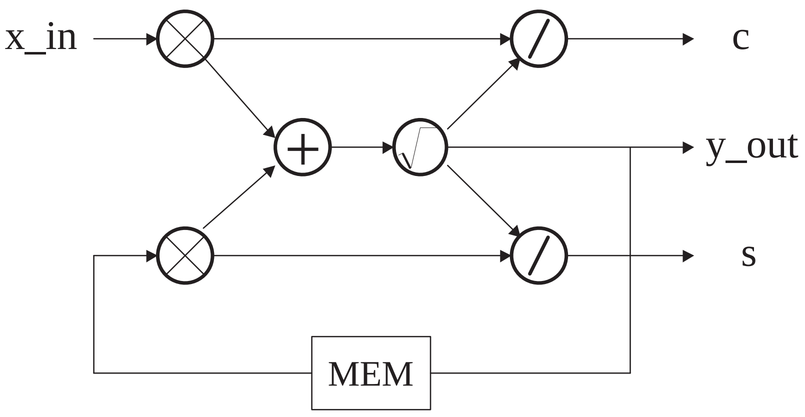


Figure 6.3: Diagonal Processing Element

Figure 6.3 shows the data flow of a diagonal PE module, which requires two multipliers, one adder, one square root, two dividers and one local memory. This datapath is a straightforward implementation of the equation pair (6.5) and (6.6). All units are in floating-point format and are modules from the NU VFloat library. All units including the floating-point divide and square root (described in Chapter 4) are fully pipelined, making the diagonal PE fully pipelined. The input to the diagonal PE is `x_in`, the outputs from the diagonal PE are `y_out`, `c` and `s`. The rotation angles sine and cosine are generated as specified in Eq. (6.5) and (6.6). This equation pair has two inputs x and y and two outputs c and s. One input x is the input `x_in` to the diagonal PE module. The other input y is the previous value stored in the internal memory of the diagonal PE. The current input value x and the previous value y are elements of the same column and neighboring rows of the input matrix. During one Givens rotation, an orthogonal transform is applied and the new value is calculated. This new value is stored in internal memory and will become the previous value used as one of the inputs for the next rotation.

6.3.3 Off-diagonal Processing Element

Figure 6.4 shows the data flow of an off-diagonal PE module, which has four multipliers, one adder, one subtractor, and one local memory. This datapath is a straightforward implementation of the equation pair (6.7) and (6.8). All units are in floating-point format and are modules from the NU VFloat library. The off-diagonal PE is also fully pipelined. The external inputs to the off-diagonal PE are r_1 and the rotation parameters `c` and `s`. The outputs from the off-diagonal PE are $r_1^{'}$ and $r_2^{'}$. Using the rotation angles c and s from the diagonal PE, the updates are performed following Eq. (6.7) and (6.8). This equation pair has four inputs $c,\ s,\ r_{1i}$ and r_{2i} and two outputs $r_{1i}^{'}$ and $r_{2i}^{'}$. The inputs $c,\ s$ and r_{1i} are the inputs `c`, `s` and r_1 from the diagonal PE modules. Another input r_{2i} is from the previous

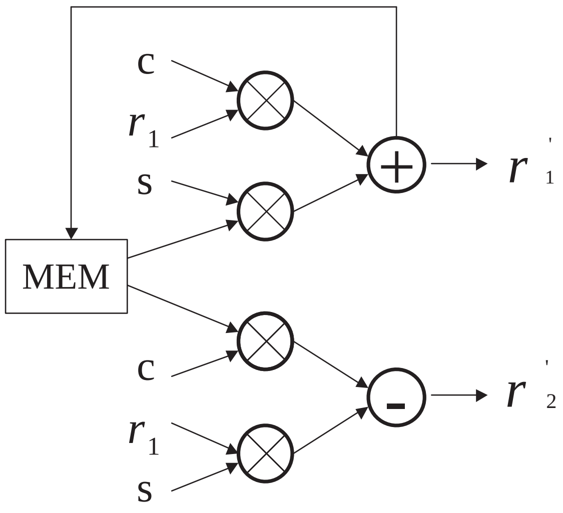


Figure 6.4: Off-diagonal Processing Element

value stored in the internal memory of the off-diagonal PE. The current input value r_1 and the previous value r_2 are elements of the same column and neighboring rows of the input matrix. The off-diagonal PE applies the orthogonal transform to the input value and the previous value, and calculates two new values. One of the new values is the current output. The other new value is stored in the internal memory and will become the previous value and one of the inputs for the next rotation.

Each diagonal and off-diagonal PE also has a control unit that triggers the processing of each PE in the entire systolic array. The control units are fairly simple; they control the flow of inputs and outputs of every PE and every memory block of the systolic system.

6.3.4 Specifications of the QR Core

The component symbol for the QR core is shown in Figure 6.5. Each element of the input matrix is streamed into the QR hardware module as input signal `din` and stored

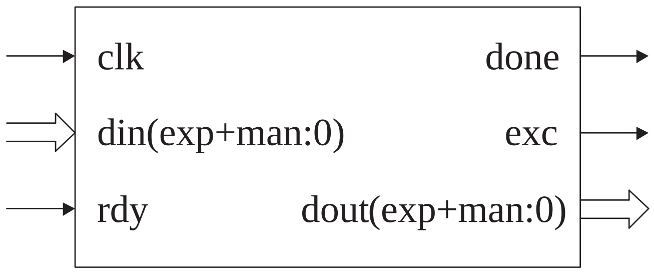


Figure 6.5: QR Component Symbol

in the on-chip FPGA memory before processing. Similarly, the decomposed matrix R is temporarily stored in on-chip FPGA memory. Each element of the final output matrix is then streamed out as output signal `dout` after the completion of the entire computation of QR decomposition. Both the input `din` and output `dout` signals are parameterizable to any floating-point format using two VHDL generics specified in the NU VFloat library: `exp` and `man`. The `exp` generic specifies the exponent bitwidth and the `man` generic specifies the mantissa bitwidth of the floating-point number. They are both compile-time variables. `rdy` and `done` are synchronization signals. The input signal `rdy` is high for one clock cycle when the first input data `din` is ready. The output `done` is asserted during the entire period when each element of the decomposed matrix R is streamed out on `dout`. The output signal `exc` is raised when an exception occurs during the computation of QR.

The Main features of our QR core are:

- Unlike all previous work, divide and square root operation in Givens rotations are not avoided in our QR decomposition. Instead, we implement the Givens rotations algorithm using the floating-point divide and square root from the VFloat library.

- The QR core has a two dimensional systolic array architecture while most other implementations only support a linear (one dimension) systolic array.

- All operations are in floating-point arithmetic to accommodate the dynamic range of the input data. Furthermore, any size floating-point format with any bitwidth exponent and mantissa, including IEEE standard formats, are supported.
- The maximum level of parallelism is explored in our implementation for the Xilinx Virtex5 FPGA that we are currently targeting. Also, we balance the usage of all hardware resources such as slices and embedded multipliers.
- The QR core is fully pipelined with high throughput and fast clock rate.
- The input matrix size can be configured at compile-time to virtually any size within the limitation of the available hardware resources. Unlike other work that supports only square matrices or tall matrices, our QR decomposition works for an input matrix of any size: square, tall or short.
- Our design is easily scalable to future larger FPGA devices, or over multiple FPGAs.
- No special operations or number system are used in our QR core. This guarantees better accuracy and less conversion overhead. More importantly, this makes our design more standard and portable to different systems, thus easier to fit into a larger system.

6.4 Experimental Results

The complete 2D systolic array for QRD was designed targeting a Xilinx Virtex5 FPGA. The design is written in VHDL, synthesized using Synplify Pro 8.8; and the bitstream is generated using Xilinx ISE 9.1i. Both input matrix data and decomposed output matrix data, as well as all operations, are in floating-point format. Our QR hardware module supports any size floating-point format with any bitwidth exponent and mantissa including

IEEE standard formats. In this section, we highlight two floating-point formats: 32(8,23) and 20(8,11). The first one is the IEEE single-precision format with 8-bit exponent and 23-bit mantissa; the second is a smaller 20-bit format with 8-bit exponent and 11-bit mantissa.

To achieve maximum parallelism, we explore fitting as many PEs as possible on a XC5VLX220 FPGA, which has 138240 slices, 192 blockRAMs, and 128 embedded DSPs. For an $m < n$ short matrix, resources vary with both the number of rows and columns. For an $m > n$ tall matrix, the resources are almost independent of the number of rows. A very tall matrix with $m >> n$ requires about the same resources as an $n \times n$ square matrix.

The size of matrix that could be processed was limited by the resources on the FPGA. Note that there are a lot of multipliers in our design. These not only include the two floating-point multipliers for each diagonal PE and the four floating-point multipliers for each off-diagonal PE, but also those multipliers implicitly included in the floating-point divider. There are three options to implement all these multipliers: using FPGA LUTs, using embedded multipliers on the FPGA chip, or using a mix of embedded multipliers and LUTs. In this work, we explore the balance of all FPGA resources, especially the on-chip embedded multipliers and LUTs so that our implementation is not limited by one FPGA resource while leaving a lot of other spare resources. Our experiments show that a Xilinx Virtex5 XC5VLX220 FPGA can fit a fully 2D parallel systolic array implementation of an input matrix with up to 7 columns for the 32-bit IEEE single-precision format and up to 12 columns for the 20-bit floating-point format.

Figure 6.6 shows the schedule for a 2D systolic array implementation for a 6×3 input matrix. Time is on the x axis and operations are on the y axis. All timing data in the following analysis assume the IEEE single-precision format. The 2D array for the 6×3 matrix has 3 rows: the first row has one diagonal PE r_1 and two off-diagonal PEs $r_{1,2}$ and $r_{1,3}$; the second row has one diagonal PE r_2 and one off-diagonal PE $r_{2,3}$; the third row had only one diagonal PE r_3. The diagonal PEs generate the rotation angles, which

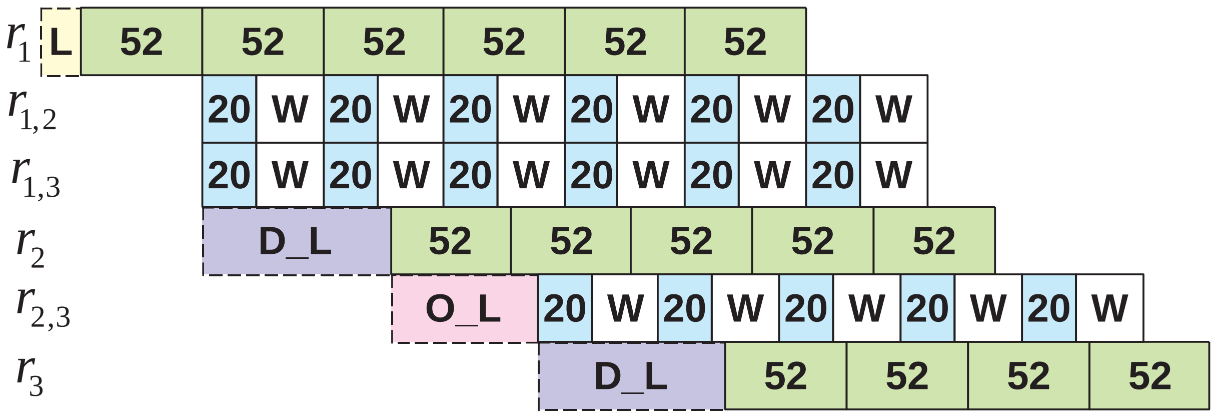


Figure 6.6: Scheduling of the 2D Systolic Array

takes 52 clock cycles (shown in Figure 6.6). The off-diagonal PEs update the two rows involved, which takes 20 clock cycles. Since the design is fully pipelined, the off-diagonal PEs wait during 32 clock cycles of the 52 cycles used by the diagonal PEs. This is shown as the **W** block in Figure 6.6. The latency of the diagonal PE is longer than that of the off-diagonal PE because the diagonal PE includes the complex floating-point divide and square root operations while the off-diagonal PE only has multiplication and addition/subtraction operations. For all diagonal PEs, there is some initial data load time; this time is shown as the **L** block for the first diagonal PE r_1 and the **D_L** blocks for the other diagonal PEs in Figure 6.6. All off-diagonal PEs also have initial data load time, shown as the **O_L** block in Figure 6.6.

When the QR operation begins, the diagonal PE r_1 starts computing first, and when completed, sends the rotation angles to the two off-diagonal PEs $r_{1,2}$ and $r_{1,3}$ in the same row. This takes about 52 clock cycles. Upon receiving rotation parameters, $r_{1,2}$ and $r_{1,3}$ start processing simultaneously. Meanwhile, the diagonal PE r_1 is working on the next two rows to generate another set of rotation parameters. $r_{1,2}$ and $r_{1,3}$ update the two row elements involved in one Givens rotation. This update takes 20 clock cycles. When it is done, $r_{1,2}$ sends the updated value to the diagonal PE r_2 in the second row, triggering the processing of r_2. While r_2 is running, all PEs in the first row r_1, $r_{1,2}$ and $r_{1,3}$ are running as well, but on the new data. The same pattern continues. When the last PE r_3 starts processing, all PEs in the systolic array are all running simultaneously, on different rows of data. One thing we should note is that all PEs in the first row r_1, $r_{1,2}$ and $r_{1,3}$ have to perform 6 Givens rotations for an input matrix with 6 rows. However, the PEs in the second row only need perform 5 rotations, and 4 rotations for the third row. Each rotation takes 52 clock cycles. At the end, the PEs in the first row stop first and the PE/PEs in the last row stop last. The larger the matrix size, the longer this overlapping period with all PEs running together. If the input matrix size is large enough, except for the very beginning and the very end, most of the time all PEs are running simultaneously. This type of parallelism contributes to the short latency of our 2D systolic array implementation. The 2D implementation has a nice linear latency $O(n)$.

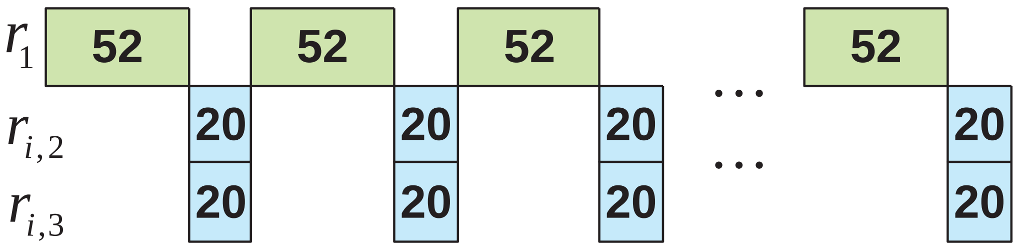


Figure 6.7: Scheduling of the 1D Systolic Array

Figure 6.7 shows the scheduling of a 1D (linear) systolic array implementation for the same 6×3 input matrix for comparison. In this schedule, wait times have been omitted. For the 1D array, instead of the whole triangular two dimensional array, only one row of processing elements is instantiated. For an input matrix with 3 columns, the 1D systolic array consists of only one diagonal PE r_1 and two off-diagonal PEs $r_{i,2}$ and $r_{i,3}$. For one Givens rotation, the diagonal PE r_1 generates the rotation angles first (this takes 52 clock cycles); then the off-diagonal PEs $r_{i,2}$ and $r_{i,3}$ update the corresponding two rows (this takes 20 clock cycles). The whole process starts over again for another rotation. There is no overlap between PEs as in the 2D implementation. The only overlap is between all off-diagonal PEs to update all elements in the two rows. The major advantage of the 2D implementation vs. the 1D implementation is the overlapping processing of all PEs in the array. The 1D implementation has a much longer quadratic latency $O(n^2)$.

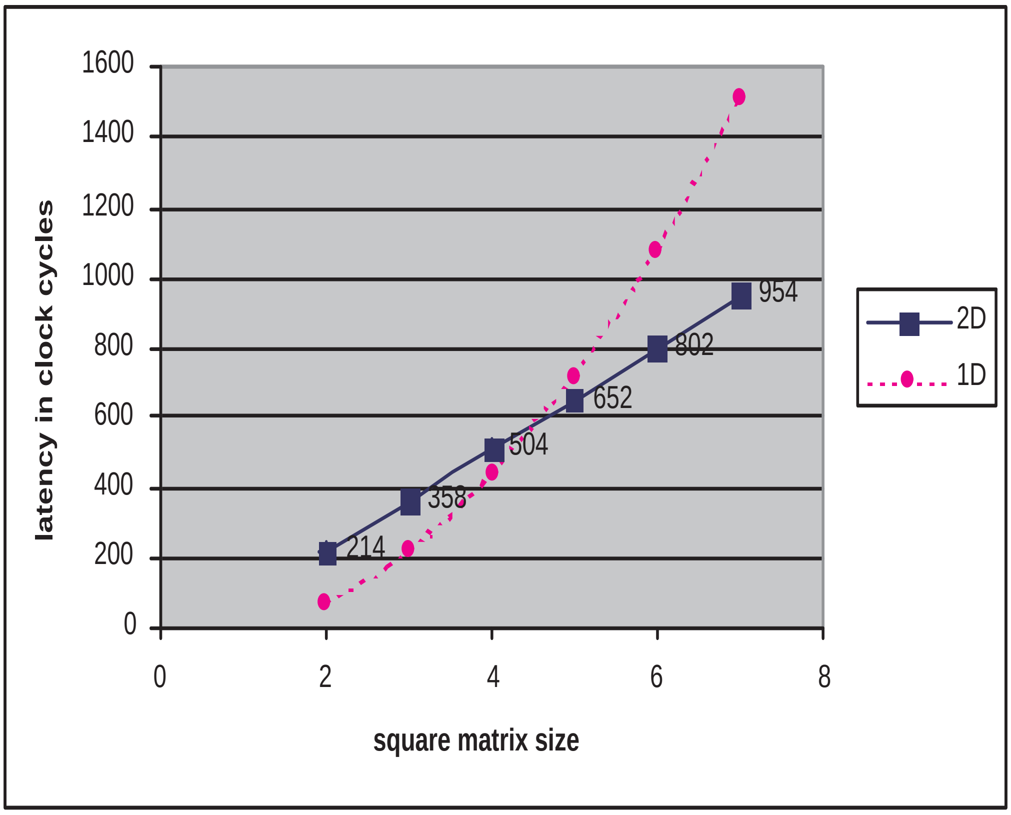


Figure 6.8: Latency vs. Matrix Size

Figure 6.8 shows the latency of a square matrix as its size varies from 2×2 to 7×7 for both the 1D and 2D implementations for the IEEE single-precision floating-point format. We estimate the latency of a one dimensional systolic array implementation for comparison. To zero out half of a $n \times n$ input matrix, which is about $\frac{n*(n-1)}{2}$ elements, the delay is $(52 + 20) \times \frac{n \times (n-1)}{2}$ clock cycles for the 1D array. For a 7×7 matrix, it is about 1512 clock cycles, which is longer than the 954 clock cycles realized in our parallel implementation.

Table 6.1: Resources and Speed: XC5VLX220

Format	Matrix size	Slices	BlockRAMs	DSPs	Freq. (MHz)	2D Latency (cycles)	1D Latency (cycles)
32(8,23)	7×7	126585	56	102	132	954	1512
20(8,11)	12×12	120094	30	106	139	1412	4290

Table 6.1 gives the resources and speed of the Xilinx XC5VLX220 for a 7×7 matrix with 32-bit IEEE single-precision format and a 12×12 matrix with 20-bit format with 8-bit exponent and 11-bit mantissa. The QR decomposition architecture mapped to a 2D interconnected array of processors is fully pipelined. Both implementations have high throughput: 132MHz for IEEE single-precision format and 139MHz for the 20-bit format. The last column of Table 6.1 is the estimated latency for the 1D implementation. Note that for a larger 12×12 matrix with 20-bit format, the latency of the 1D implementation is much longer than that of the 2D implementation because it increases quadratically instead of linearly with matrix size. For the 20-bit data format, the latency of the diagonal PE is 45 clock cycles and the latency of the off-diagonal PE is 20 clock cycles. To zero out half of a 12×12 input matrix, which is about $\frac{12*(12-1)}{2} = 66$ elements, the delay is $(45 + 20) \times 66 = 4290$ clock cycles for the 1D array. For small bitwidth floating-point data, the advantage of our 2D systolic array hardware implementation compared to the 1D implementation is significant. Thus, for applications that need the dynamic range of

floating-point but not the precision of IEEE single-precision data, we can use a smaller floating-point format to achieve better performance and save more chip area. This further demonstrates the usefulness of having variable-precision floating-point modules instead of standard IEEE formats floating-point modules.

We estimate the peak performance of our 2D systolic array QR implementation for the 12×12 input matrix with 20-bit floating-point data. For a 12×12 matrix, there are a total of 78 PEs in the 2D systolic array. At peak, 42 PEs are running at the same time. For a larger matrix, we shall see all PEs running simultaneously for some period of time in the middle of the computation. As we discussed before, the larger the matrix, the longer this overlapping period is. From Figure 6.3, we see that each diagonal PE has 6 single-precision floating-point operations: two multiplications, one addition, one square root, two divisions. From Figure 6.4, we see that each off-diagonal PE also has 6 floating-point operations: four multiplications, one addition, and one subtraction. The peak performance of our QR decomposition can be computed as $6 * 42 * 139 * 10^6 = 35$GFLOPs. The average performance is about half of the peak performance for a 12×12 input matrix.

6.4.1 Discussion

The largest matrix size that can fit on a given device depends on the degree of parallelism, the data wordlength in the implementation, and the available resources. In this work, we explore the maximum degree of parallelism to obtain the best performance on one Xilinx XC5VLX220 FPGA. However, there are a wide range of configurations from a fully parallel array of processor implementations such as ours to a single processing element implementation. The performance of the implementation is proportional to the number of processing elements, which is determined by the hardware resources. Therefore, increased performance is at the cost of increased device size. For large matrices, a larger FPGA or multiple FPGAs become the choice to achieve better performance.

Our 2D systolic array architecture is highly scalable since each of the PEs requires only its nearest neighbor interconnections. This interconnected array of processors structure scales well with matrix size, and works for matrices of virtually any size within the limitations of hardware resources. Especially for large matrices, it may be desirable to extend the design to use a set of FPGA devices. Since there is no global communication through all PEs of the 2D systolic array, the data transfer overhead of multiple FPGAs will not be too significant. We can keep the PEs for the same row in the same FPGA and PEs for neighboring rows in neighboring FPGAs if they do not fit in one FPGA. The data transfer only happens once in one direction between the two neighboring FPGAs. Assuming N FPGAs, there are $N - 1$ data transfers. This data transfer increases the idling time of some PEs (the first row in an additional FPGA) before they can start computation. These times are shown in the dashed blocks in Figure 6.6. For large matrix size requiring multiple FPGAs, this idling period becomes less significant and the overlapping period with all PEs running simultaneously becomes more significant.

In this work, the largest matrix size that can fit on one Xilinx XC5VLX220 FPGA is limited by the on-chip embedded multipliers because a large number of multipliers are needed in the 2D systolic array design. Addition and multiplication are the two fundamental operations in almost any signal processing application. However, multipliers are typically much larger than adders in hardware especially for large sizes. In future FPGA architectures, it would be nice to have a larger portion of embedded multipliers with better optimized performance. To overcome the multiply bottleneck in our design, we use a mix of embedded multipliers and FPGA LUTs for the multipliers in our design. In this way, we can balance the usage of different resources. The 7×7 input matrix with 32-bit IEEE single-precision data format uses 91% LUTs and 80% embedded multipliers. The 12×12

input matrix with 20-bit data format uses 87% LUTs and 83% embedded multipliers. Xilinx LogiCORE [107] multipliers are used in this work. The exploration to balance FPGA resources takes quiet a while. For every single multiplier in our design, we experimented with different implementations - using FPGA LUTs, using embedded multipliers, or using a mix of embedded multipliers and LUTs. The Xilinx LogiCORE allows the designer to choose among the three major types of multiply implementation. However, the designer cannot fine tune the percentage of embedded multipliers and LUTs globally. In the future, it would be nice if the tools could better support both coarse-grained and fine-grained breakdown of embedded multipliers and LUTs for multiply implementations.

6.5 Summary

An FPGA-based systolic array implementation of QR decomposition using Givens Rotations is presented in this chapter. The maximum available parallelism is explored for a Xilinx XC5VLX220 FPGA with balanced usage of hardware resources such as slices and embedded multipliers – at least 80% of all resources are used in our designs. Input, output and all operations are in floating-point arithmetic; any size floating-point format including IEEE standard formats are supported. Floating-point divide and square root are not avoided in our QR decomposition implementation, demonstrating the use of the NU VFloat library. Compared to other implementations that use special operations such as CORDIC or special number systems such as logarithmic number system (LNS), this straightforward implementation has the advantage of better portability. The input matrix size can be configured at compile-time to virtually any size. We support square, tall and short matrices. In this work, a truly two dimensional systolic array architecture was implemented. Therefore, the latency of our systolic array implementation only increases linearly with matrix size, making it scale well for larger matrices. The QR decomposition is fully pipelined with high throughput, fast clock rate and high maximum frequency, making it suitable for high-speed FPGA implementation with a peak performance of 35 GFLOPs and average performance of 18 GFLOPs. In the next chapter, we present conclusions and discuss the future plans for our research.

Chapter 7

Conclusions and Future Work

7.1 Summary

This dissertation has provided a description of the floating-point divide and square root implementations on modern FPGAs with embedded multipliers and on-chip BlockRAMs. They are standard modules just like other floating-point IP cores, and can aid in designing those applications with dynamic data ranges that require floating-point computations. Having high performance floating-point divide and square root modules solves the bottleneck of many applications that require divide and/or square root operations. Both the exponent and mantissa of our floating-point divide and square root are parameterized at compile time, enabling any arbitrary floating-point formats including the IEEE 754 standard single-precision and double-precision formats. This gives designers the flexibility and convenience to explore the optimal bitwidths of each operation for a specific application. Custom datapaths with optimal bitwidths for each divide and square root operation can be built using our hardware modules. This allows more parallelism and makes possible fast circuitry with smaller area and latency as well as lower power dissipation than adhering to a standard format.

Unlike most other implementations that use digit recurrence or iterative algorithms, the divide and square root designs in this research are based on table lookup and Taylor series expansion, and make use of memories and multipliers embedded on the FPGA chip. These algorithms are well suited to FPGA implementations and exhibit a good tradeoff between area, latency and throughput. They are all fully pipelined, so can easily fit into the pipeline of a large design. Our divide and square root have very low latency compared to most other divide and square root implementations including the Xilinx LogiCORE Floating-point modules, especially for large bitwidth format. Compared to the latency of other implementations that increases linearly as the mantissa bitwidth increases, the latency of our implementations increases very slowly with the mantissa bitwdith due to the algorithms used.

Our floating-point divide and square root, along with the NU VFloat library, provides a tool that makes the design of these algorithms easy. More importantly, they allow for hardware implementation of many signal and image processing applications that require floating-point divide and/or square root. Our floating-point divide and square root implementations outperform most other implementation including the Xilinx LogiCORE Floating-point Operators. Therefore, it is not only feasible but also practical to use them in the real high performance computing.

Two different applications are presented to demonstrate the usefulness of our floating-point divide and square root modules. The applications selected were implemented in such a way that divide and square root operations were avoided by using special operations or special number systems; or the computational part excluding floating-point divide and/or square root is accelerated on an FPGA while the floating-point division and square root are done on the host processor. Now with floating-point divide and square root modules, we can further accelerate the run-time of these hardware implementations. Furthermore, designs using our floating-point modules are more standard and portable than those use CORDIC or LNS.

The first application presented is K-means clustering for multispectral satellite images using our floating-point divide. Our implementation is the first one that implements the complete K-means computation in FPGA hardware. The results on a Mercury VantageRT Reconfigurable Computing Board [73] with Xilinx Virtex-II XC2V6000 FPGA show that the hardware implementation achieves a speedup of over 2150x for core computation time and about 11x for total run-time including data transfer time. They also show that the divide in FPGA hardware is 100 times faster than in software. Speedup is obtained by moving the mean updating step from the host PC to FPGA hardware, eliminating the data transfer overhead between the host and the FPGA for each iteration. More importantly, this relieves the host to work on other tasks concurrently with K-means clustering, leading to greater potential speedup due to the coarse-grained parallelism.

The second application is QR decomposition using both our floating-point divide and square root. Our implementation does not avoid the floating-point divide and square root by using CORDIC or LNS. This is not only straightforward, but also makes our design more standard and portable to different systems, thus easier to fit into a larger system. In this work, we explore the maximum degree of parallelism to obtain the best performance on one Xilinx XC5VLX220 FPGA. This is achieved by first using a truly two dimensional systolic array architecture with a portion of overlap time when all processing elements of the systolic array are running simultaneously. Secondly, acceleration is achieved by custom datapaths with optimal bitwidth for each operation in reconfigurable hardware to further increase this fine-grained parallelism. Moreover, any dynamic range and any arbitrary format input data is supported with the variable precision floating-point divide and square root, along with the NU VFloat library. The latency of our 2D systolic array implementation increases linearly with matrix size instead of quadratically with a 1D systolic array implementation. Thus, it scales well for larger matrices. For small bitwidth floating-point data, the advantage of our 2D systolic array hardware implementation compared to the 1D implementation is significant. Thus, custom floating-point formats using the variable-precision features of the modules in the NU VFloat library allow a user to achieve better performance and save more chip area. The QR implementation is fully pipelined with high throughput and fast clock rate. The peak performance for a 20-bit data 12×12 input matrix is about 35 GFLOPs and the average performance is about 18 GFLOPs. Our QR design is also easily scalable to future larger FPGA devices, or over multiple FPGAs.

7.2 Future Research

There are several possible directions for this research in the future.

First, regarding to the floating-point divide and square root along with the NU VFloat library, we plan to explore divide and square root implementations using other small multiplier and look-up table based algorithms. There are a number of other series that can convergence faster than Taylor series expansion such as Chebyshev Series. The Chebyshev series has many advantages:

- The Chebyshev series usually converges rapidly [45].
- The Chebyshev series approximation is accurate and numerically more stable than other approximations. It is the most accurate approximation among all polynomials of the same degree so higher accuracy is maintained [80].
- Using the Chebyshev series can usually minimize the order of the polynomial approximation, which means fewer terms are required to obtain an equally accurate estimate [5, 88].

- All coefficients of the Chebyshev series are nicely bounded between [-1,1], so it is less susceptible to rounding errors [80].

Therefore, using the Chebyshev series can possibly achieve the same precision with less hardware resources and better performance.

In addition to floating-point divide and square root, more components can be added to the NU VFloat library including multiply accumulate (MAC) and inverse square root. Multiply accumulate is an essential operations in many DSP algorithms such as FIR filter, and adding the MAC module can benefit the hardware implementations of these applications. Compared to MAC, the inverse of the square root operation does not occur as often. However, it is frequently the bottleneck of any algorithm in which it does occur due to its complexity. For example, in the Givens rotations of our QR decomposition, we use a divide followed by a square root to implement the inverse square root operation. Those are the most complicated part of our hardware implementation, and consume most of the hardware resources. In the future, we can replace the square root followed by divide operations in the Givens rotations algorithm with one inverse square root operation. One option is to use an algorithm [26] similar to the one that we used for square root to implement the inverse square root. In this case, similar hardware resource usage and performance is expected.

There are a few key areas of improvement for our QR decomposition. First, as we mentioned earlier, the Givens rotations algorithm can benefit from replacing the divide followed by square root operation by one inverse square root operation. Since the new inverse square root module will be similar to the square root module, this can potentially reduce both the area and latency of the diagonal PE by approximately half. Secondly, to support the fully two dimensional systolic array implementation for larger input matrices, it is desirable to implement the QR decomposition over multiple FPGA devices considering the density of current FPGA technology. Additional work will target several reconfigurable computing engines for our QR decomposition implementation with some real hardware results.

Another algorithm we plan to address is solving singular value decomposition (SVD) using the Givens rotations algorithm. Singular value decomposition is similar to a QR decomposition on both sides of the input matrix. Therefore, it shares a lot of similarity with QR decomposition but with double the complexity. Due to the complexity of the algorithms, it is difficult to implement on FPGAs. The successful implementation of QR decomposition demonstrates that SVD on FPGAs is possible with reasonable performance.

Last but not least, in the future, using our floating-point divide and square root, our long range plan is to explore more DSP applications that were originally limited by the floating-point divide and/or square root as the major performance bottleneck.

Bibliography

[1] Implementing Matrix Inversions in Fixed-Point Hardware. http://www.xilinx.com/publications/magazines/dsp_01/xc_pdf/p32-35_dsp-matrix.pdf.

[2] http://www.xilinx.com/ise/dsp_design_prod/acceldsp/accelware/.

[3] Researching FPGA Implementations of Baseband MIMO Algorithms using AccelDSP. http://www.xilinx.com/products/design_resources/dsp_central/resource/wimax_mimo_acceldsp.pdf.

[4] Actel Incorporated. *ACTTM Family Field Programmable Gate Array Data book*, 1992.

[5] R. C. Agarwal, F. G. Gustavson, and M. S. Schmookler. Series Approximation Methods for Divide and Square root in the Power3^{TM} Processor. In *Proc. 14th IEEE Symposium on Computer Arithmetic*, pages 116–123, 1999.

[6] http://www.altera.com/.

[7] Altera Corporation. *Altera FLEX 8000 Programmable Logic Device Family Data Sheet*.

[8] http://www.altera.com/.

[9] Altera Wireless Solutions Channel Card SeriesCellular Infrastructure. http://www.altera.com/literature/po/ss_wrless_cellular_infra.pdf.

[10] Smart Antennas - Beamforming. http://www.altera.com/end-markets/wireless/advanced-dsp/beamforming/wir-beamforming.html.

[11] ALTERA CORDIC Reference Design. `http://www.altera.com/literature/an/an263.pdf`.

[12] Accelerating WiMAX System Design with FPGAs. `http://www.altera.com/literature/wp/wp_wimax.pdf`.

[13] Handel-C, Software-Compiled System Design. `http://www.celoxica.com/`.

[14] A VHDL library of parametrisable floating-point and LNS operators for FPGA. `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/`.

[15] P. Belanović. Library of Parameterized Hardware Modules for Floating-Point Arithmetic with an Example Application. M.S. Thesis, Dept. of Electrical and Computer Eng., Northeastern Univ., 2002.

[16] P. Belanović and M. Leeser. A library of parameterized floating-point modules and their use. In *12th International Conference on Field Programmable Logic and Application (FPL'02)*, pages 657–666, September 2002.

[17] D. Boppana, K. Dhanoa, and J. Kempa. FPGA based Embedded Processing Architecture for the QRD-RLS Algorithm. In *12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pages 330–331, April 2004.

[18] G. A. Covington, C. L.G. Comstock, A. A. Levine, J. W. Lockwood, and Y. H. Cho. High speed document clustering in reconifigurable hardware. In *16th Annual Conference on Field Programmable Logic and Applications (FPL'06)*, pages 411–417, Aug. 2006.

[19] J. Detrey and F. de Dinechin. A VHDL library of LNS operators. In *37th Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 2227–2231, Pacific Grove, California, USA, November 2003. IEEE Signal Processing Society.

[20] J. Dido, N. Geraudie, et al. A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 50–55, February 2002.

[21] `http://www.digitalcoredesign.com/`.

[22] P. C. Diniz and G. Govindu. Design of Field-Programmable Dual-precision Floating-Point Arithmetic Units. In *Proceedings of the 16th international conference on field-programmable logic and applications (FPL'06)*, pages 733–736, August 2006.

[23] R. Dohler. Squared Givens Rotations. *IMA Journal of Numerical Analysis*, 11(1):1–5, 1991.

[24] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*, chapter 10. John Wiley & Sons, Inc., New York, NY, 2001.

[25] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Mass., 1994.

[26] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, 49(7):628–637, July 2000.

[27] M. Estlick, M. Leeser, J. Theiler, and J. Szymanski. Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware. In *Proceedings of the Ninth International Symposium on Field Programmable Gate Arrays*, pages 103–110. ACM, February 2001.

[28] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on VLSI Systems*, 2(3):365–367, September 1994.

[29] A. Filho, A. C. Frery, C. C. de Arajo, H. Alice, J. Cerqueira, J. A. Loureiro, M. E. de Lima, M. das Graas S. Oliveira, and M. M. Horta. Hyperspectral images clustering on reconfigurable hardware using the K-means algorithm. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI'03)*, pages 99–104, Sept. 2003.

[30] M. P. Fitton, S. Perry, and R. Jackson. Reconfigurable Antenna Processing with Matrix Decomposition using FPGA based Application Specific Integrated Processors. In *2004 Software Defined Radio Technical Conference*, Nov. 2004.

[31] J. G. F. Francis. The QR transformation. *The Computer Journal*, 4(4):332–345, 1962.

[32] C. V. Freiman. Statistical analysis of certain binary division algorithms. In *IRE Proc.*, volume 49, pages 91–103, 1961.

[33] A. A. Gaar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang. Automating customisation of floating-point designs. In *Proceedings of the 12th international conference on field-programmable logic and applications (FPL'02)*, pages 523–533, Sept. 2002.

[34] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. K. Cheung. Unifying Bit-width Optimisation for Fixed-point and Floating-point Designs. In *12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 79–88, April 2004.

[35] A. A. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi. Floating Point Bitwidth Analysis via Automatic Differentiation. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT'02)*, pages 158–165, December 2002.

[36] M. Gay and I. Phillips. Real-time adaptive beamforming – FPGA implementation using QR decomposition. *Journal of Electronic Defense*, September 2005.

[37] W. M. Gentleman and H. T. Kung. Matrix Triangularization by Systolic Arrays. In *Real-time Signal Processing IV, Proc. SPIE 298*, pages 19–26, 1981.

[38] W. Givens. Computation of plane unitary rotations transforming a general matrix to triangular form. In *J. Soc. Indust. Appl. Math.*, volume 6, pages 26–50, 1958.

[39] C. S. Gloster, Jr. and I. Sahin. Floating-Point Modules Targeted for Use with RC Compilation Tools. In *Earth Science Technology Conference (ESTC)*, August 2001.

[40] M. Gokhale, J. Frigo, K. McCabe, J. Theiler, C. Wolinski, and D. Lavenier. Experience with a hybrid processor: K-Means clustering. *The Journal of Supercomputing*, 26(2):131–148, Sept. 2003.

[41] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[42] R. Goldbert, G. Even, and P.-M. Seidel. An FPGA Implementation of Pipelined Multiplicative Division with IEEE Rounding. In *IEEE Symposium on Field-programmable Custom Computing Machines (FCCM '07)*, Apr. 2007.

[43] G. H. Golub and C. F. Van Loan. *Matrix Computations*, chapter 5. Johns Hopkins University Press, 1996.

[44] G. Govindu, R. Scrofano, and V. K. Prasanna. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In *The 2005 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2005)*, June 2005.

[45] J. F. Hart. *Computer Approximations*, chapter 3. John wiley & Sons, Inc., New York, 1968.

[46] A. Hermanek, J. Schier, and P. Regalia. Architecture design for FPGA implementation of finite interval CMA. In *Proceedings of the 12th European Signal Processing Conference*, pages 1–4, 2004.

[47] A. S. Householder. Unitary Triangularization of a Nonsymmetric Matrix. *Journal of the ACM (JACM)*, 5(4):339 – 342, Oct. 1958.

[48] P. Hung, H. Fahmy, O. Mencer, and M. J. Flynn. Fast division algorithm with a small lookup table. In *33rd Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1465–1468, Oct. 1999.

[49] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A cad suite for high-performance FPGA design. In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–24, Napa, CA, April 1999. IEEE Computer Society, IEEE.

[50] IEEE Standards Board and ANSI. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Press, 1985. IEEE Std 754-1985.

[51] A. Jaenicke and W. Luk. Parameterised floating-point arithmetic on FPGAs. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '01)*, volume 2, pages 897–900, May 2001.

[52] M. Karkooti, J.R. Cavallaro, and C. Dick. FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm. In *39rd Asilomar Conference on Signals, Systems and Computers*, pages 1625–1629, Oct. 2005.

[53] D. Lavenier. FPGA implementation of the KMeans clustering algorithm for hyperspectral images. Los Alamos National Laboratory LAUR 00-3079, 2000.

[54] B. Lee and N. Burgess. Parameterisable floating-point operations on FPGA. In *36th Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1064–1068. IEEE Signal Processing Society, November 2002.

[55] M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. J. Szymanski, and J. Theiler. Applying reconfigurable hardware to the analysis of multispectral and hyperspectral imagery. In *Proc. SPIE 4480*, pages 100–107, 2001.

[56] M. Leeser, J. Theiler, M. Estlick, N. Kitaryeva, and J. J. Szymanski. Effect of data truncation in an implementation of pixel clustering on a custom computing machine. In *Reconfigurable Technology for Computing and Applications II, Proc. SPIE 4212*, pages 80–89, 2000.

[57] M. Leeser, J. Theiler, M. Estlick, and J. J. Szymanski. Design tradeoffs in a hardware implementation of the K-means clustering algorithm. In *Proc. SAM: First IEEE Sensor Array and Multichannel Signal Processing Workshop*, pages 520–524, March 2000.

[58] M. P. Leong, M. Y. Yeung, C. K. Yeung, C. W. Fu, P. A. Heng, and P. H. W. Leong. Automatic Floating to Fixed Point Translation and its Application to Post-Rendering 3D Warping. In *7th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '99)*, pages 240–248, April 1999.

[59] Y. Li and W. Chu. Implementation of single precision floating point square root on FPGAs. In *5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, pages 226–232, April 1997.

[60] J. Liang, R. Tessier, and O. Mencer. Floating point unit generation and evaluation for FPGAs. In *11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 185–194, April 2003.

[61] G. Lienhart, A. Kugel, and R. Manner. Using floating-point arithmetic on FPGAs to accelerate scientific N-Body simulations. In *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, pages 182–191, April 2002.

[62] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '98)*, pages 206–215, April 1998.

[63] J. Liu, M. Chang, and C.-K. Cheng. An iterative division algorithm for FPGAs. In *Fourteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'06)*, pages 83–89, February 2006.

[64] L. Louca, T. A. Cook, and W. H. Johnson. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96)*, pages 107–116, April 1996.

[65] M. E. Louie and M. D. Ercegovac. Mapping division algorithms to field programmalbe gate arrays. In *1992 Conference Record of the 26th Asilomar Conference on Signals, Systems and Computers*, pages 371–375, 1992.

[66] M. E. Louie and M. D. Ercegovac. A digit-recurrence square root implementation for field programmable gate arrays. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 178–183, 1993.

[67] M. E. Louie and M. D. Ercegovac. On digit-recurrence division implementations for field programmalbe gate arrays. In *Proc. 11th Symposium on Computer Arithmetic*, pages 202–209, 1993.

[68] B. Louka and M. Tchuente. Givens elimination on systolic arrays. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 638–647, 1988.

[69] T. Maruyama. Real-time K-Means clustering for color images on reconfigurable hardware. In *Proceedings of the 18th International Conference on Pattern Recognition*, volume 2, pages 816–819, 2006.

[70] R. Matousek, M. Tich, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman. Logarithmic Number System and Floating-Point Arithmetics on FPGA. In *Lecture Notes in Computer Science*, volume 2438, pages 175–188. Springer, 2002.

[71] O. Mencer. PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs. In *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, pages 67–76, April 2002.

[72] O. Mencer, M. Platzner, M. Morf, and M. J. Flynn. Object-oriented domain specific compilers for programming FPGA's. *IEEE Transactions on VLSI Systems*, 9(1):205–210, February 2001.

[73] `http://www.mc.com/`.

[74] T. K. Moon and W. C. Stirling. *Mathematical Methods and Algorithms for Signal Processing*, chapter 5. Prentice Hall International, 2000.

[75] `http://www.nallatech.com/`.

[76] Variable Precision Floating-Point Modules. `http://www.ece.neu.edu/groups/rcl/projects/floatingpoint/`.

[77] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. In *IEEE Transactions on Computers*, pages 154–161, 1997.

[78] I. Ortiz and M. Jimenez. Scalable pipeline insertion in floating-point division and square root units. In *Proceedings of the 2004 47th Midwest Symposium on Circuits and Systems (MWSCAS'04)*, volume 2, pages II–225–II–228, July 2004.

[79] S. Paschalakis and P. Lee. Double precision floating-point arithmetic on FPGAs. In *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 352–358, December 2003.

[80] W. H. Press, S. A. Teukilsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing, Second Edition*, chapter 5. Cambridge University Press, Cambridge, UK, 2002.

[81] Quixilica Floating Point Cores. `http://www.tekmicro.com/`.

[82] Quixilica Floating-Point QR Processor Core. `http://www.eonic.co.kr/data/datasheet/transtech/FPGA/qx_qr.pdf`.

[83] J. R. Rice. Experiments on Gram-Schmidt Orthogonalization. *Mathematics of Computation*, 20(94):325–328, 1966.

[84] J. E. Robertson. A new class of digital division methods. *IRE Trans. Electronic Computers*, 7:218–222, September 1958.

[85] E. Roesler and B. E. Nelson. Novel optimizations for hardware floating-point units in a modern FPGA architecture. In *12th International Conference on Field-Programmable Logic and Applications (FPL'02)*, pages 637–646, Sept. 2002.

[86] T. Saegusa and T. Maruyama. An FPGA implementation of K-means clustering for color images based on KD-Tree. In *16th Annual Conference on Field Programmable Logic and Applications (FPL'006)*, Aug. 2006.

[87] I. Sahin, C. S. Gloster, and C. Doss. Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems. In *2000 MAPLD International Conference*, September 2000.

[88] J. Sawada. Formal Verification of Divide and Square Root Algorithms Using Series Calculation. In *Proceedings of ACL2 Workshop*, 2002.

[89] J. Schier and A. Hermanek. Using logarithmic arithmetic to implement the recursive least squares (QR) algorithm in FPGA. In *14th International Conference on Field-Programmable Logic and Applications (FPL'04)*, pages 1149–1151, August 2004.

[90] J. Schier and J. Kadlec. Using logarithmic arithmetic for FPGA implementatin of the Givens Rotations. In *Proceedings of the Sixth Baiona Workshop on Signal Proceeding in Communications*, pages 199–204, September 2003.

[91] A. Sergyienko and O. Maslennikov. Implementation of Givens QR-Decomposition in FPGA. In *Lecture Notes in Computer Science*, volume 2328, pages 458–465. Springer, 2002.

[92] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 155–162, April 1995.

[93] P. Soderquist and M. Leeser. Division and square root: choosing the right implementation. *IEEE Micro*, 17(4):56–66, July/August 1997.

[94] I. Stamoulis, M. White, and P. F. Lister. Pipelined Floating-Point Arithmetic Optimized for FPGA Architectures. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL)*, volume 1673 of *LNCS*, pages 365–370, August-September 1999.

[95] P. Sucha, Z. Hanzalek, A. Hermanek, and J. Schier. Efficient FPGA Implementation of Equalizer for Finite Interval Constant Modulus Algorithm. In *IEEE International Symposium on Industrial Embedded Systems, IES'06*, pages 1–10, October 2006.

[96] P. Sucha, Z. Hanzalek, A. Hermanek, and J. Schier. Scheduling of Iterative Algorithms with Matrix Operations for Efficient FPGA Design–Implementation of Finite Interval Constant Modulus Algorithm. *Journal of VLSI Signal Processing Systems*, 46(1):35–53, January 2007.

[97] Sweeney. High-speed arithmetic in binary computers. *Proc. of IRE*, 49:67–91, January 1961.

[98] A. J. Thakkar and A. Ejnioui. Pipelining of double precision floating point division and square root operations. In *ACM Proceedings of the 44th Annual Southeast Regional Conference*, pages 488–493, March 2006.

[99] K. D. Tocher. Techniques of multiplication and division for automatic binary computers. *Quart. J. Mech. Appl. Math.*, 11:364–384, 1958.

[100] K. D. Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays(FPGA)*, pages 171–180, February 2004.

[101] J. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, 1959.

[102] R. L. Walke, R. W. M. Smith, and G. Lightbody. Architectures for Adaptive Weight Calculation on ASIC and FPGA. In *33rd Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1375–1380, Oct. 1999.

[103] R. L. Walke, R. W. M. Smith, and G. Lightbody. 20 GFLOPS QR processor on a Xilinx Virtex-E FPGA. In *Advanced Signal Processing Algorithms, Architectures, and Implementations X, Proc. SPIE*, volume 4116, pages 300–310, Nov. 2000.

[104] X. Wang, S. Braganza, and M. Leeser. Advanced components in the variable precision floating-point library. In *IEEE Symposium on Field-programmable Custom Computing Machines (FCCM '06)*, pages 249–258, Apr. 2006.

[105] X. Wang and B. E. Nelson. Tradeoffs of designing floating-point division and square root on virtex FPGAs. In *11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 195–203, April 2003.

[106] `http://www.xilinx.com/`.

[107] XILINX LogiCORE Floating-Point Operator v3.0 Product Specification. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/floating_point_ds335.pdf.

[108] Xilinx, Inc. *XC4000 Logic Cell Array Family - Technical Data*, 1990. San Jose, California.

[109] Implementing a Real-Time Beamformer on an FPGA Platform. http://www.xilinx.com/publications/xcellonline/xcell_60/xc_pdf/p36-40_60-beam.pdf.

[110] XILINX LogiCORE CORDIC v3.0. `http://www.xilinx.com/bvdocs/ipcenter/data_sheet/cordic.pdf`.