Variable Precision Floating Point Reciprocal, Division and Square Root for Major FPGA Vendors

A Thesis Presented

by

Xin Fang

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

August 7, 2013
© Copyright 2013 by Xin Fang
All Rights Reserved
Abstract

Variable precision floating point operations have various fields of applications including scientific computing and signal processing. Field Programmable Gate Arrays (FPGAs) are a good platform to accelerate such applications because of their flexibility, low development time and cost compared to Application Specific Integrated Circuits (ASICs) and low power consumption compared to Graphics Processing Units (GPUs). Increasingly scientists are interested in variable precision floating point operations not limited to single or double precision operations implemented on FPGAs, in order to make those operations more resource efficient and more suited to their own applications.

Among those operations, the performance of reciprocal, division and square root can differ based on the algorithm implemented. They can highly affect the total performance of the application running them. In this thesis, we improve these three operations using a table based approach. Our implementation is written in Very High Speed Integrated Circuits Hardware Description Language (VHDL) and implemented on FPGAs. These components have been implemented using both Altera and Xilinx development environments, the two major FPGA vendors. Also these implementations provide a good tradeoff among hardware resource utilization, maximum clock frequency and latency. Users can change the latency by adjusting the parameters of the components. In addition to supporting the IEEE 754 standard representations which include single and double precision, these components can be customized by
the user to specify the length of the exponent and the mantissa. This will contribute to more resource and energy efficiency in systems where variable precision is used.
Acknowledgements

I want to thank my advisor Professor Miriam Leeser for her patient guidance and support during this project. Without her instructive comments or the encouragement, I could not have finished this thesis. I would also like to give my appreciation to my committee members Professor Yunsi Fei and Professor Kaushik Chowdhury for their comments and direction. I want to thank my parents for their endless support and love and friends who supported me along the way. Also, I am really appreciative of the professors who taught me always inspired me in many research aspects.
Contents

1 Introduction 1

2 Background 3

2.1 Floating Point Representation 3

2.2 Implementing Advanced Floating Point Components 5

2.2.1 Digit Recurrence Method 6

2.2.2 Iterative Method 8

2.2.3 Table-based Method 9

2.3 Related Work 10

3 Algorithms Description 12

3.1 VFloat Library 12

3.2 Reciprocal and Division 14

3.3 Square Root 16

4 Implementation 18

4.1 Implementation Overview 18
CONTENTS

4.2 Common Elements ........................................ 19
   4.2.1 Denormalizer ........................................ 19
   4.2.2 Round to Normal .................................... 19
4.3 Reciprocal ................................................ 20
4.4 Division .................................................. 21
4.5 Square Root ............................................... 22
4.6 Conclusion ................................................ 24

5 Experimental Results .................................... 25
   5.1 Hardware Components on Virtex 6 ....................... 26
   5.2 Hardware Components on Stratix V ....................... 27
   5.3 Reciprocal ............................................... 28
   5.4 Division ................................................ 31
   5.5 Square Root ............................................ 33

6 Conclusions and Future Work ......................... 36
List of Figures

2.1 SRT Algorithm Diagram ........................................... 7
2.2 Iterative Method ..................................................... 9
3.1 Black Box of Components ......................................... 13
4.1 Top Components ..................................................... 19
4.2 Reciprocal Components ........................................... 20
4.3 Division Components .............................................. 22
4.4 Square Root Components .......................................... 23
5.1 Virtex 6 LXT FPGA Summary ..................................... 26
5.2 Stratix V Architecture ............................................. 28
5.3 Stratix V GX Device Features ..................................... 29
5.4 Simulation Result of Double Precision Reciprocal ............. 29
5.5 Simulation Result of Double Precision Division ................ 31
5.6 Simulation Result of Double Precision Square Root .......... 33
List of Tables

2.1 Composition of Binary Floating Point ........................................ 3
2.2 Exponent Bias in IEEE 754 Representation .................................. 4
5.1 Reciprocal Result and Comparison of Altera .............................. 28
5.2 Division Result and Comparison of Xilinx ................................. 31
5.3 Division Result and Comparison of Altera ................................. 32
5.4 Square Root Result and Comparison of Altera ............................. 34
5.5 Square Root Result and Comparison of Xilinx ............................. 34
Chapter 1

Introduction

Variable precision floating point operations are widely used in many fields of computer and electrical engineering. Floating point arithmetic operations are included in most processing units. There are many floating point operations including addition, subtraction, multiplication, division, reciprocal and square root. The Northeastern Reconfigurable Computing Lab has developed its own variable precision floating point library called VFLOAT [13, 19], which is vendor agnostic, easy to use and has a good tradeoff between hardware resources, maximum clock frequency and latency. Field Programmable Gate Arrays (FPGAs), due to their flexibility, low power consumption and short development time compared to Application Specific ICs (ASICs), are chosen as the platform for the VFloat library to run on. Very-high-speed integrated circuits Hardware Description Language (VHDL) is used to describe these components. Xilinx and Altera are the two main suppliers of programmable logic devices. Each company has its own Integrated Development Environment (IDE). Both IDE from Altera and Xilinx have been used to implement this cross platform
CHAPTER 1. INTRODUCTION

project. VFloat is an open-source library unlike the intellectual property cores the IP Cores from Xilinx or Megacores from Altera, contributing more flexibility to the development process.

In the operations of the variable precision floating point library, reciprocal, division and square root are much more difficult to implement than addition, subtraction and multiplication and their running time is much longer. There is limited ability to improve the performance or the resource usage for operations like addition, subtraction and multiplication. However, there are several methods to choose from to implement reciprocal, division and square root, including digit recurrence, iterative method and table-based method. Improving division and square root can significantly speed up the performance of floating point applications. Our goal is to find a good tradeoff among resource usage on an FPGA, the maximum clock frequency of the operation, and the number of clock cycles latency. This thesis examines a table-based approach to division, reciprocal and square root that achieves this goal.
Chapter 2

Background

2.1 Floating Point Representation

There are two main methods to represent a numeric value in a computer. One is fixed point format, and the other is floating point format. The difference between these two methods is that the former one has a radix point which has a fixed location to separate the integer and fractional parts of the numeric value. The floating point format has three parts to represent the numeric value: the signed bit, the exponent and the mantissa. The advantage of using the floating point format is that floating point can represent a wider range of values than fixed point format when using the same number of bits. The floating point format has been standardized and is widely used for scientific computation. This minimizes anomalies and improves numerical quality.

<table>
<thead>
<tr>
<th>Format</th>
<th>Sign Bit(s)</th>
<th>Exponent Bit(e)</th>
<th>Mantissa Bit(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>1</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>Double Precision</td>
<td>1</td>
<td>11</td>
<td>52</td>
</tr>
<tr>
<td>Quadruple Precision</td>
<td>1</td>
<td>15</td>
<td>112</td>
</tr>
</tbody>
</table>

Table 2.1: Composition of Binary Floating Point
IEEE 754 is the technical standard for floating point computation created by the Institute of Electrical and Electronics Engineers (IEEE) in 1985 and updated in 2008 [2]. In this standard, the arithmetic format of a floating point number is defined as follows. We define $b$ as the base which is either 2 or 10, $s$ as the sign bit, $e$ as the exponent and $c$ as the mantissa. For our project, $b = 2$. Three binary floating point basic formats are single precision, double precision and quadruple precision. Table 2.1 shows the corresponding bit widths of sign, exponent and mantissa for each format. Notice that there are many combinations not included in IEEE 754. An example is sign bit is 1, exponent is 9 bits and mantissa is 30 bits. However using this non standard format could save some resources in a flexible technology like FPGAs and still accomplish the computing task. That is why variable precision floating point is also considered while building our floating point library.

The value of a floating point number is $(-1)^s \times 1.c \times b^{e-bias}$. First, when $s$ is zero, the number is positive. Otherwise, it’s negative. Second, the significand digits is $1.c$ not $c$ because the first bit of the significand digit of any normalized floating point number is always ’1’ and omitting it will save one bit in the representation. Third, to make the exponent range positive and easy to compare, a bias is added to the actually exponent for the exponent representation in IEEE 754. Table 2.2 shows the

<table>
<thead>
<tr>
<th>Format</th>
<th>Exponent in IEEE 754</th>
<th>Exponent Bias</th>
<th>Real Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>1 to 254</td>
<td>127</td>
<td>-126 to 127</td>
</tr>
<tr>
<td>Double Precision</td>
<td>1 to 2046</td>
<td>1023</td>
<td>-1022 to 1023</td>
</tr>
<tr>
<td>Quadruple Precision</td>
<td>1 to 32766</td>
<td>16383</td>
<td>-16382 to 16383</td>
</tr>
</tbody>
</table>

Table 2.2: Exponent Bias in IEEE 754 Representation
2.2. IMPLEMENTING ADVANCED FLOATING POINT COMPONENTS

Another aspect in the IEEE 754 standard is the rounding method. There are four rounding rules included. They are rounding to nearest, rounding toward zero, rounding toward positive infinity and rounding to negative infinity. For rounding to nearest, there are also sub-rules when the number falls midway, which is round to nearest ties to even and round to nearest ties away from zero. Rounding to zero method is known as truncation.

2.2 Implementing Advanced Floating Point Components

In order to implement floating point reciprocal, division and square root components, there are three parts of the floating point number to consider based on the previous introduction of floating point representation, which are sign bit, exponent and mantissa. The computation method of sign bit and exponent is based on the operation implemented and is straightforward. For reciprocal, the sign bit of the result is the same as that of the operand and exponent is the result of subtracting the operand exponent from zero. For division, the sign bit of the result is the output of XOR logic of the two sign bits from the operands and the exponent is obtained from the subtraction the exponent of the divisor from that of the dividend. For square root, the sign bit of the result should be zero. Otherwise, if the operand is negative, an exception is generated. Exponent for square root is the floor function of half of the
2.2. IMPLEMENTING ADVANCED FLOATING POINT COMPONENTS

operand’s exponent. Notice that if the operand’s exponent is odd, this will affect the calculation of the mantissa for square root.

The computation of the mantissa can differ based on the algorithm implemented, affecting latency, maximum clock frequency and resource utilization of the hardware implementation. There are three main methods used to implement floating point reciprocal, division and square root operations. They are digit recurrence (or subtractive), iterative (or multiplicative) and table-based method. Each method is discussed below.

2.2.1 Digit Recurrence Method

Generally speaking, the digit recurrence method for division and square root has two main steps. Step one is to select one number from the quotient digit selection function. Step two is to update the quotient remainder based on the number selected. After that step one and step two are repeated until the required precision is reached. One digit of the result will be generated after the two steps of one recurrence.

For floating point division using the digit recurrence method, one digit of the result is computed per clock cycle. This method is based on the long division commonly taught in grade school for performing division by hand. The most widely used digit recurrence method is known as the SRT algorithm named after the three people, Sweeney, Robertson, and Tocher, who invented it [15]. The textbook Digital Arithmetic [6] gives more details. Freiman [8] provides a good example of the SRT method based on non-restoring division; the radix of the example is 2. For binary
arithmetic, the approach used to obtain more than one bit per cycle employs a radix higher than 2.

Assume we use binary representation. Then the quotient digit selection set will be \{-1, 0, 1\} which is minimally and also maximally redundant. It is the best for quotient digit selection for binary non-restoring division. Figure 2.1 illustrates the operation of the SRT algorithm to compute \( x/d \).

In this figure, the subtraction module computes the residual recurrence function
2.2. IMPLEMENTING ADVANCED FLOATING POINT COMPONENTS

\[ w[j + 1] = r \cdot w[j] - d \cdot q[j + 1] \]

The quotient digit selection module performs the following steps to choose \( q_{j+1} \) from the set \(-1, 0, 1\). We have the following quotient digit selection function:

\[
q_{j+1} = \begin{cases} 
1 & 2 \cdot W_j >= 1/2; \\
0 & -1/2 =< 2 \cdot W_j < 1/2; \\
-1 & 2 \cdot W_j < -1/2; 
\end{cases}
\]

Send the \( q_{j+1} \) to the carry save adder. Also the result of the subtraction will act as a new \( w \) element to compute the next \( q \) element. After all the \( q \) values are generated based on the required precision of the quotient, or when the residual number is zero, the quotient will be at the output of the carry save adder.

The same steps apply to the computation of floating point square root [12]. The differences are the digit selection function and the method of computing the residual.

2.2.2 Iterative Method

Iterative methods are based on multiplications that give you intermediate results that converge to the number of bits of precision required. Two widely used iterative methods for floating point numbers are Newton Raphson and Goldschmidt [17, 18, 16, 9]. The difference between digit recurrence method and iterative method is that the former one only generates one bit per recurrence, while the latter one generates multiple bits per iteration and the output gets closer to the real result after each iteration.
2.2. IMPLEMENTING ADVANCED FLOATING POINT COMPONENTS

Figure 2.2: Iterative Method

For example, using Newton-Raphson method the reciprocal can be computed iteratively. Figure 2.2 shows the scheme of it. In order to find $1/d$, the iteration is seeded with $x_0$, an estimate to $1/d$; the iteration is: $x_{i+1} = x_i \ast (2 - d \ast x_i)$. The algorithm iterates until $S$ approximates the required precision. It takes only 3 iterations to get the proper result for a single precision operand if the output of the look-up table is 8 bits [18].

Goldschmidt’s algorithm also uses an iterative loop to calculate the quotient. The difference between this algorithm and Newton-Raphson algorithm is that for Goldschmidt’s algorithm the iteration multiplies both the dividend and the divisor. The iteration completes when the divisor is approximately equal to one and the required precision is reached. The value by which the dividend and divisor are multiplied in each iteration is two minus the current value of the divisor.

2.2.3 Table-based Method

Two table-based methods are discussed here. The first is previously used to implement variable precision division and square root in the VFLOAT library [13, 19]. The second is an improved algorithm whose table size grows more slowly as the bitwidth of the inputs grows. These methods use a stored value for the initial approximation to the result. They use Taylor series expansion to obtain the required number of bits
of precision.

The first approach, from [10], is the division previously used in VFLOAT library. It makes use of two multipliers and one look-up table to implement division. In the single precision floating point division implementation, the size of the first multiplier is $24 \times 24$ bits of input with 48 bits of output; the second multiplier has input $28 \times 28$ bits with an output of 56 bits. The look-up table has 12 bits for input, 28 for output, for a total size of approximately 16K bytes. However, the disadvantage of this algorithm is that the size of the look-up table increases exponentially with the bit width of the input. It is therefore impractical to implement a double precision division with this approach since the LUT would be prohibitively large. For this reason we have changed to a different division algorithm that scales better as the size of the division operation grows [7]. This algorithm requires a smaller look-up table and several small multipliers. The remainder of this thesis covers details of this algorithm and its implementation. The article [7] provides the algorithm for square root as well as division. A detailed explanation of these algorithms is given in the next chapter.

2.3 Related Work

The closest related work is FloPoCo [3]. It is an open-source generator of floating point arithmetic modules for FPGAs. The difference is that it is not a library of arithmetic operators, but a generator of VHDL modules written in C++. FloPoCo is
2.3. RELATED WORK

a command-line tool and can generate a single VHDL file containing the description of the operator, including fixed point and floating point operators. For floating point, the library includes addition, subtraction, multiplication, division, square root, square, etc. It can also support variable precision floating point numbers.

Comparing our library to FloPoCo, both of them can generate floating point operation modules described in VHDL, targeting both major FPGA vendors Altera and Xilinx. However, there are several differences between them. First, the floating point representation in FloPoCo is not strictly the format of IEEE 754 floating point representation. A 2 bit exception field is the first two bits in the representation, 00 for zero, 01 for normal numbers, 10 for infinities, and 11 for NaN. Second, operators generated by FloPoCo are correctly rounded to nearest. However operators generated by VFloat library have input port ROUND to select whether the rounding mode is truncation or round to nearest. Third, for floating-point square root FloPoCo uses classical digit-recurrence algorithm and polynomial approximation. VFloat library use table-based method to implement square root. Fourthly, an operator from FloPoCo can be either pipelined or combinational, while VFloat library components can not be totally combinational logic. However, their pipelining effort is tentative, meaning users might not get the frequency that they ask for. Finally, based on the experience of trying FloPoCo, the modules they generate are not in readable. Modifying the modules is difficult compared to modules from VFloat library.
Chapter 3

Algorithms Description

In this chapter, we present the algorithm we implemented for reciprocal, division and square root in the VFloat library. First section is a brief introduction of VFloat library from Reconfigurable Computing Lab in Northeastern University. Second and third sections are the description of the algorithm [7] implemented in reciprocal, division and square root components in the VFloat library. The rest of this chapter focuses on the algorithm used to compute the mantissa for reciprocal, division and square root. X and Y represent normalized inputs with the implied one explicitly represented.

3.1 VFloat Library

VFloat is a library of variable precision floating point units written in VHDL, targeting Altera and Xilinx FPGAs developed by the Reconfigurable Computing Laboratory of Northeastern University. Components include floating point arithmetic (addition, subtraction, multiplication, reciprocal, division, square root, accumulation) and format conversion (fix2float, float2fix). Operand can be variable precision
3.1. VFLOAT LIBRARY

Figure 3.1: Black Box of Components

Floating point, beyond the standard IEEE 754 formats. Any bitwidth exponent or mantissa is supported.

Figure 3.1 shows the standard input and output ports for components in the VFloat library. Each component has inputs READY, STALL, ROUND and EXCEPTION_IN and outputs DONE and EXCEPTION_OUT specifically to handle pipelining. The READY and DONE signals are used for determining when the inputs are ready and when the results are available for use. STALL allows a bubble to be inserted into the pipeline if needed. Round has two modes: round to zero or truncate, and round to nearest. The exception signals propagate an exception flag with the value that may be incorrect through the pipeline. For reciprocal and division, an exception is identified if the divisor is zero. For square root, an exception is identified if the operand is a negative number. Otherwise, the exception input is propagated to the exception output.
3.2 Reciprocal and Division

To find the reciprocal $1/Y$ or the quotient $X/Y$, the algorithm needs three steps: reduction, evaluation, and post-processing. Generally speaking, reduction step and evaluation step are the same for both reciprocal and division. The only difference is within the post-processing step.

In the reduction step after normalizing, the fractional part of the floating point number is $1 \leq Y < 2$. Assume $Y$ has an $m$ bit significand and $k$ is $\lceil (m + 2)/4 \rceil + 1$; $Y^{(k)}$ represents the truncation of $Y$ to $k$ bits. Define $R$ as the reciprocal of $Y^{(k)}$. Define $M = R$ for computing the reciprocal and division. For example, in double precision based on the equation above, $m=53$ and $k = 14$. So $R$ can be determined using a look-up table with a 14 bits address. The number of bits for the look-up table output for $R$ is 16.

In the evaluation step $B$ is defined as the Taylor series expansion of

$$f(A) = 1/(1 + A)$$

where $A$ is defined as $(Y \times R) - 1$. Also note that $-2^{-k} < A < 2^k$. For $z = 2^{-k}$, $A$ can be represented as:

$$A = A_2z^2 + A_3z^3 + A_4z^4 + ...$$
where $|A_i| \leq 2^k - 1$. We ignore the smaller terms that do not contribute to the final result.

Using the Taylor series expansion,

$$B = f(A) = C_0 + C_1A + C_2A^2 + C_3A^3 + C_4A^4 + \cdots$$

$$\approx C_0 + C_1(A_2z^2 + A_3z^3 + A_4z^4)$$

$$+ C_2(A_2z^2 + A_3z^3 + A_4z^4)$$

$$+ C_3(A_2z^2 + A_3z^3 + A_4z^4)^3$$

$$+ C_4(A_2z^2 + A_3z^3 + A_4z^4)^4$$

$$\approx C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6$$

Here, $C_i = 1$ when $i$ is even, $C_i = -1$ when $i$ is odd. Simplifying we get:

$$B = f(A) \approx 1 - A_2z^2 - A_3z^3 + (-A_4 + A_2^2)z^4 +$$

$$2A_2S_3z^5 - A_3^3z^6$$

$$\approx (1 - A) + A_2^2z^4 + 2A_2A_3z^5 - A_3^3z^6$$

The equation above is used in the implementation of reciprocal and division.

**In the post-processing step** final result is given by multiplication.

**For reciprocal** the result of reciprocal of $Y$ is given by the product of $M$ and $B$: $1/Y = M \times B$.

**For division** the result of division $X/Y$ is given by the product of $M$, $B$ and $X$: $X/Y = M \times B \times X$. 
3.3 Square Root

For computing square root, there are three steps similar to the reciprocal computation. First step is reduction, second is evaluation, and the last step is post-processing.

In the reduction step the difference between the reduction step of computing square root and reciprocal is that after getting R, M is assigned to \(1/\sqrt{R}\). So another look-up table to compute the inverse square root of R is needed. Notice that if the exponent is odd, there will be \(\sqrt{2}\) as a coefficient to multiply by the result at the last step. For the last step, we check the last bit of the exponent. 0 means the exponent is an even number, so we assign \(1/\sqrt{R}\) to M. If not, M equals to \(\sqrt{2}/\sqrt{R}\). To compute \(\sqrt{2}/\sqrt{R}\), we create another look-up table.

In the evaluation step \(f(A)\) is not \(1/(1 + A)\) as the above. Instead let

\[
f(A) = \sqrt{1 + A}
\]

\(f(A)\)'s Taylor series expansion is still the same as that of the reciprocal’s:

\[
B = f(A) = C_0 + C_1 A + C_2 A^2 + C_3 A^3 + C_4 A^4 + \cdots \\
\approx C_0 + C_1 (A_2 z^2 + A_3 z^3 + A_4 z^4) \\
+ C_2 (A_2 z^2 + A_3 z^3 + A_4 z^4) \\
+ C_3 (A_2 z^2 + A_3 z^3 + A_4 z^4)^3 \\
+ C_4 (A_2 z^2 + A_3 z^3 + A_4 z^4)^4 \\
\approx C_0 + C_1 A + C_2 A_2 z^4 + 2C_2 A_2 A_3 z^5 + C_3 A_2 z^6
\]

However, the coefficients change accordingly: \(C_0 = 1, C_1 = 1/2, C_2 = -1/8, C_3 = \)
3.3. SQUARE ROOT

1/16. Thus,

\[ B = f(A) = 1 + \frac{A}{2} - \frac{1}{8}A_2^2z^4 - \frac{1}{4}A_2A_3z^5 + \frac{1}{16}A_2^3z^6 \]

In the post-processing step the final result of the square root is given by the product of M and B: \( \sqrt{Y} = M \times B \).

In the next chapter, we will discuss implementations of these components in detail.
Chapter 4

Implementation

4.1 Implementation Overview

Variable precision reciprocal, division and square root are designed as part of the VFloat library discussed in chapter 3. Figure 4.1 shows the dataflow in the top level component of the library. The component consists of the module to denormalize the input floating point numbers, the core computation which includes reciprocal, division or square root and round to normal component for adjusting to the standard floating point format. There are two main modules in common among all components which are denormalize and round to normal. So for this chapter we will first discuss the common elements implementation and then the core components of reciprocal, division and square root separately.
4.2 Common Elements

4.2.1 Denormalizer

IEEE 754 standard floating point number consists of sign bit, exponent and mantissa. As described in chapter 2, the stored part of the mantissa does not include the first '1' as the integer. Since this '1' is required for computation, the denormalizer component adds it as the most significant bit of the fractional part. Based on the number of operands, we might need one denormalizer for reciprocal and square root or two for division.

4.2.2 Round to Normal

There are four rounding mode specified in the IEEE standard [2]: round to zero, round to nearest, round to negative infinity and round to positive infinity. In our
4.3. Reciprocal

Fig. 4.2: Reciprocal Components

library, there are two options for rounding. One is round to zero, which is truncation. The other is round to nearest. If input signal rounding is zero to a VFloat component, round to nearest is implemented; otherwise it’s round to zero. Round to normal component also removes the integer '1' of the mantissa used in the computation. After rounding and normalizing of this component, the floating point result for this component in IEEE standard format will be on the output port.

4.3 Reciprocal

Figure 4.2 shows the components and dataflow within the reciprocal. This corresponds to the algorithm described in section 3.2.

After obtaining the denormalized number, the sign bit of the result is the same as that of the denormalized number. For the exponent part, first we need to obtain the actual value of the exponent by subtracting the bias from the exponent. The second step is to subtract it from zero, then add the corresponding bias to the number obtained from the second step. This is how to get the exponent of the result.
4.4. DIVISION

Take double precision reciprocal operation as an example. The component uses the reciprocal table to calculate R and 4 multipliers. The look-up table has 14 bits of address and 16 output bits for a total of 32K bytes. The four multipliers are the following sizes. Multiplier YR has 17 bit and 58 bit inputs and a 71 bit output and its purpose is to multiply Y and R in order to get A. Multiplier S has two 14 bit inputs and a 28 bit output; its purpose is to compute \( A_2 \cdot A_2 \) and \( A_2 \cdot A_3 \). Multiplier M has 28 bit and 14 bit inputs and a 42 bit output; it computes the cube of \( A_2 \). Multiplier L has one 16 bit and one 58 bit input and a 74 bit output; it computes \( R \cdot B \). The pipeline of those multipliers can be adjusted by the parameter in the Core Generator (Xilinx) or MegaWizard (Altera). That’s a way to modify the number of clock cycles latency of the components. In addition, parameterized adders and logic gates from the VFLOAT library are also used.

4.4 Division

For floating point division, we need to denormalize the two input operands, compute the sign bit, compute the exponent and perform the division of the significands. The sign bit of the result is the XOR logic output of the signs of the two inputs. The exponent of the result is the difference between the exponents of the inputs with the bias suitably adjusted.

Figure 4.3 shows the the components and dataflow within the division. Division
4.5 SQUARE ROOT

The first step in computing square root is to check that the sign bit is positive. If not, the exception output is set high to indicate that an exception situation has occurred. For computing the exponent, first get the exponent value by subtracting the denormalized exponent by the bias. Second, if the intermediate number is even, then divide it by 2. Otherwise, subtract 1 and then divide by 2. Next, add the bias to the number obtained from the second step, and this is the temporary exponent part of the result.
If the exponent is an odd number, an extra factor of $\sqrt{2}$ will be multiplied to the fractional part of the mantissa result.

Figure 4.4 shows the components and dataflow of the square root. This corresponds to the algorithm described in section 3.3. The square root component uses reciprocal look-up table R to calculate $R$, look-up table M to get $1/\sqrt{M}$ when the exponent is even, the M_Mul2 look-up table to get $\sqrt{2}/\sqrt{M}$ when the exponent is odd and 4 multipliers. The multiplier is the IP Cores implemented using Core Generator from Xilinx IDE or Megacores using MegaWizard from Altera IDE. The number of pipeline stages of the multipliers can be adjusted by the parameter within Core Generator or MegaWizard.

For example, in double precision operation, the look-up table R has 14 address bits and 16 output bit for a total of 32K bytes. The look-up table M has 14 address bits and 55 output bits. For look-up table M_Mul2, the input is 14 bits and the output is 58 bits. For the four multipliers, multiplier YR has 17 bit and 58 bit inputs.
and a 71 bit output; its purpose is to multiply Y and R in order to get A. Multiplier S has two 14 bit inputs and a 28 bit output; Multiplier M has 28 bit and 14 bit inputs and a 42 bit output; Multiplier L has one 16 bit and one 58 bit input and a 74 bit output.

4.6 Conclusion

In this chapter, we have described the implementation of reciprocal, division and square root based on the table-based method described in the paper written by Ercegovac and Lang [7]. In the next chapter, we present results from implementing these components on Altera and Xilinx FPGAs.
Chapter 5

Experimental Results

Our implementation of the algorithms for variable precision floating point is written in VHDL and targets both of the most popular commercial FPGA vendors: Altera and Xilinx. For our experimental results, we built our reciprocal, division and square root components and then simulated and synthesized on the two main FPGA vendors’ platforms. We used a range of precisions supported by our variable precision floating point format. For Altera, we synthesized with the Altera IDE tool and targeted a Stratix V device. For Xilinx we used the Xilinx IDE and targeted a Virtex 6 Device. For tools we use Altera Quartus II 13.0 and the Xilinx ISE Design Suite 13.4. The designs make use of embedded multipliers and RAMs, which require using the intellectual property components provided with each set of tools. For Altera these are called Megacores; for Xilinx they are called IP Cores. The two manufacturers also use different simulators; Altera ships a version of ModelSim called ModelSim-Altera 10.1d while Xilinx has its own simulator, ISim, that is integrated with its tool suite. Both simulators were used to validate these designs. For the rest of the
### 5.1 Hardware Components on Virtex 6

Virtex 6 family [20] is a high performance programmable silicon family from Xilinx. The family has three sub-families, LXT, SXT and HXT. Each has different ratio of features to address the different needs. Virtex 6 FPGAs are built on 40-nm copper process technology, which is a programmable alternative to ASIC technology. Figure 5.1 shows the Virtex 6 LXT FPGA summary by device. XC6VLX75T FPGA is the one we chose for synthesis.

From the figure we know that in Virtex 6 FPGA family, there are logic cells, configurable logic blocks, DSP48E1 Slices, block RAM block, etc. A memory based

**Figure 5.1: Virtex 6 LXT FPGA Summary**

<table>
<thead>
<tr>
<th>Device</th>
<th>Logic Cells</th>
<th>Configurable Logic Blocks (CLBs)</th>
<th>DSP48E1 Slices</th>
<th>Block RAM Blocks</th>
<th>Interface Blocks for PCIe Express</th>
<th>Ethernet MAC(1)</th>
<th>Maximum Transceivers</th>
<th>Total IO Banks(2)</th>
<th>Max Speed (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC6VLX75T</td>
<td>74,496</td>
<td>11,540</td>
<td>1,045</td>
<td>298</td>
<td>312</td>
<td>516</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>XC6VLX130T</td>
<td>128,000</td>
<td>20,000</td>
<td>1,740</td>
<td>460</td>
<td>528</td>
<td>264</td>
<td>10</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>XC6VLX195T</td>
<td>199,600</td>
<td>31,200</td>
<td>3,040</td>
<td>640</td>
<td>698</td>
<td>344</td>
<td>10</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>XC6VLX240T</td>
<td>241,152</td>
<td>37,980</td>
<td>3,650</td>
<td>786</td>
<td>832</td>
<td>416</td>
<td>12</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>XC6VLX305T</td>
<td>364,032</td>
<td>56,880</td>
<td>4,130</td>
<td>856</td>
<td>832</td>
<td>416</td>
<td>12</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>XC6VLX550T</td>
<td>549,868</td>
<td>85,920</td>
<td>6,200</td>
<td>1,264</td>
<td>1,264</td>
<td>632</td>
<td>18</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>XC6VLX760</td>
<td>758,784</td>
<td>118,560</td>
<td>8,280</td>
<td>1,440</td>
<td>1,440</td>
<td>720</td>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>XC6VSX315T</td>
<td>314,860</td>
<td>49,200</td>
<td>5,050</td>
<td>1,344</td>
<td>1,400</td>
<td>704</td>
<td>12</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>XC6VSX475T</td>
<td>476,160</td>
<td>74,400</td>
<td>7,640</td>
<td>2,016</td>
<td>2,128</td>
<td>1,064</td>
<td>18</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

5.1. HARDWARE COMPONENTS ON VIRTEX 6
logic cell usually consists of a look-up table, a D type flip-flop and multiplexer. For one configurable logic block (CLB), there are two slices. One slice consists of four LUTs, eight flip flops, multiplexers and arithmetic carry logic. The LUT can be configured as one 6-input LUT (64 bit ROMs) with one output, or two 5 input LUTs (32 bit ROMs) with separate outputs but same addresses or logic inputs. From the data sheet, between 25% to 50% of total slices can use the LUTs as distributed 64-bit RAM or 32 bit shift register (SRL32) or two SRL16s. For DSP48E1 Slices, each slice contains one $25 \times 18$ multiplier, an adder and an accumulator. Each block RAM is 36 Kbits in size but also can be used as two independent 18 Kb blocks. Synthesis tools have many options to take advantage of the highly efficient logic, arithmetic and memory features.

5.2 Hardware Components on Stratix V

Stratix V Series FPGAs [1] are Altera’s 28-ns FPGAs which are optimized for high performance, high bandwidth applications. Figure 5.2 shows the architecture and features of Stratix V. The Stratix V device family contains GT, GX, GS and E sub-families.

The core logic fabric consists of high performance adaptive logic modules (ALMs). Each ALM has eight inputs with a fracturable LUT, two embedded adders and four dedicated registers. Variable-precision DSP blocks contain two 18 by 18 multipliers. The results can be summed forming a multiply accumulator. M20K memory blocks
5.3. RECIPROCAL

Table 5.1: Reciprocal Result and Comparison of Altera

<table>
<thead>
<tr>
<th>Method</th>
<th>Device</th>
<th>Latency</th>
<th>Max Freq.</th>
<th>Total Latency</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our 1st</td>
<td>Stratix V</td>
<td>9 CC</td>
<td>124 MHz</td>
<td>73 ns</td>
<td>502 ALMs, 706 Reg, 7 DSP block</td>
</tr>
<tr>
<td>Our 2nd</td>
<td>Stratix V</td>
<td>12 CC</td>
<td>191 MHz</td>
<td>63 ns</td>
<td>523 ALMs, 792 Reg, 7 DSP block</td>
</tr>
<tr>
<td>Our 3rd</td>
<td>Stratix V</td>
<td>17 CC</td>
<td>221 MHz</td>
<td>77 ns</td>
<td>627 ALMs, 1141 Reg, 7 DSP block</td>
</tr>
<tr>
<td>Megacore</td>
<td>Stratix V</td>
<td>27 CC</td>
<td>297 MHz</td>
<td>91 ns</td>
<td>817 ALMs, 2391 Reg, 18 DSP block</td>
</tr>
</tbody>
</table>

provide users good memory block performance and it can simplify floorplanning and routability. Optional hard error correction code (ECC) protection enables reliable delivery of the data. Figure 5.3 shows the GX device features. To obtain the following synthesis results, we use the device 5SGXB6 as the FPGA device.

5.3 Reciprocal

Figure 5.4 shows a simulation result for double precision reciprocal operation.
### 5.3. RECIPROCAL

#### Figure 5.3: Stratix V GX Device Features

<table>
<thead>
<tr>
<th>Features</th>
<th>55GXA3</th>
<th>55GXA4</th>
<th>55GXA5</th>
<th>55GXA7</th>
<th>55GXA9</th>
<th>55GX8B</th>
<th>55GX8B</th>
<th>55GX8B</th>
<th>55GX8B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Elements (K)</td>
<td>340</td>
<td>420</td>
<td>490</td>
<td>622</td>
<td>840</td>
<td>952</td>
<td>490</td>
<td>597</td>
<td>840</td>
</tr>
<tr>
<td>Registers (K)</td>
<td>513</td>
<td>634</td>
<td>740</td>
<td>939</td>
<td>1,268</td>
<td>1,437</td>
<td>740</td>
<td>902</td>
<td>1,268</td>
</tr>
<tr>
<td>14.1-Gbps Transceivers</td>
<td>12, 24, or 36</td>
<td>24 or 36, or 48</td>
<td>24, 36, or 48</td>
<td>36 or 48</td>
<td>36 or 48</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>PCIe hard IP Blocks</td>
<td>1 or 2</td>
<td>1 or 2</td>
<td>1, 2, or 4</td>
<td>1, 2, or 4</td>
<td>1, 2, or 4</td>
<td>1 or 4</td>
<td>1 or 4</td>
<td>1 or 4</td>
<td>1 or 4</td>
</tr>
<tr>
<td>Fractional PLLs</td>
<td>20&lt;sup&gt;5&lt;/sup&gt;</td>
<td>24</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>24</td>
<td>24</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>M20K Memory Blocks</td>
<td>957</td>
<td>1,900</td>
<td>2,304</td>
<td>2,560</td>
<td>2,640</td>
<td>2,640</td>
<td>2,640</td>
<td>2,640</td>
<td>2,640</td>
</tr>
<tr>
<td>M20K Memory (Mbits)</td>
<td>19</td>
<td>37</td>
<td>45</td>
<td>50</td>
<td>52</td>
<td>52</td>
<td>41</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>Variable Precision Multipliers (18x18)</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>704</td>
<td>704</td>
<td>798</td>
<td>798</td>
<td>704</td>
<td>704</td>
</tr>
<tr>
<td>Variable Precision Multipliers (27x27)</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>352</td>
<td>352</td>
<td>399</td>
<td>399</td>
<td>352</td>
<td>352</td>
</tr>
</tbody>
</table>

#### Figure 5.4: Simulation Result of Double Precision Reciprocal
5.3. **RECIPROCAL**

For reciprocal operation, Xilinx does not have a specific reciprocal IP Core. Instead reciprocal must be implemented using the floating point division IP Core. Altera has a MegaCore called ALT FpINA which is the MegaCore for reciprocal operation. We use the Altera MegaWizard to generate reciprocal Megacore and synthesize the component to obtain clock cycle latency, maximum clock frequency of operation and resource utilization from the report. Table 5.1 compares latency, maximum clock frequency and resource utilization of our component with those of the reciprocal MegaCore. CC is the number of clock cycles in the table.

We know from Chapter 4 that we have four Megacores which should be generated by the MegaWizard built in Quartus II. They are Multiplier YR, Multiplier S, Multiplier M, Multiplier L. For each Megacore, there is a parameter to select for its number of pipeline stages, which we can choose to modify the whole operation’s latency, thus changing the maximum clock frequency and resource utilization.

In our 1st method, the pipeline stage of Multiplier YR, S, M, L are all 1. In our 2nd method, the pipeline stage of Multiplier YR, S, M, L are 2, 1, 1 and 3 respectively. In our 3rd method, the pipeline stage of Multiplier YR, S, M, L are 2, 2, 4 and 3 respectively.

The Megacore from Altera has fixed latency, which is 27 clock cycles. It can not be modified within MegaWizard and that is a big disadvantage of their approach. From the results we can see that our component is more flexible and more customizable than Altera’s. Also our latency can get as low as 9 clock cycles with comparably
5.4 Division

Figure 5.5 shows a simulation result for double precision division operation.

We implemented our division and compare it to other popular designs. Results are shown in Table 5.2 for Xilinx and Table 5.3 for Altera. The clock latency of our implementation is 14 clock cycles. The maximum clock frequency for our design on Stratix V from Altera is 145 MHz. Implemented on a Xilinx Virtex 6, the maximum clock frequency is 148 MHz.

We compared our design to several popular methods for implementing division,
Table 5.3: Division Result and Comparison of Altera

<table>
<thead>
<tr>
<th>Method</th>
<th>Device</th>
<th>Latency</th>
<th>Max Freq.</th>
<th>Total Latency</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our 1st</td>
<td>Stratix V</td>
<td>14 CC</td>
<td>121 MHz</td>
<td>116 ns</td>
<td>818 ALMs, 931 Logic Reg, 11 DSP block</td>
</tr>
<tr>
<td>Our 2nd</td>
<td>Stratix V</td>
<td>16 CC</td>
<td>145 MHz</td>
<td>110 ns</td>
<td>1004 ALMs, 1105 Logic Reg, 13 DSP block</td>
</tr>
<tr>
<td>MegaCore</td>
<td>Stratix V</td>
<td>10 CC</td>
<td>176 MHz</td>
<td>57 ns</td>
<td>525 ALMs, 1247 Logic Reg, 14 DSP block</td>
</tr>
<tr>
<td>MegaCore</td>
<td>Stratix V</td>
<td>24 CC</td>
<td>237 MHz</td>
<td>101 ns</td>
<td>849 ALMs, 1809 Logic Reg, 14 DSP block</td>
</tr>
<tr>
<td>MegaCore</td>
<td>Stratix V</td>
<td>61 CC</td>
<td>332 MHz</td>
<td>184 ns</td>
<td>9379 ALMs, 13493 Logic Reg</td>
</tr>
<tr>
<td>Digital Recur</td>
<td>Stratix V</td>
<td>36 CC</td>
<td>219 MHz</td>
<td>164 ns</td>
<td>2605 ALMs, 5473 Logic Reg</td>
</tr>
<tr>
<td>Newton Raphson</td>
<td>Stratix V</td>
<td>18 CC</td>
<td>268 MHz</td>
<td>67 ns</td>
<td>444 ALMs, 823 Logic Reg, 2 M20K, 9 DSP</td>
</tr>
</tbody>
</table>

including the floating point cores provided from each manufacturer. Table 5.2 shows the synthesis results of several methods using the Xilinx IDE. Xilinx allows the designer to choose the pipeline stage of the division as a parameter in Core Generator. We chose to compare to latencies of 8, 14 and 20 clock cycles. Note that given the same clock latency, our design provides a better maximum clock frequency. We use more LUTs, but fewer slice registers than a comparable Xilinx IP Core. The last design is reported from a paper that presents a multiplicative division [11]. This design can only support double precision with an error of 2 units in the last place (ulps) while our error is one ulp. Also, this design has long latency although a fast maximum clock frequency. These numbers are reported by the authors; we did not implement their design.

Table 5.3 shows the synthesis results of several methods using the Altera IDE. The Megacore from Altera has only 3 fixed latencies that you can choose, 10, 24 and
61 clock cycles. From the table, it is clear that the number of clock latency of our first double precision division is the second smallest in the table. The design with lower latency uses the floating point MegaCore provided by Altera. Note that this design uses more DSP blocks than our design. We can change the latency as our second implementation shows by adjusting the pipeline parameter of MegaCores. Thus, our module is more flexible than the Altera Megacore division. The last two designs repeat results previously published. The Radix 4 digit recurrence implemented by FloPoCo [4] has larger maximum clock frequency but also longer latency [5]. A similar situation applies to the method with Polynomial Approximation (d=2) plus Newton Raphson algorithm [14].

## 5.5 Square Root

Figure 5.6 shows one simulation result of double precision square root operation.

Table 5.4 is the double precision square root synthesis result for Altera Stratix V.
5.5.  \textit{SQUARE ROOT}

<table>
<thead>
<tr>
<th>Method</th>
<th>Device</th>
<th>Latency</th>
<th>Max Freq.</th>
<th>Total Latency</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>Stratix V</td>
<td>11 CC</td>
<td>120 MHz</td>
<td>92 ns</td>
<td>856 ALMs, 1001 Logic Reg, 12 DSP block</td>
</tr>
<tr>
<td>MegaCore</td>
<td>Stratix V</td>
<td>30 CC</td>
<td>206 MHz</td>
<td>146 ns</td>
<td>1030 ALMs, 1986 Logic Reg</td>
</tr>
<tr>
<td>MegaCore</td>
<td>Stratix V</td>
<td>57 CC</td>
<td>342 MHz</td>
<td>167 ns</td>
<td>1493 ALMs, 3570 Logic Reg</td>
</tr>
</tbody>
</table>

Table 5.4: Square Root Result and Comparison of Altera

<table>
<thead>
<tr>
<th>Method</th>
<th>Device</th>
<th>Latency</th>
<th>Max Freq.</th>
<th>Total Latency</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our 1st</td>
<td>Virtex 6</td>
<td>8 CC</td>
<td>107 MHz</td>
<td>75 ns</td>
<td>989 Reg, 6117 Slice LUTs</td>
</tr>
<tr>
<td>Our 2nd</td>
<td>Virtex 6</td>
<td>11 CC</td>
<td>179 MHz</td>
<td>61 ns</td>
<td>3363 Reg, 6252 Slice LUTs</td>
</tr>
<tr>
<td>IP Core</td>
<td>Virtex 6</td>
<td>11 CC</td>
<td>98 MHz</td>
<td>112 ns</td>
<td>657 Reg, 1940 Slice LUTs</td>
</tr>
<tr>
<td>IP Core</td>
<td>Virtex 6</td>
<td>14 CC</td>
<td>121 MHz</td>
<td>116 ns</td>
<td>791 Reg, 1961 Slice LUTs</td>
</tr>
</tbody>
</table>

Table 5.5: Square Root Result and Comparison of Xilinx

There are only two latencies to choose from inside the MegaWizard for the square root Megacore. One is 30 clock cycles and the other is 57 clock cycles. Our square root operation has much smaller latency than Altera’s. We use 12 DSP blocks but fewer ALMs and logic registers on the FPGA.

Table 5.5 is the double precision square root synthesis results for Xilinx Virtex 6. In our 1st method, the number of pipeline stages of Multipliers YR, S, M, L are all 1. In our 2nd method, the number of pipeline stage of Multipliers YR, S, M, L are 2, 1, 1, 3 respectively. For the Xilinx IP Core, the latency can be adjusted as a parameter while generating it, which is a big advantage over Altera’s Megacore. In order to compare with the IP Core, we assign the double precision square root IP core’s latency as 11 clock cycles which is the same as our second method. Synthesis shows that the maximum clock frequency of our application is 179MHz, while Xilinx IP Core is only 98 MHz, which means our component is much faster than their’s.
5.5. SQUARE ROOT

In summary, our components are flexible and provide a good tradeoff between clock frequency, latency and resource utilization. They can be easily be incorporated into a larger floating point pipeline. The clock cycle latency can be adjusted by choosing parameters within Core Generator (Xilinx) or MegaWizard (Altera) for multiplier cores. In addition, the components in the VFloat library can be implemented on either major vendor’s FPGAs.
Chapter 6

Conclusions and Future Work

We have presented variable precision floating point reciprocal, division and square root implementations that produce hardware components with a good tradeoff of maximum clock frequency, number of clock cycle latency and resource utilization. These implementations are cross-platform, meaning they can be implemented on both Altera and Xilinx FPGAs. Also our designs make use of the embedded multiplier Cores and embedded RAM Cores commonly found in modern FPGA fabric.

In the future, we plan to improve our library even further. In particular, we will focus on improving the division frequency by focusing on optimizing the critical path and trying different levels of pipelining. In addition, we plan to implement variable precision reciprocal square root as a new component of our VFLOAT library [13]. Finally, this library will be applied to applications including cognitive radio, medical electronics, etc.
Bibliography


