Architectural Design Space Exploration of Reciprocal and Square Root for Arbitrary Precision Fixed-point and Floating-point Arithmetic

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Electrical Engineering

by
Fang Lin
Fall 2015
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the
Thesis of Fang Lin:

Architectural Design Space Exploration of Reciprocal and Square Root for Arbitrary
Precision Fixed-point and Floating-point Arithmetic

Amirhossein Alimohammad, Chair
Department of Electrical and Computer Engineering

Ke Huang
Department of Electrical and Computer Engineering

Li An
Department of Geography

oct-19-2015
Approval Date
DEDICATION

To
My family
ABSTRACT OF THE THESIS

Architectural Design Space Exploration of Reciprocal and Square Root for Arbitrary Precision Fixed-point and Floating-point Arithmetic

by

Fang Lin

Master of Science in Electrical Engineering
San Diego State University, 2015

Efficient implementations of elementary operations, such as reciprocal, square root, and reciprocal square root, can result in significantly more compact and higher throughput designs. Designers have to carefully choose the best suited algorithms among numerous available techniques for custom implementations of these elementary functions that meet their design requirements. This thesis first explores the design trade-offs for alternative high-performance implementations of these three operations in single-precision representation. Various trade-offs at the algorithmic and architectural levels are considered, such as the convergence rate of the algorithm, numerical stability and accuracy, inherent parallelism, resource requirements, latency, as well as the ability to share hardware datapath among the implementations of the required functions. The comparative study and quantitative analysis of high-performance architectures should help designers to select an efficient implementation of the most suitable algorithm for their custom designs. Another goal of the thesis is to implement reciprocal, square root, and reciprocal square root operations in fixed-point and floating-point formats with arbitrary precisions. These elementary functions, developed in MATLAB, are then modeled in Verilog hardware description language and implemented on a field-programmable gate array (FPGA). The implementation results should provide valuable insight for choosing suitable architectures for efficient realization of these elementary functions. Finally, the optimized designs are compared with other state-of-the-art implementations of these modules.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ABSTRACT</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>xii</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>1  INTRODUCTION TO COMPARATIVE ANALYSIS AND HARDWARE IMPLEMENTATION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivations</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Contribution of the Thesis</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Design and Verification Flow</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Organization of the Thesis</td>
<td>4</td>
</tr>
<tr>
<td>2  MULTIPLICATIVE ITERATIVE, NON-ITERATIVE, AND HYBRID METHODS</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Fixed-point Format</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Floating-point Format</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Iterative Techniques</td>
<td>6</td>
</tr>
<tr>
<td>2.3.1 Newton-Raphson (NR) Algorithm</td>
<td>6</td>
</tr>
<tr>
<td>2.3.2 Goldschmidt (GS) Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Non-iterative Techniques</td>
<td>14</td>
</tr>
<tr>
<td>2.4.1 Ito’s Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>2.4.2 Hung’s Algorithm</td>
<td>16</td>
</tr>
<tr>
<td>2.4.3 Jeong’s Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>2.5 Hybrid Algorithm</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Comparative Analysis</td>
<td>23</td>
</tr>
<tr>
<td>2.7 Summary</td>
<td>25</td>
</tr>
<tr>
<td>3  ARBITRARY PRECISION SCHEME FOR FIXED-POINT RECIPROCAL, SQUARE ROOT,</td>
<td>26</td>
</tr>
<tr>
<td>AND RECIPROCAL SQUARE ROOT</td>
<td></td>
</tr>
<tr>
<td>3.1 Fixed-point Design and Implementation</td>
<td>26</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Characteristics of the optimized NR reciprocal implementations</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Characteristics of the optimized NR reciprocal square root implementations</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Characteristics of the optimized GS reciprocal implementations</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Characteristics of the optimized GS reciprocal square root implementations</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Characteristics of the optimized NR reciprocal datapaths utilizing Ito’s initial approximations</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Characteristics of reciprocal and square root implementations using the optimized Hung <em>et al.</em> technique</td>
<td>18</td>
</tr>
<tr>
<td>2.7</td>
<td>Characteristics of implementing reciprocal using the optimized Jeong <em>et al.</em> technique</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>Characteristics of the optimized datapaths for both reciprocal and reciprocal square root</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Implementation characteristics of the fixed-point reciprocal with arbitrary precision</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Implementation characteristics of the fixed-point square root operation with arbitrary precision</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Implementation characteristics of the fixed-point reciprocal square root with arbitrary precision</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Various floating-point formats</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementation characteristics of reciprocal for floating-point inputs format with arbitrary precisions</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Implementation characteristics of square root for floating-point inputs with arbitrary precisions</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Implementation characteristics of reciprocal square root for floating-point inputs with arbitrary precisions</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>Implementation characteristics of division in floating-point format with arbitrary-precision</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation characteristics of the division in fixed-point format with arbitrary-precision</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>Implementation characteristics of different single-precision reciprocal realizations</td>
<td>60</td>
</tr>
</tbody>
</table>
Table 6.2. Implementation characteristics of different double-precision reciprocal realizations. ................................................................. 60

Table 6.3. Implementation characteristics of different single-precision square root realizations. .................................................................................. 60

Table 6.4. Implementation characteristics of different double-precision square root realizations. ................................................................. 61

Table 6.5. Implementation characteristics of different single-precision division realizations. .................................................................................. 61

Table 6.6. Implementation characteristics of different double-precision division realizations. .................................................................................. 62
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Fixed-point format</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Floating-point format</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>The datapath for NR reciprocal</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>The reciprocal error plot utilizing NR approach for single-precision accuracy</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>The datapath for NR reciprocal square root</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>The reciprocal square root error plot utilizing NR method single-precision accuracy</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>The datapath for GS reciprocal</td>
<td>12</td>
</tr>
<tr>
<td>2.8</td>
<td>The reciprocal error using GS approach for single-precision accuracy</td>
<td>13</td>
</tr>
<tr>
<td>2.9</td>
<td>The datapath for GS reciprocal square root</td>
<td>14</td>
</tr>
<tr>
<td>2.10</td>
<td>The reciprocal square root error plot utilizing GS approach for single-precision accuracy</td>
<td>15</td>
</tr>
<tr>
<td>2.11</td>
<td>The reciprocal error plot utilizing Ito’s approach for single-precision accuracy</td>
<td>17</td>
</tr>
<tr>
<td>2.12</td>
<td>The reciprocal square root error plot utilizing Ito’s approach for single-precision accuracy</td>
<td>17</td>
</tr>
<tr>
<td>2.13</td>
<td>The datapath of Hung et al. technique for (a) reciprocal and (b) square root</td>
<td>18</td>
</tr>
<tr>
<td>2.14</td>
<td>The reciprocal error plot utilizing Hung’s approach for single-precision accuracy</td>
<td>18</td>
</tr>
<tr>
<td>2.15</td>
<td>The reciprocal square root error plot utilizing Hung’s approach for single-precision accuracy</td>
<td>19</td>
</tr>
<tr>
<td>2.16</td>
<td>The datapath of Jeong et al. scheme</td>
<td>19</td>
</tr>
<tr>
<td>2.17</td>
<td>The error measurement for Jeong reciprocal</td>
<td>20</td>
</tr>
<tr>
<td>2.18</td>
<td>Plots of $\frac{1}{x}$, $\sqrt{x}$ and $1/\sqrt{x}$ over $x \in [1, 2)$</td>
<td>21</td>
</tr>
<tr>
<td>2.19</td>
<td>Block diagram of the unified datapath (OURS+REC+ISQT)</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Fixed-point calculation of $f(x)$ over $x$ with an arbitrary precision</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Bitwidth selection for the reciprocal operation</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Bitwidth selection for the reciprocal square root operation</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>Bitwidth selection for the square root operation</td>
<td>35</td>
</tr>
</tbody>
</table>
Figure 3.5. Fixed-point arbitrary-precision module ........................................... 36
Figure 3.6. Validation process ........................................................................ 39
Figure 4.1. Implementation of the three elementary functions in floating-point format supporting arbitrary precisions. .................................................. 41
Figure 4.2. Exponent result of reciprocal ...................................................... 42
Figure 4.3. Exponent result of square root ................................................... 42
Figure 4.4. Exponent result of reciprocal square root ........................................ 43
Figure 4.5. Floating-point module with arbitrary precision ............................. 46
Figure 5.1. Floating-point multiplier ............................................................. 49
Figure 5.2. Arbitrary-precision floating-point block diagram for division ............ 50
Figure 5.3. Arbitrary-precision fixed-point block diagram for division ............... 51
Figure 5.4. Arbitrary-precision fixed-point module for division ....................... 51
Figure 5.5. Arbitrary-precision fixed-point module for division ....................... 52
Figure 6.1. The critical path of the NR reciprocal datapath ............................... 53
Figure 6.2. The datapath for pipelined NR reciprocal (direct implementation) .... 54
Figure 6.3. The datapath for the pipelined arbitrary-precision floating-point mutiplier 54
Figure 6.4. The datapath for the pipelined normalized square root ................... 55
Figure 6.5. The datapath for the pipelined arbitrary-precision floating-point scheme .. 56
Figure 6.6. The reciprocal error plot utilizing Ito’s approach for double-precision accuracy ................................................................. 57
Figure 6.7. The datapath for NR reciprocal utilizing Ito’s initial approximation approach. 57
Figure 6.8. The datapath for unfolded NR reciprocal utilizing Ito’s initial approximation approach .................................................. 58
Figure 6.9. The reciprocal error plot utilizing Ito’s approach for double-precision accuracy with two iterations ........................................ 59
ACKNOWLEDGMENTS

Writing this thesis has been a long journery. Many people helped me in different phases of this research, and I am certain that without their support, guidance and fortification, this research would have never achieved this success. I am grateful to God for the faith, energy, and hope to complete this thesis.

First and foremost I would like to express my deepest gratitude to my advisor Dr. Amir Alimohammad for his excellent guidance, caring, and patience. He provided abundant resources to start this research and continuous sustenance through the journey. His precise insight to the Computer Arithmetic and great expertise in the VLSI design always led me to the right direction when I encountered major obstacles. His high standard enabled me to accomplish my research with a high quality. Also, I really respect and appreciate his prudent scientific attitude, which inspired me to become a better researcher.

My sincere thanks also go to Prof. Ke Huang and Prof. Li An for sharing expertise, valuable guidance and insightful comments. It is a great honor to have them on my thesis committee.

I thank my fellow labmates in VLSI Design and Test Laboratory: Poornima Sumanth, Nikhilesh Bhagat, Ganesh Jaykumar and Abinaya Ashwin for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for the food you brought to me when I was starving at the midnights. Also I appreciate my mentor in Apple: Rahiem Burrell who trained me to finish every task excellent and be optimistic to the tough situations all the way.

Last but not the least, I am very thankful to my parents: Lei Lin and Yonghui Yu, for giving birth to me and supporting me selflessly throughout my life.
CHAPTER 1
INTRODUCTION TO COMPARATIVE
ANALYSIS AND HARDWARE
IMPLEMENTATION

1.1 INTRODUCTION

Basic arithmetic operations, such as reciprocal, square root, and reciprocal square root, play very important roles in digital signal processing (DSP) and modern microprocessors. Choosing the most suitable algorithm among the many alternatives for the efficient calculation of these elementary functions is an important design decision. This decision requires careful consideration of the design metrics, such as resource requirements, computational complexity, numerical accuracy, throughput and latency.

This thesis focuses on the design, comparative analysis, and implementation of the three basic arithmetic operations: reciprocal, square root, and reciprocal square root. This thesis demonstrates how these three operations can be modeled, designed, and implemented in hardware to meet various resource constraints and performance requirements. More specifically, the goals of this thesis are as follows:

• Comparatively analyze the multiplicative methods for the design and implementation of the three transcendental operations: reciprocal, square root, and reciprocal square root;
• Design and hardware implementation of these three arithmetic operations in fixed-point and floating-point representations with arbitrary precisions;
• Design and hardware implementation of arbitrary-precision divider in fixed-point and floating-point formats;
• Exploration and optimization of the designs for performance and area;
• Comparative analysis of our implementation results with Xilinx and Altera dedicated cores and state-of-the-art published works.

1.2 MOTIVATIONS

Conceptually, the simplest and fastest way to approximate the reciprocal, square root and reciprocal square root of a normalized \( x \in [1, 2) \) is to use a direct table look-up, where the fraction \( d = x - 1 \) is the applied address and the look-up table (LUT) contents at address \( d \) store the value \( f(x) \) of the function at \( x \). While table look-up is fast and requires only a small LUT on a significand with short wordlength \( n \), for single-precision representation (i.e., 24-bit \( x \)) the size of the required memory becomes prohibitively large. Several techniques have thus
been proposed to reduce the size of the LUT, such as partial product arrays [1], the add-table lookup-add scheme [2], and the bipartite table method [3], which use one or at most a few tables and adders. These table-bound methods are inefficient for single-precision computations due to the rapid growth in the size of the LUTs with the accuracy of the result. For example, the bipartite table method in [3] uses two relatively large tables of sizes of about $2^{16} \times 24$ bits and $2^{16} \times 8$ and a subtractor, but no multipliers, to achieve single-precision accuracy.

Various techniques have been proposed to reduce the required size of the memories. These algorithms fall into two main categories: iterative methods [4] and non-iterative techniques [5]. Iterative methods can in turn be sub-classified into the multiplicative algorithms and the digit-recurrence approaches. The multiplicative algorithms use a series of multiplications, subtractions, bit inversions, and shift operations to iteratively refine an initial guess. The multiplicative algorithms typically converge at a quadratic rate, which means that the number of accurate bits in the approximation roughly doubles at every iteration. In contrast, the subtractive digit-recurrence approaches compute the elementary functions directly and have a linear convergence rate, with one bit of increased accuracy produced at every iteration.

Reconfigurable arithmetic logic units (ALUs) require datapaths that support various precisions for different applications. The work in [6] presents a floating-point division design in double-precision format with a maximum error of two units in the last place (ulp) and without using rounding. In [7], the main building blocks for the variable long-precision (VLP) arithmetic are discussed and the goal is to allow reduced reconfiguration time between arithmetic operations. In [8] a low-latency division algorithm PolyGSopt was introduced based on the Goldschmidt algorithm, which are accurate up to one bit of the precision in double-precision floating-point format. A general architecture was implemented for division and square root based on the Goldschmidt’s and the Newton-Raphson algorithms in [9]. Quantitative analysis of the floating-point arithmetic on field-programmable gate arrays (FPGAs) was discussed in [10] providing stages selection analysis in a pipelined datapath. The work shows an efficient algorithm for the variable-precision division, which is based on the traditional convergence algorithm, but it would work well only on the normalized inputs with even number of fractional bitwidth. A floating-point library, which can be used to achieve more parallelism and less power dissipation than adhering to a standard format, is presented in [11] and [12]. In [13], a LUT-based double-precision floating-point divider is presented. The FPGA implementations for the reciprocal, division, and square root operations are described in [14]. It uses the traditional Taylor’s and convergence methods and the arbitrary-precision operations in fixed-point format were not considered.
1.3 Contribution of the Thesis

This thesis first explores the algorithmic trade-offs and architectural datapaths for the high-performance implementation of the three elementary functions, reciprocal, square root, and reciprocal square root. We are interested in approximating \( f(x) \) within the input domain \( 1 \leq x < 2 \) to single-precision accuracy, and then extend our approach to support arbitrary precision accuracy. This means that both the most significant bit (MSB) and the approximated 23-bit fraction of the results are exactly the same as the MSB and 23 bits to the right of the radix point in the single-precision representation. Multiplicative implementations can provide lower latency and lower cost than the subtractive digit recurrence method. Due to the faster convergence of the iterative multiplicative techniques these approaches are better suited for high-performance computations than the digit-recurrence approach. Hybrid methods achieve both a reduction in memory size compared with the table-bound schemes and significant speed-ups compared with the compute-bound methods. Moreover, the significantly increased number of dedicated multipliers in the state-of-the-art FPGAs makes these methods even more attractive for high-performance implementations.

Choosing the most appropriate algorithms according to different needs depends on the available on-chip memory, logical resources, dedicated multipliers, and latency. After comparing potential algorithms, we choose certain algorithms as the basic building blocks and use them to explore the arbitrary-precision arithmetic in both fixed-point and floating-point formats. For the fixed-point format, the arbitrary variable-precision arithmetic contains four blocks: unsign-sign, normalization, normalized calculation, scaling and denormalization. The most challenging part is to cooperate scaling and normalizing the intermediate and final results along with finding optimal precisions for the datapaths. The datapaths for both fixed-point and floating-point formats with arbitrary precisions are presented in detail along with the exhaustive verification of bit-true fixed-point and floating-point results and their FPGA implementations.

Then, we designed arbitrary-precision schemes for the division operation using multiplicative approaches in fixed-point and floating-point formats. The implementations of the two schemes are presented with different bitwidths selection to provide a more comprehensive comparison. Various optimization techniques for performance enhancement and hardware resource minimization are applied to our designs. The comparative analysis of our designs and other published work is presented. All our designs are written in Verilog HDL and synthesized by Xilinx ISE 14.7 on a Virtex-7 XC7V2000T and a Virtex-6 XC6VCX75T FPGA at speed grades -1 and -2.
1.4 Design and Verification Flow

To design, implement, and verify the basic arithmetic operations, we follow the following steps:

- Modeling various potential multiplicative techniques in single-precision format in MATLAB using a custom library of fixed-point and floating-point modules supporting arbitrary precisions.

- Verifying various multiplicative methods in single-precision format through exhaustive bit-true simulation in MATLAB. The simulator uses a comprehensive library of bit-true fixed-point and floating-point arithmetic and logical operations implemented in Mex-C. These compiled libraries include parameterizable modules, with adjustable bit-widths, that provide a flexible simulation environment for the bit-true comparison of approximated values of \( f(x) \) with accurate function values in double-precision floating-point format.

- Implementing the multiplicative methods in single-precision format on an FPGA, and compare the FPGA outputs with the MATLAB simulation results for functional verification.

- Comparatively analyzing the hardware implementation results, such as resource utilization, performance, and latency.

- Design and implementation of alternative multiplicative methods for reciprocal, square root, and reciprocal square root in fixed-point and floating-point formats supporting arbitrary precisions.

- Design and implementation of the arbitrary-precision division using multiplicative approach in fixed-point and floating-point formats.

- Optimization of our designs for performance and hardware resources utilization.

- Comparing our optimized implementation results with other state-of-the-art academic and industrial published works.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 explores the architectural and algorithmic trade-offs of different multiplicative iterative and non-iterative techniques. In Chapter 3, Chapter 4, and Chapter 5, the implementations of the three arithmetic operations and the division operation in fixed-point and floating-point representations with supporting arbitrary-precision are presented. Chapter 6 presents the optimization technique and comparisons among various implementations. Chapter 7 makes some concluding remarks.
CHAPTER 2
MULTIPLICATIVE ITERATIVE,
NON-ITERATIVE, AND HYBRID METHODS

Traditionally, the three basic arithmetic operations, reciprocal, square root, and reciprocal square root are implemented either using multiplicative method or the digit-recurrence method. This thesis focuses on the multiplicative methods for calculating these three operations by iteratively improving an initial approximation.

2.1 FIXED-POINT FORMAT

Modern digital signal processors (DSPs) and microprocessors may involve high-precision digitized representation of real values. No finite number system can represent any real value. A sequence of \( n \) bits in a register can represent an integer and/or a fractional number. Any real value could be separated into an integer part and a fractional part. The delimitation of the integer part and the fractional part is called the radix point. There are two systems that can be used to represent a subset of real numbers: fixed-point format and floating-point format. In hardware implementation of the fixed-point format, the radix point is implied in a pre-determined fixed position, as shown in Fig. 2.1. The wordlength and the position of the radix point are two main attributes of the fixed-point format. The \( n \) bits are separated into two parts: \( i \) bits for the integer part and \( f \) bits are used for the fractional part.

\[
\begin{array}{c|c|c}
\text{An } n \text{-bit sequence} & X_{i-1}X_{i-2}\ldots X_0 & . & X_{-2}X_{-3}\ldots X_{-1} \\
\text{integer part} & \text{binary point} & \text{fractional part}
\end{array}
\]

Figure 2.1. Fixed-point format.

2.2 FLOATING-POINT FORMAT

Floating-point arithmetic is commonly used in algorithms where the values vary over a relatively large dynamic range. The difference between floating-point and fixed-point arithmetic and is that the later one has a radix point with a predetermined location to divide the fractional part and the integer part. On the other hand, the floating-point representation uses three fields to represent real values: the sign, exponent, and fraction.
Normalized IEEE floating-point numbers have the form \((-1)^s x 2^{E-B}\) where \(s \in \{0, 1\}\) is the sign, the significand \(x = 1.d_1d_2 \ldots d_{k+1} \ldots d_{W_f}\) is an \(W_f + 1\)-bit normalized unsigned binary number within \([1, 2)\), and \(d = d_1d_2 \ldots d_k d_{k+1} \ldots d_{W_f}\) represents the portion of the significand to the right of the binary point, where \(d_i \in \{0, 1\}\). The unsigned exponent field \(E\) is biased, meaning that the stored value \(E\) is biased by \(B\), where \(B = 2^{W_e-1} - 1\). Fig. 2.2 shows the floating-point format. In the single-precision floating-point format, the significand precision is 24 bits where the leading one is implicit and only the fractional \(W_f = 23\) bits are stored, the exponent \(E\) is 8 bits, \(B = 127\), and the value of the exponent \(Val(E) \in [-2^7, 2^7 - 1]\). In the double-precision floating-point format, the significand precision is 53 bits where the leading one is implicit, the exponent is 11 bits, \(B = 1023\), and the value of the exponent \(Val(E) \in [-2^{10}, 2^{10} - 1]\). Let \(f(x)\) denote one of the elementary functions \(x^{-1}, x^{0.5}\) and \(x^{-0.5}\) over the interval of interest \(x \in [1, 2)\). Normalization of an input argument \(x\) to the interval \([1, 2)\) and construction of the final result is assumed to be performed before and after function approximation, respectively [15], [16].

### 2.3 Iterative Techniques

#### 2.3.1 Newton-Raphson (NR) Algorithm

The Newton-Raphson (NR) algorithm is based on Taylor’s theorem for finding successively improved approximations to a root of a differentiable function, starting with an initial estimate [17]. If the error of the initial approximation is \(e_o \leq 2^{-\eta}\), then to obtain an error \(e \leq 2^{-\ell}\) for some \(\ell > \eta\) using the quadratically converging NR algorithm, the required number of iterations is \(\lceil \log_2(\ell/\eta) \rceil\) [4]. We note that the latency of the NR iterative technique is directly proportional to the number of iterations required to achieve a desired precision, which in turn depends on the accuracy of the initial guess.

Approximating the reciprocal \(\frac{1}{x}\) using the NR recurrence can be written as \(x_{i+1} = x_i(2 - x x_i)\), where \(i = 0, 1, \ldots\). Fig. 2.3 shows the datapath for the reciprocal approximation obtained using one or more iterations. Here \(2^k\) starting reciprocal estimates for \(x \in [1, 2)\) lying between \((0.5, 1]\), can be pre-computed and stored in a LUT to obtain a more accurate second estimate. The LUT is addressed using the \(k\) most significant bits of the fraction \(d\) to obtain \(x_0\). In this datapath, the difference \(2 - d x_i\) is replaced with a bit inversion.
Figure 2.3. The datapath for NR reciprocal.

of $d_{x_i}$, performed by the \textit{bitinv} block. When a relatively large LUT is available, accurate initial estimates can be stored in an on-chip memory so that only one iteration is required to achieve the target accuracy. Therefore, the result can be obtained using the datapath in Fig. 2.3, but minus the multiplexer and the feedback loop shown with the dashed lines (NR+REC1).

In this case, as we will show, a relatively large LUT is required to support single-precision accuracy. However, when the on-chip memory size must be very limited, such as in area-constrained applications, accurate calculation of reciprocal requires two or more iterations and hence, imposes longer latency, as shown in Fig. 2.3 datapath. In applications where latency is critical, one can use a larger memory for a more accurate initial approximation in iterative techniques, or use non-iterative techniques, or use hybrid methods, respectively.

Since the wordlength of the modules impacts the latency, performance, and resource usage, it is important to find and use the smallest possible bitwidths for the operands and LUT variables in the datapath. Note that this is only correct when these modules are implemented on an ASIC or on the configurable slices of a FPGA, for example. When on-chip dedicated resources instead of configurable slices are utilized, increasing the precision may not have a direct impact on the performance and/or resource utilization. For example on a Xilinx Virtex-7 XC7V2000T FPGA, multiplication of two operands with the bitwidth varying between 21 and 35 bits uses the same number (i.e., 4) of dedicated built-in DSP48 modules. The same concept is true for Block memories. The 18-Kbit block RAMs (BRAMs) can be configured differently from $16K \times 1$ to $512 \times 36$. Therefore, 512 32-bit or 36-bit initial approximations utilize the same number of on-chip BRAMs.

There is a trade-off between the precision and size of the LUT and the required bitwidth of the intermediate variables in the arithmetic and logical modules. To determine the
minimum wordlength for all variables and the values stored in the LUT, we performed an exhaustive simulation-based error analysis of the datapath in Fig. 2.3. Using our custom library of bit-true fixed-point arithmetic and logical routines in Mex-C (for fast simulation), we computed 24-bit fixed-point values of \( f(x) \) over the full input domain of \( x \in [1, 2) \) at an input precision of \( 2^{-23} \) and compared those approximations with double-precision reference values. For this step, one can use other available fixed-point libraries, such as the Fixed-Point Toolbox from MATLAB [18]. In order to have the same 24-bit values compared to the transcendental function values in double-precision, the maximum error (i.e., the difference between approximated 24-bit fixed-point value and double-precision value after round to nearest) must be smaller than \( 2^{-23} \). Fig. 2.4 shows the error plot of the reciprocal calculation using NR method for single-precision accuracy.

![Error Plot](image)

**Figure 2.4.** The reciprocal error plot utilizing NR approach for single-precision accuracy.

Since the size of the required on-chip memory grows exponentially with \( k \), minimizing \( k \) is crucial to minimizing the on-chip memory usage. Our exhaustive simulation starts by minimizing the value of \( k \) for a predefined number of iterations (one or two) so that the approximation is accurate to the target precision. Without loss of generality, we only consider these two cases, however, the same procedure can be applied for three or more iteration scenarios. In this step, all variables (i.e., the initial approximate values stored in the LUT and the intermediate variables of the datapath) are given in full precision. After the minimum value of \( k \) has been determined, in the second step, the wordlengths of the intermediate signals and LUT entries are jointly minimized such that the final error is still kept less than the \( 2^{-23} \) upper bound. A detailed description of this procedure is shown in Algorithm 1. The
fixed-point format of the LUTs and the datapath are denoted using the notation \((WL, WF)\), where \(WL\) and \(WF\) denote the wordlength and the fraction length of the variables, respectively.

**Algorithm 1** Finding minimum values of \(k\), the wordlength and the fraction length of LUT entries and datapath signals.

- **Step 1:** Set the intermediate variables and LUT entries to double-precision format. Set the upper bound error to \(2^{-23}\).

- **Step 2:** For a faster simulation, choose \(N\) initial test points (e.g., \(10^5\)) by uniformly segmenting the \([1, 2)\) interval.

- **Step 3:** Find smallest value of \(m > 1\) such that the error \(e\) (i.e., the difference between the double-precision value and approximated fixed-point value of \(f(x)\)) is less than the upper bound for all test points.
  - (a) Start with \(k = 2\) and increment \(k\) until the error \(e\) is smaller than the upper bound for all test points.

- **Step 4:** Find minimum \(WL\) for LUT entries:
  - (a) If the entries of the LUT are unsigned choose the \((WL, WL - 1)\) fixed-point format for representing the values, otherwise, use the \((WL, WL - 2)\) format.
  - (b) Start by setting \(WL = 36\). \(^1\) Reduce \(WL\) until the error \(e\) becomes greater than the upper bound error.

- **Step 5:** Find minimum \(WL\) for datapath signals:
  - (a) Start by setting \(WL = 35\). \(^2\)
  - (b) Reduce \(WL\) until the error \(e\) becomes greater than \(2^{-23}\).

- **Step 6:** Exhaustive simulation of the error
  - (a) Set \(N = 2^{23}\) and use the chosen fixed-point formats for LUT entries and datapath signals.
  - (b) For all possible values of \(x = 1 + \delta\), where \(\delta = 2^{-23}\), compare the approximated fixed-point value (i.e., the output of the datapath) with the value of \(f(x)\) in double precision.
  - (c) If the error \(e\) becomes greater than the upper bound, restart the error analysis by increasing \(WL\) for the datapath signals and/or LUT entries.
  - (d) If \(WL > 36\) for LUT entries and \(WL > 35\) for datapath signals, increment \(k\) and go to Step 4.

\(^1\) Here we set the initial \(WL = 36\) since BlockRAMs in Virtex-4 are 18-Kbit that can be configured into \(512 \times 36\). The initial \(WL\) can be set to other values for custom designs.

\(^2\) Here we set the initial \(WL = 35\) since dedicated multipliers with input operands of 21 to 35 bits utilize the same number of DSP48 modules and run at the same clock frequency on a Xilinx Virtex-4. The initial \(WL\) can be set to other values for custom designs.

---

Table 2.1 shows the characteristics of the optimized datapath for the NR reciprocal operation with one iteration (\(NR+REC1\)) and with two iteration (\(NR+REC2\)). The row labeled “Datapath format” gives the fixed-point representation of the intermediate variables. The row labeled “Latency” gives the number of clock cycles required to obtain a result after applying the corresponding input to the datapath. The latency of a memory read operation and logical operations (such as bit inversion and shift operations) is one cycle. \(\tau_m\) and \(\tau_a\) denote the delay of a two-input multiplier and a two-input adder, respectively. It is important to note that when the target platform is an FPGA, \(\tau_m\) is the same for a range of different bitwidths. For example, multiplication of two input operands with bitwidth varying between 21 and 35 runs at the
same 596 MHz clock frequency. As the assumed bitwidth of the datapaths in this work lies between 26 and 31, we use the same multiplier latency \( \tau_m \) in all of these datapaths. For these implementations, the arithmetic and logic modules are not pipelined and only the output of the datapath is registered. As shown in Table 2.1, when latency is not a constraint (i.e., two iterations can be used), then the NR+REC2 design requires only roughly half of the memory space compared to the NR+REC1 datapath. Note that increasing the number of iterations further reduces the size of the LUT; however, for a high-performance implementation, we present our results for one and two iterations only. To generate 24-bit reciprocal function values with single-precision accuracy, it requires a 64×10-bit memory when using two iteration or a 4096×15-bit memory when using one iteration. The implementation results of NR+REC1 and NR+REC2 datapaths on a Xilinx Virtex-7 XC7V2000T with the speed grade of -2 are presented in Table 2.1.

<table>
<thead>
<tr>
<th>Design</th>
<th>NR+REC1</th>
<th>NR+REC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of iterations</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( k )</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>LUT size</td>
<td>( 2^{12} \times 15 )</td>
<td>( 2^{6} \times 10 )</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(15, 14)</td>
<td>(10, 9)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(26, 25)</td>
<td>(26, 25)</td>
</tr>
<tr>
<td>Latency</td>
<td>( 2\tau_m + 2 )</td>
<td>( 4\tau_m + 4 )</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>145</td>
<td>141</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>81</td>
<td>179</td>
</tr>
<tr>
<td>DSP48</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.1. Characteristics of the optimized NR reciprocal implementations.

Similar to the reciprocal operation, the NR iterative equation for square root can be written as \( x_{i+1} = 0.5(x_i + \frac{x}{x_i}) \) [4], which requires a division. A less complex strategy is to calculate the reciprocal square root \( \frac{1}{\sqrt{x}} \) and then multiply the result by \( x \) to obtain \( \sqrt{x} \). The iterative NR equation for \( \frac{1}{\sqrt{x}} \) can be written without divisions as \( x_{i+1} = 0.5 \times x_i(3 - x x_i^2) \) [4]. Note that subtracting from three can be approximated using bit inversion since \( 3 - x x_i^2 = 1 + (2 - x x_i^2) \) and the term \( 2 - x x_i^2 \) corresponds to a two’s complement [4], which can be approximated by a bit inversion. Fig. 2.5 shows the datapath for the above recurrence when one iteration (i.e., the datapath without the multiplexer and feedback loop shown with the dashed lines in NR+ISQT1), and two iterations (NR+ISQT2) are required to obtain the desired accuracy. The SHR block denotes the right arithmetic shift operation and the bit inversion and increment takes one cycle.

Fig. 2.6 shows the error plot for the reciprocal square root calculation using NR method for single-precision accuracy. The largest error between fixed-point calculation and MATLAB double-precision calculation is kept less than the \( 2^{-23} \) upper bound. Table 2.2
Figure 2.5. The datapath for NR reciprocal square root.

Figure 2.6. The reciprocal square root error plot utilizing NR method single-precision accuracy.

shows the characteristics of the datapath in Fig. 2.5. Similar to reciprocation, to obtain the result with 24-bit precision with one iteration, the NR+ISQT1 datapath needs a relatively large LUT to store sufficiently accurate initial approximations of the reciprocal square root. The memory requirement of the NR+ISQT2 datapath is almost half that of the memory required by NR+ISQT0 at the expense of longer latency. The implementation results of NR+ISQT1 and NR+ISQT2 datapaths on a Xilinx Virtex-7 XC7V2000T are presented in Table. 2.2.
Table 2.2. Characteristics of the optimized NR reciprocal square root implementations.

<table>
<thead>
<tr>
<th>Design</th>
<th>NR+ISQT1</th>
<th>NR+ISQT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of iterations</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$k$</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>LUT size</td>
<td>$2^{14} \times 15$</td>
<td>$2^{10} \times 11$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(15, 14)</td>
<td>(11, 10)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(29, 27)</td>
<td>(26, 24)</td>
</tr>
<tr>
<td>Latency</td>
<td>$3\tau_m + \tau_a + 2$</td>
<td>$6\tau_m + 2\tau_a + 4$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>161</td>
<td>141</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>114</td>
<td>273</td>
</tr>
<tr>
<td>DSP48</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

2.3.2 Goldschmidt (GS) Algorithm

Another multiplicative technique for approximating elementary functions is the binomial series expansion, also called the Goldschmidt (GS) algorithm [19]. The attractiveness of this technique for hardware implementation is that the two multiplications can be performed simultaneously in parallel avoiding the two or more dependent multiplications in the NR method. Moreover, both the square root and reciprocal square root can be calculated with the same computational complexity and latency. Let $x_0 = Y_0$ be a suitably accurate initial estimate of $\frac{1}{x}$. Starting with $g_0 = x_0$ and $d_0 = x x_0$, the updated reciprocal approximation is given by $g_{j+1} = g_j Y_{j+1}$, where $Y_{j+1} = 2 - d_j$ and $d_{j+1} = d_j Y_{j+1}$ for $j = 0, 1, 2, \ldots$. Fig. 2.7 shows the datapaths of the GS reciprocal algorithm with one iteration (GS+REC1, i.e., GS+REC2 without the feedback dashed line) and two iterations (GS+REC2) are required to obtain the desired accuracy. Table 2.3 shows the implementation characteristics of the datapaths in Fig. 2.7 on a Xilinx Virtex-7 XC7V2000T FPGA after being optimized using our bit-true simulator to achieve single-precision accuracy. By
Figure 2.8. The reciprocal error using GS approach for single-precision accuracy.

comparing Tables 2.1 and 2.3 one can see that utilizing two paralleled multipliers in the GS+REC2 datapath reduces the latency compared to the latency of the NR+REC2 datapath.

<table>
<thead>
<tr>
<th>Design</th>
<th>GS+REC1</th>
<th>GS+REC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of iterations</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>k</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>LUT size</td>
<td>$2^{12} \times 15$</td>
<td>$2^{6} \times 13$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(15, 14)</td>
<td>(13, 12)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(27, 26)</td>
<td>(27, 26)</td>
</tr>
<tr>
<td>Latency</td>
<td>$2\tau_m + 2$</td>
<td>$3\tau_m + 4$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>164</td>
<td>164</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>77</td>
<td>103</td>
</tr>
<tr>
<td>DSP48</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.3. Characteristics of the optimized GS reciprocal implementations.

To approximate the reciprocal square root using the GS algorithm, let $x_0 = Y_0$ be an initial estimate of $\frac{1}{\sqrt{x}}$ and let $d_0 = x$. The reciprocal square root approximation is updated by $g_{j+1} = g_j Y_{j+1}$, where $Y_{j+1} = 1.5 - 0.5 \, d_{j+1}$ and $d_{j+1} = d_j Y_j^2$ for $j = 0, 1, 2, \ldots$. Fig. 2.9 shows the datapath of the GS reciprocal square root algorithm with one iteration (GS+ISQT1, i.e., GS+ISQT2 without the feedback dashed line) and two iterations (GS+ISQT2) are required to obtain the desired accuracy. Table 2.4 shows the characteristics of the optimized datapaths in Fig. 2.9. Similar to the NR scheme, the GS+ISQT2 datapath requires almost half of the memory required by the GS+ISQT1 design at the expense of a longer latency. While the latencies of the datapaths in Tables 2.2 and 2.4 are close to each other, one can reduce the latency of the GS+ISQT2 datapath by using an additional multiplier and calculating $g_{j+1}$ and $Y_{j+1}^2$ in parallel. Fig. 2.10 shows the absolute error plot of the GS reciprocal square root.
Figure 2.9. The datapath for GS reciprocal square root.

<table>
<thead>
<tr>
<th>Design</th>
<th>GS+ISQT1</th>
<th>GS+ISQT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of iterations</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$k$</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>LUT size</td>
<td>$2^{15} \times 15$</td>
<td>$2^6 \times 11$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(15, 14)</td>
<td>(11, 10)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(27, 26)</td>
<td>(27, 26)</td>
</tr>
<tr>
<td>Latency</td>
<td>$3 \tau_m + \tau_a + 2$</td>
<td>$6 \tau_m + 2 \tau_a + 4$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>141</td>
<td>141</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>169</td>
<td>293</td>
</tr>
<tr>
<td>DSP48</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2.4. Characteristics of the optimized GS reciprocal square root implementations.

calculation for single-precision accuracy.

Several accurate initial approximation techniques have been proposed that can be utilized in conjunction with other iterative techniques, such as the NR and GS algorithms. The idea is to use slightly more logic and a smaller LUT to produce a sufficiently accurate initial approximation that allows the iterative algorithm to converge to an accurate result with no iterations. These methods trade off between the latency and the memory requirements of an iterative algorithm and the required resources and latency of the initial approximation logic.

2.4 NON-ITERATIVE TECHNIQUES

Non-iterative techniques compute the function value directly without convergent iterations and can be sub-divided into compute-bound methods [5], [20] and hybrid methods [21], [22]. Compute-bound schemes minimize the storage size but require a
relatively large number of multiplications and additions, resulting in longer execution times. For example, the rational approximation scheme in [5] requires at least one division operation and the methods in [23] and [24] use a very small LUT together with square, cubic, or higher-degree polynomial approximations. Various optimized techniques for computing square and cubic polynomials have been proposed [25].

### 2.4.1 Ito’s Algorithm

Ito et al. [26] and [27] propose to use a piecewise linear approximation to produce an accurate initial approximation for the reciprocal and reciprocal square root recurrence. In [26] they propose linear approximation schemes in which both pre-computed reciprocals of $x$ and the differences between successive reciprocals of $x$ are stored in a table. This method requires a LUT and a multiply-accumulate unit. The scheme in [27] requires a relatively small memory with a simple computation. For reciprocation, the initial approximation to the reciprocal $x^{-1}$ of a given normalized denominator $x = 1.d_1d_2\ldots d_n$, where $1 \leq x < 2$, can be obtained by computing $x_0 = \hat{c}\hat{x}$ where $\hat{x} = 1.d_1d_2\ldots d_k\bar{d}_{k+1}\bar{d}_{k+2}\ldots \bar{d}_n$. Here $\bar{d}_i$ denotes the complement of the $i$-th bit of $x$, and

$$\hat{c} = \frac{1}{x_h(x_h + 2^{-k})} - \frac{2^{-2k-3}}{x_h^4}.$$ 

Note that the addition in the linear approximation is removed by using a slightly modified operand $\hat{x}$. This modification also reduces the table size as only one coefficient has to be stored instead of two. Similarly, a piecewise linear approximation is proposed in [27] for the

![Figure 2.10. The reciprocal square root error plot utilizing GS approach for single-precision accuracy.](image)
reciprocal square root as \( x_0 = \tilde{x} \) where \( \tilde{x} = 1.d_1d_2 \ldots d_{k-1}\tilde{d}_k\tilde{d}_{k+1} \ldots \tilde{d}_n \) and

\[
\tilde{c} = 2c_1 - \frac{7 \times 2^{-2k-4}}{\tilde{x}_h^3 \sqrt{\tilde{x}_h}}.
\]

Table 2.5 shows the characteristics of the NR reciprocal (ITO+NR+REC) and reciprocal square root (ITO+NR+ISQT) implementations utilizing Ito’s approximations for single-precision accuracy. The precision of the datapath for Table 2.5 applies to both the multiply module in the initialization datapath and also to the datapath of the NR algorithm. An important point to note is that ITO+NR+REC and ITO+NR+ISQT use memory blocks as small as the one used in NR techniques with one iteration, while their latencies are only one \( \tau_m \) greater than the NR techniques for reciprocal and reiprocal square root with one iteration. Therefore, significant memory savings are achieved at the expense of slightly increased latency. Fig. 2.11 and Fig. 2.12 show the absolute error plots of the reciprocal and reciprocal square root calculation utilizing Ito’s approximation for single-precision accuracy, respectively.

### 2.4.2 Hung’s Algorithm

Another non-iterative solution uses numerical series expansions. Hung et al. [21] use a Taylor series to implement the reciprocal operation in which \( \frac{1}{x} \) is approximated by combining the first two terms of the Taylor series expansion as \( (x_h - x_l)/x_h^2 \), where \( x_h \) denotes the \((k + 1)\)-st most significant bit of \( x \) (i.e., \( x_h = 2^0 + 2^{-1}d_1 + \cdots + 2^{-k}d_k \) and \( 1 \leq x_h \leq 2 - 2^{-k} \)) and \( x_l = x - x_h \) (i.e., \( 0 \leq x_l < 2^{-k} \)). Fig. 2.13 (a) shows a corresponding datapath (HUNG+REC) that approximates the reciprocal of \( x \) using a LUT, where \( 2^k \) different values of \( 1/x_h^2 \) are precomputed and stored in a \( 2^k \times c_{WL} \)-bit memory. Using the same definitions for \( x_h \) and \( x_l \), Hung et al. also propose to use \( (x(x_h - x_l/2))/x_h^{1.5} \) to approximate the square root [21]. The corresponding datapath (HUNG+SQT) is shown in Fig. 2.13 (b).

Table 2.6 shows the characteristics of the optimized datapaths for the reciprocal and square root using Hung’s two approximations for single-precision accuracy. It can be noticed that

<table>
<thead>
<tr>
<th></th>
<th>ITO+NR+REC</th>
<th>ITO+NR+ISQT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT size</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(12, 12)</td>
<td>(13, 13)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(27, 26)</td>
<td>(28, 26)</td>
</tr>
<tr>
<td>Latency</td>
<td>( \tau_m + (2\tau_m + 2) )</td>
<td>( \tau_m + (3\tau_m + 3) )</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>95</td>
<td>84</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>272</td>
<td>176</td>
</tr>
<tr>
<td>DSP48</td>
<td>5</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 2.5. Characteristics of the optimized NR reciprocal datapaths utilizing Ito’s initial approximations.
compared to other methods, Hung’s method is able to run much faster due to its smaller execution part. Unfortunately, the results in Table 2.6 shows that a relatively large memory is required. Fig. 2.14 and Fig. 2.15 show the error plots of the reciprocal and reciprocal square utilizing Hung’s approach for single-precision accuracy, respectively.

### 2.4.3 Jeong’s Algorithm

An improved scheme that requires a smaller memory is the technique proposed by Jeong et al. [28]. This scheme uses an approximation of the form $A(2 - Ax)$, where $A = (x_h - x_l)/x_h^2$. The corresponding datapath (JEONG+REC) is shown in Fig. 2.16.

Table 2.7 shows the characteristics of the datapath in Fig. 2.16 for single-precision accuracy. While the JEONG+REC datapath requires a much smaller memory compared to
Figure 2.13. The datapath of Hung et al. technique for (a) reciprocal and (b) square root.

Figure 2.14. The reciprocal error plot utilizing Hung’s approach for single-precision accuracy.

<table>
<thead>
<tr>
<th>k</th>
<th>HUNG+REC</th>
<th>HUNG+SQT</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>LUT size</td>
<td>$2^{12} \times 26$</td>
<td>$2^{12} \times 26$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(26, 25)</td>
<td>(27, 26)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(26, 25)</td>
<td>(27, 26)</td>
</tr>
<tr>
<td>Latency</td>
<td>$\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a + 1$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>232</td>
<td>92</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>102</td>
<td>107</td>
</tr>
<tr>
<td>DSP48</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.6. Characteristics of reciprocal and square root implementations using the optimized Hung et al. technique.
Figure 2.15. The reciprocal square root error plot utilizing Hung’s approach for single-precision accuracy.

HUNG+REC, it has a longer latency. Fig. 2.17 shows the reciprocal error plot utilizing Jeong’s approach for single-precision accuracy.

Takagi [22] also proposed an efficient method for generating a power of an operand (e.g., $x^{-1}$ and $x^{-0.5}$) that can be used to approximate the reciprocal and square root operations. To approximate $x^k$, Takagi’s method requires a table look-up, a multiplication, and an operand modification, which is carried out by a bitwise inversion, a shift operation, and/or a redundant binary Booth recoding. For approximating the reciprocal and reciprocal square root, a $2^{11} \times 25$ LUT and a $(22, 22)$ multiplier (without the need for an addition) are required for each operation.
Figure 2.17. The error measurement for Jeong reciprocal.

<table>
<thead>
<tr>
<th></th>
<th>JEONG+REC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>6</td>
</tr>
<tr>
<td>LUT size</td>
<td>$2^n \times 14$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(14, 13)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(31, 30)</td>
</tr>
<tr>
<td>Latency</td>
<td>$3\tau_m + \tau_a + 2$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>94</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>277</td>
</tr>
<tr>
<td>DSP48</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2.7. Characteristics of implementing reciprocal using the optimized Jeong et al. technique.

2.5 Hybrid Algorithm

One efficient strategy for the numerical approximation of transcendental functions is to use polynomials. Fig. 2.18 plots $f(x)$ for the three elementary functions over the desired domain $[1, 2)$. Since over this interval these functions are well behaved and slowly varying, the interval $x \in [1, 2)$ can be partitioned into $2^k$ sub-intervals (segments), where the $k$ most significant bits of the fraction $d$ are used as a segment index. Then a low-degree polynomial $g(x) = \sum_{j=0}^{p} a_j x^j$ can be used to approximate $f(x)$ within each segment. After defining $g(x)$ for each segment (i.e., finding its coefficients), the value of $f(x_i)$ can be approximated by calculating the value of the fitting polynomial $g(x)$ at $x = x_i$.

Hybrid algorithms allow high-speed function approximation by combining relatively small LUTs with linear or quadratic interpolating polynomials [29]. For example, [16] and [30] use first-order and second-order interpolating polynomials, respectively. The designs in [31] and [32] use second-order piecewise polynomial approximations of the functions based on Chebyshev approximations, Taylor expansions, and minmax approximations, respectively.
In [15], Schulte and Swartzlander use a uniform segmentation and polynomial approximation (SCH+REC+ISQT) in which the coefficients for each segment are determined using a Chebyshev series approximation. The polynomial terms are generated in parallel and summed using a multi-operand adder. Ercegovac et al. [31] propose a three-step method (ERC+REC+ISQT) based on argument reduction, evaluation using a few Taylor series terms, and then post processing. In the argument reduction step the input is scaled to be close to one, and after evaluation of the function using a few Taylor series terms, the result is post-processed. Cao and Wei [16] used a second-degree interpolating polynomial (CAO+REC+ISQT) and store only one polynomial coefficient and the function value for each sub-interval and then calculate the other two coefficients. Jain et al. [30] also used a quadratic interpolating polynomial with modified arithmetic to eliminate most of the multiplication operations (JAIN+REC+SQT).

In Schulte’s scheme, the size of the required memory (see Table II in [15]) using a linear approximation that produces exactly rounded results is actually too large (i.e., $2^{18} \times 57$). Thus we present the characteristics of this method assuming a quadratic polynomial approximation. Comparing our implementation OURS+REC+ISQT with the four competing datapaths in Table 2.8, we see that our unified design and the JAIN+REC+SQT datapath require only one adder and one multiplier. Note that our quadratic design requires 2.4 times fewer memory bits than JAIN+REC+SQT. Even though the ERC+REC+ISQT design requires fewer memory bits than the other designs in Table 2.8, it uses the greatest number of dedicated multipliers and it also has the longest latency. Note that the sum of the required resources for implementing just the reciprocal and square root using the multiplicative
techniques, whose characteristics are given in Tables 2.1 to 2.7, is significantly more than the resources required by the hybrid schemes.

Das Sarma and Matula [33] propose the faithful interpolation method for approximating the elementary functions, which reduces the table size required for the piecewise linear approximation compared to Ito’s scheme [26]. In this method, only the approximate reciprocals of $x$ are stored in a table. Although this technique can be applied for approximating several elementary functions, only the implementation results for the reciprocal operation were described. For single-precision accuracy, this method requires a $2^{12} \times 26$ LUT, a $(15, 15)$ multiplier and a 24-bit adder for approximating the reciprocal. This method also uses redundant binary Booth recoding. Pineiro et al. [32] propose an enhanced minmax quadratic approximation and an optimized evaluation of the second-order polynomial using a specialized squaring unit and fused accumulation tree. The coefficients were approximated using the symbolic algebra system Maple. Our quadratic polynomial approximation, which produces exactly rounded results, requires 1.2 times fewer memory bits for calculating $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$ than the unified designs in [32], which produce results with a lower accuracy (see Table 2 in [32]).

Table 2.8 presents the characteristics of several unified techniques that implement both the reciprocal and reciprocal square root functions. Earlier design in [34] uses the uniform domain segmentation and curve fitting approximations to accurately implement the desired elementary functions to single-precision floating-point accuracy. While we used a fully pipelined datapath in [34] to achieve maximum data throughput, in this thesis we employ a compact implementation of quadratic polynomial approximation using shared arithmetic modules, as shown in Fig. 2.19 (OURS+REC+ISQT). While each function requires a separate memory to store the polynomial coefficients, sharing a single shared datapath among all functions produces an especially compact implementation. The three LUTs are addressed using the $k$ most significant bits of the fraction $d_h = \ldots d_k$ and $d_l = \ldots d_n$. The $\textit{Rounding}$ block uses the round-to-nearest-odd technique and transforms the approximated values into 24-bit rounded results. On a Xilinx Virtex-7 XC7V2000T FPGA, the proposed datapath in Fig. 2.19 can run at 132 MHz, requiring 237 configurable slice LUTs and three DSP48 modules to obtain results with single-precision accuracy.
2.6 Comparative Analysis

It was shown that the non-iterative techniques have less latencies, but require larger memories than the iterative methods. By comparing iterative methods one can see that utilizing two multipliers in the GS+REC1 datapath reduces the latency compared to the latency of the NR+REC1 datapath and requires fewer LUTs and DSP48 blocks.

Non-iterative methods ITO+NR+REC and ITO+NR+ISQT use memory blocks as small as the one used in NR techniques with two iterations, while their latencies are only one $\tau_m$ greater than the NR techniques for reciprocal and reciprocal square root with one iteration. The size of the look-up table increases linearly with the bitwidth of the operand instead of increasing exponentially. Significant memory savings are achieved at the expense of slightly increased latency. Thus, if the operand’s bitwidth is very large, such as in the double-precision format, we should consider using Ito’s method.
By using numerical series expansions, Hung’s method is able to achieve the required precision with the highest frequency and the least additional calculation blocks while it requires a relatively large memory for the LUT. Thus, in applications that the bitwidth of the operand is relatively small and high performance is required, Hung’s method is a good choice to calculate reciprocal and square root.

To decrease the size of memory in Hung’s method, Jeong’s method apply the same rationale as Hung and also use the iterative mechanism. As a tradeoff, two more multipliers are added on the datapath to achieve the required precision while using significantly smaller LUT. Comparing with the Table 2.5, we can notice that the performance and resource utilization of Jeong’s method is very similar to those of Ito’s method.

As the density of modern FPGAs continuously increasing, Hung’s method offers the most suitable approach for FPGA systems, which provides the highest frequency while utilizing fewer hardware resources when the bitwidth of the input is relatively small. Ito’s method is able to use smallest memories while needing more controller and computing logics. Thus, in this work, the calculation of reciprocal and square root in fixed-point and floating-point formats supporting arbitrary precisions will use Hung’s method for high performance implementation without strict limitations on the memory size, or when the operand bitwidth is relatively small. We will use Ito’s method when the operand bitwidth is larger than 32 bits, or when the memory size is limited.
2.7 SUMMARY

Choosing suitable algorithms and efficient hardware implementation of elementary functions requires careful consideration of various parameters, such as the convergence rate of the algorithm, resource constraints, and required performance. This chapter explored the fundamental trade-offs between alternative multiplicative techniques for high-performance implementation of the three key elementary operations $\frac{1}{x}$, $\sqrt{x}$ and $\frac{1}{\sqrt{x}}$ with single-precision floating-point accuracy.

Choosing the most suitable technique depends on the available on-chip memory, logical resources, dedicated multipliers, and affordable latency. To permit the quantitative analysis of the design trade-offs, we developed a set of parameterizable fixed-point and floating-point software routines in Mex-C that allows us to minimize the precision of datapath variables and the size of the LUTs for a given precision (in this chapter, single-precision accuracy). We performed exhaustive bit-true simulations across the approximation interval $[1, 2)$ and measured the absolute errors to ensure that the proposed datapath supported 24-bit final accuracy required for single-precision floating-point arithmetic, while minimizing the size of the required hardware.
CHAPTER 3
ARBITRARY PRECISION SCHEME FOR
FIXED-POINT RECIPROCAL, SQUARE
ROOT, AND RECIPROCAL SQUARE ROOT

This chapter presents the design and implementation of datapaths for calculating reciprocal, square root, and reciprocal square root in fixed-point representation with arbitrary precisions, i.e., the results of calculating \( f(x) \) can be accurate up to \( m \) bits after the radix point. To support arbitrary precisions, the length of the integer part and the fractional part should be both parametric.

3.1 FIXED-POINT DESIGN AND IMPLEMENTATION

3.1.1 Introduction

The hardware implementation of three elementary functions should support input values with arbitrary precisions in various binary fixed-point formats. To support this, the datapath contains 4 blocks: sign, normalization, operation, and denormalization, as shown in Fig. 3.1. First, the absolute value of the input fixed-point number \( x \) is derived as the datapath operates on unsigned values. The sign will be stored for using in later stages. Then, the “norm” block normalizes the input into 1.WF format by shifting \( x \) and storing the shift direction and the number of positions. Next module will calculate the reciprocal, square root, or reciprocal square root of the normalized input using Hung’s or Ito’s approach, as presented in Chapter 2. The denormalization process varies for different arithmetic operations of reciprocal, square root, and reciprocal square root. The last step is simply assigning the sign to the denormalized results.

3.1.2 Arithmetic Design

Assume that the input \( x \) is defined in an arbitrary fixed-point format of \((WI,WF)\) where \( WI \) and \( WF \) denote the integer and the fractional length of the input \( x \), respectively, and the output is in fixed-point format \((WIO,WFO)\), where \( WIO \) and \( WFO \) denote the output integer length and the fractional length, respectively. Algorithm 2 describes the reciprocal operation. Note that the sign and normalization blocks are the same for all three operations. The flag \( sign \) will be used to save the sign of input. Then the absolute value of \( x \) will be normalized. The output of the normalization block has the same wordlength as the input. After calculating
Algorithm 2 Fixed-point calculation of $\frac{1}{x}$ with an arbitrary precision

Input $x$ is a $WL$-bits input in $(WI,WF)$ fixed-point format;

2: The output is in $(WIO,WFO)$ fixed-point format;

Store the sign of the input $x$ in “sign”;

4: $x_{in} = |x|$;

Find the position of leading one in the input $x$;

6: Normalize the input to $[1,2)$: $x_n = \text{norm}(x_{in}) = x_{in} \times 2^{\text{dir} \times \text{pos}}$;

- Shift right $x$ if $x > 1$ and shift left if $x < 1$ so that $x$ resides in $[1,2)$;

- The number of shift positions is stored in “pos” register;

- The direction of the shift operation is stored in “dir”. If the shift direction is left, $dir = 1$, if the direction is right, $dir = -1$;

- The normalized number is in $(1, WI+WF-1)$ format;

Calculate the reciprocal of the normalized number using Hung’s or Ito’s approach to generate $out_n$

- $out_n = \frac{1}{x \times 2^{\text{dir} \times \text{pos}}}$

8: Append $WF$ zeros to the integer part of $out_n$;

Denormalization: Shift the normalized result $out_n$ based on the direction “dir” and shift positions “pos”: $out = out_n \times 2^{\text{dir} \times \text{pos}} = \frac{1}{x}$;

10: if $dir == -1$ then

    $out = out_n >> \text{pos}$;

12: else

    $out = out_n << \text{pos}$;

14: end if

Adjusting the bidwidth of the integer and fractional parts;

16: Apply the sign back to the output;
Figure 3.1. Fixed-point calculation of \( f(x) \) over \( x \) with an arbitrary precision.

\( f(x) \) for \( x \in [1, 2) \), the output will be within \( [0.5, 1) \), \( [1, \sqrt{2}) \), and \( [1, 1/\sqrt{2}) \) for reciprocal, square root, and reciprocal square root, respectively. The denormalization block operates differently for each of the three operations. For calculating reciprocal, there is no scaling required, and shifting for denormalization has the same direction and number of positions as those for normalization. However, for square root and reciprocal square root, the number of shift positions are different than those for the normalization step. Algorithm 3 and 4 present the denormalization process of square root and reciprocal square root operations, respectively.

The denormalization process for square root and reciprocal square root are as follows:

- The shifting direction of the denormalization process for square root is reverse of the shifting direction used in the normalization step.

- The shifting direction of the denormalization process for reciprocal square root is the same as the shifting direction used in the normalization step.

- For square root denormalization, if \( pos \) is an even number, shift \( out_n \frac{pos}{2} \) positions to the shifting direction. Hence:

\[
out = out_n \div \sqrt{2^{dir \times pos}} = out_n \div 2^{\frac{dir \times pos}{2}} \tag{3.1}
\]

When the shifting direction of normalization is right (\( dir = -1 \)), the denormalization process should shift the \( out_n \) to the left \( \frac{pos}{2} \) positions. Hence:

\[
out = out_n \div \sqrt{2^{-pos}} = out_n \times 2^{\frac{pos}{2}} = out_n < \frac{pos}{2} \tag{3.2}
\]
Algorithm 3 Denormalization process for fixed-point square root

Calculate the square root of the normalized input $x_n$ using Hung’s or Ito’s approach to generate $out_n = \sqrt{x \times 2^{dir \times pos}}$;

2: Append $\frac{W_I}{2}$ zeros to the integer part of $out_n$;

Denormalization: Shift the normalized result $out_n$ based on the direction “$dir$” and shift positions “$pos$”: $out = out_n \div \sqrt{2^{dir \times pos}} = \sqrt{x}$;

4: $out_{odd} = out_n \times \sqrt{2}$;

if $(pos \mod 2) == 1$ then

6: $pos_{half} = \frac{pos-1}{2}$;

   if $dir==1$ then

8: $out = out_{odd} << pos_{half}$;

   else

10: $out = out_{odd} >> pos_{half} + 1$;

   end if

12: else

$pos_{half} = \frac{pos}{2}$;

14: if $dir==1$ then

16: $out = out_n << pos_{half}$;

   else

18: $out = out_n >> pos_{half}$;

   end if

end if
When the shifting direction of normalization is left ($dir = 1$), the denormalization process should shift the $out_n$ to the right $\frac{pos}{2}$ positions. Hence:

$$out = out_n \div \sqrt{2^{pos}} = out_n \div 2^{\frac{pos}{2}} = out_n >> \frac{pos}{2} \quad (3.3)$$

If $pos$ is an odd number, the normalized result should be scaled by multiplying by $\sqrt{2}$ and then shifting $out_n^{\frac{pos-1}{2}}$ or $\frac{pos+1}{2}$ positions to the shifting direction. Hence:

$$out = out_n \div \sqrt{2^{dir \times pos}} = out_n \div 2^{\frac{dir}{2}} \div 2^{\frac{dir \times (pos-1)}{2}} \quad (3.4)$$

When the shifting direction of normalization is right ($dir = -1$), the denormalization process should shift $out_n^{\frac{pos-1}{2}}$ positions to the right. Hence:

$$out = out_n \times \sqrt{2} < < \frac{pos - 1}{2} \quad (3.5)$$

$$out = out_n \times \sqrt{2} >> \frac{pos + 1}{2} \quad (3.6)$$

When the shifting direction of normalization is left ($dir = 1$), the denormalization process should also shift the $out_n^{\frac{pos+1}{2}}$ positions to the left. Hence:

$$out = out_n \times 2^{\frac{pos+1}{2}} = out_n \times 2^{\frac{pos+1}{2}} \quad (3.7)$$

$$out = out_n \times \sqrt{2} >> \frac{pos + 1}{2} \quad (3.8)$$

• For reciprocal square root denormalization, if the $pos$ is an even number, shift $out_n^{\frac{pos}{2}}$ positions to the shifting direction. Hence:

$$out = out_n \times \sqrt{2^{dir \times pos}} = out_n \times 2^{\frac{dir \times pos}{2}} \quad (3.9)$$

When the shifting direction of the normalizatiob is right ($dir = -1$), the denormalization process should also shift the $out_n^{\frac{pos}{2}}$ positions to the right. Hence:

$$out = out_n \times \sqrt{2^{-pos}} = out_n \times 2^{-\frac{pos}{2}} = out_n >> \frac{pos}{2} \quad (3.10)$$

When the shifting direction of the normalization is left ($dir = 1$), the denormalization process should also shift the $out_n$ to the left $\frac{pos}{2}$ positions. Hence:

$$out = out_n \times \sqrt{2^{pos}} = out_n \times 2^{\frac{pos}{2}} = out_n << \frac{pos}{2} \quad (3.11)$$

If $pos$ is an odd number, the $out_n$ should be scaled by multiplying $\frac{1}{\sqrt{2}}$ and then shifting $\frac{pos-1}{2}$ or $\frac{pos+1}{2}$ positions to the shifting direction. Hence:

$$out = out_n \times \sqrt{2^{dir \times pos}} = out_n \times 2^{\frac{dir}{2}} \times 2^{dir \times \frac{pos-1}{2}} \quad (3.12)$$
Algorithm 4 Denormalization process for fixed-point reciprocal square root

Calculate the reciprocal square root of the normalized number using Hung’s or Ito’s approach to generate $out_n = \frac{1}{\sqrt{2^{dir \times pos}}}$;

2: Append $\frac{WF}{2}$ zeros to the integer part of $out_n$;

Denormalization: Shift the normalized result $out_n$ based on the direction “dir” and shift positions “pos”: $out = out_n \times \sqrt{2^{dir \times pos}} = out_n \times 2^{\frac{dir \times pos}{2}} = out_n \times 2^{\frac{dir \times (pos-1)}{2}} \times \sqrt{2} = \frac{1}{\sqrt{x}}$;

4: $out_{odd} = out_n \times \sqrt{2}$;

if $(pos \mod 2) == 1$ then

6: $pos_{half} = \frac{pos-1}{2}$;

if $dir == -1$ then

8: $out = out_{odd} >> pos_{half}$;

else
9: $out = out_{odd} << pos_{half}$;

end if

12: else

$pos_{half} = \frac{pos}{2}$;

14: if $dir == -1$ then

16: $out = out_n >> pos_{half}$;

else
17: $out = out_n << pos_{half} + 1$;

end if

end if
When the shifting direction of the normalization is right (dir = \(-1\)), the
denormalization process should shift \(\text{out}_n \frac{\text{pos} - 1}{2}\) position to the right. Hence:

\[
\text{out} = \text{out}_n \times 2^{-\frac{1}{2}} \times 2^{\frac{-(\text{pos} - 1)}{2}} = \text{out}_n \times \frac{1}{\sqrt{2}} \times 2^{\frac{-(\text{pos} - 1)}{2}}
\]  

(3.13)

\[
\text{out} = \text{out}_n \times \frac{1}{\sqrt{2}} >> \frac{\text{pos} - 1}{2}
\]  

(3.14)

When the shifting direction of the normalization is left (dir = 1), the denormalization
process should shift \(\text{out}_n\) to the left \(\frac{\text{pos} + 1}{2}\) positions. Hence:

\[
\text{out} = \text{out}_n \times 2^{\frac{1}{2}} \times 2^{\frac{-\text{pos} - 1}{2}} = \text{out}_n \times \frac{1}{\sqrt{2}} \times 2^{\frac{-\text{pos} - 1}{2}}
\]  

(3.15)

\[
\text{out} = \text{out}_n \times \frac{1}{\sqrt{2}} << \frac{\text{pos} + 1}{2}
\]  

(3.16)

For example, assume \(x = 16\) is stored in binary format \((WI, WF) = (6, 6)\) as
\(x = 010000.000000\). The normalized value of \(x\) is \(x_n = 1.0000000000000\) is derived by shifting
right \(\text{pos} = 4\) positions. Assume the output is in \((WIO, WFO) = (6, 6)\) format. The
normalized results for the three operations \(\frac{1}{x}, \sqrt{x}, \frac{1}{\sqrt{x}}\) are the same as \(\text{out}_n = 1.0000000000000\).
For reciprocal operation, the denormalization process should shift the normalized result to the
right, as the same direction as used in the normalization, \(\text{pos} = 4\) bits to get the final output
\(\text{out}_{rec} = 000000.0000100\), which is 0.0625. For square root operation, the denormalization
process should shift the normalized result to left \(\text{pos}_{half} = 2\) units to generate the final output
\(\text{out}_{sqrt} = 000100.000000\), which is 4. For reciprocal square root, the denormalization process
should shift the normalized result to the right \(\text{pos}_{half} = 2\) bits to generate the final output
\(\text{out}_{isqrt} = 000000.010000\), which is 0.25. Assume \(x = 8\) is in \((WI, WF) = (8, 8)\) format as
\(x = 00010000.00000000\). The normalized value of \(x\) is \(x_n = 1.000000000000000\), which can
be obtained by shifting \(x\) right \(\text{pos} = 3\) bits. Assume the output format is the same as
\((WIO, WFO) = (8, 8)\). The normalized results for the three operations \(\frac{1}{x}, \sqrt{x}, \frac{1}{\sqrt{x}}\) are the
same as \(\text{out}_n = 1.000000000000000\). For reciprocal, the denormalization process should shift
\(\text{out}_n\) to the right \(\text{pos} = 3\) bits to get the final output \(\text{out}_{rec} = 00000000.00100000\), which is
0.125. Because \(\text{pos}\) is an odd number and \(\text{dir} = -1\), for square root, we should first scale \(\text{out}_n\)
by \(\sqrt{2}\), and shift left the result by \(\frac{\text{pos} - 1}{2}\) = 1 bit, and hence, \(\text{out}_{sqrt} = 000000010.11010100\),
which is 2.828125. For reciprocal square root, scaling \(\text{out}_n\) by \(\sqrt{2}\), and shifting the result
right by \(\frac{\text{pos} + 1}{2}\) = 2 bits to get \(\text{out}_{isqrt} = 00000000.01011010\), which is 0.3515625.

Another challenging process in the denormalization is to adjust the bitwidth of the
datapath. Note that for reciprocal and reciprocal square root, the bitwidth selection of the
result’s fractional length depends on the bitwidth of the input’s integer length, while the
integer length of result depends on the input’s fractional length. The user can define the output
bitwidth with arbitrary precisions. This discussion is separated into two cases, the input \( x > 1 \), and the input \( x < 1 \). Fig. 3.2, 3.3, and 3.4 present the three operations’ bitwidth selection mechanism for these two cases.

![Table](image)

**Figure 3.2.** Bitwidth selection for the reciprocal operation.

In the first case, the input \( x > 1 \) and the leading one is on the lefthand side (LHS) of the binary point. Consider \( x = 01110111.0100 \) with the format \((W_I, W_F) = (8, 4)\) and the decimal value 119.25. The normalized input is thus \( x_n = 1.11011101000 \) in \((W_{I_n}, W_{F_n}) = (1, 11)\) format, which is 1.86328125 in decimal. For the reciprocal operation, the output of the normalized block is 0.10001001011 with the decimal value 0.53662109375, for reciprocal square root is 0.10111011100 with the decimal value 0.732421875, and for square-root is 1.01011101100 with the decimal value 1.365234375. Notice that for the reciprocal operation, the final result will be smaller than 0.000001 because \( x > 01000000.0 \). For reciprocal square root, the final result will be smaller than 0.001, and if the operation is square root, the final result will be larger than 01000.0. Thus, different operations have their own optimal datapath bitwidths. The bitwidth of the fractional part of the reciprocal and
### Figure 3.3. Bitwidth selection for the reciprocal square root operation.

**x≥1:**  
**Binary representation**  

<table>
<thead>
<tr>
<th>Input</th>
<th>01110111 0100</th>
<th><strong>Bitwidth (8,4)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix point</td>
<td></td>
<td>WM WF</td>
</tr>
</tbody>
</table>

**Normalized input:**  

<table>
<thead>
<tr>
<th></th>
<th>1 1101101000</th>
<th>1 WM+WF-1</th>
</tr>
</thead>
</table>

**Normalized output:**  

<table>
<thead>
<tr>
<th></th>
<th>0 1011011100</th>
<th>1 WM+WF-1</th>
</tr>
</thead>
</table>

**Highest precision output:**  

<table>
<thead>
<tr>
<th></th>
<th>0 0001011101100</th>
<th>1 (WM-1)/2 + WL-1</th>
</tr>
</thead>
</table>

**Adjusted output:**  

|              | 0 0010111011 | WO WFO            |

---

**x<1:**  
**Binary representation**  

<table>
<thead>
<tr>
<th>Input</th>
<th>0000 0000011</th>
<th><strong>Bitwidth (4,8)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix point</td>
<td></td>
<td>WM WF</td>
</tr>
</tbody>
</table>

**Normalized input:**  

<table>
<thead>
<tr>
<th></th>
<th>1 1000000000</th>
<th>1 WM+WF-1</th>
</tr>
</thead>
</table>

**Normalized output:**  

<table>
<thead>
<tr>
<th></th>
<th>0 1101000100</th>
<th>1 WM+WF-1</th>
</tr>
</thead>
</table>

**Highest precision output:**  

<table>
<thead>
<tr>
<th></th>
<th>0001 001110</th>
<th>1+(WF/2) WL-1-(WF/2)</th>
</tr>
</thead>
</table>

**Adjusted output:**  

<table>
<thead>
<tr>
<th></th>
<th>01001 01</th>
<th>WIO WFO</th>
</tr>
</thead>
</table>
\[
\begin{align*}
\text{X} \geq 1: & \quad \text{Binary representation} & \quad \text{Bitwidth (8,4)} \\
\text{Input:} & \quad 01110111 & 0100 \quad \text{Radix point} & \quad \text{Radix point} \\
\text{Normalized input:} & \quad 11011101000 \quad 1 \quad \text{W} + \text{W} - 1 \\
\text{Normalized output:} & \quad 01011101100 \quad 1 \quad \text{W} + \text{W} - 1 \\
\text{Highest precision output:} & \quad 01010 \quad 1101110 \quad 1 + \text{W}/2 \quad \text{W} - 1 - \text{W}/2 \\
\text{Adjusted output:} & \quad 01010 \quad 1101110 \quad \text{W}0 \quad \text{WFO} \\
\text{X} < 1: & \quad \text{Binary representation} & \quad \text{Bitwidth (4,8)} \\
\text{Input:} & \quad 0000 \quad 00000011 \quad \text{Radix point} & \quad \text{Radix point} \\
\text{Normalized input:} & \quad 1000000000 \quad 1 \quad \text{W} + \text{W} - 1 \\
\text{Normalized output:} & \quad 11010001000 \quad 1 \quad \text{W} + \text{W} - 1 \\
\text{Highest precision output:} & \quad 000110111011011 \quad 1 \quad \text{W} / 2 + \text{W} - 1 \\
\text{Adjusted output:} & \quad 00011011101110 \quad \text{W}0 \quad \text{WFO}
\end{align*}
\]

Figure 3.4. Bitwidth selection for the square root operation.
reciprocal square root operations depends on the integer bitwidth. The normalized operation block will ensure $WL$ bits precision, but this $WL$ bits could be shifted to either the integer part or the fractional part based on the shifting direction. For the reciprocal operation, when $x > 1$ and $x[WL-2]=1$, the result of the reciprocal will have at least $WI-2$ zeros leading the fractional part, and the normalized operation block ensures $WL$ bits precision after the $WI-2$ zeros. Thus the highest precision for this case is $2^{-(WI-2+WL-1)}$ as Fig. 3.2 shows. The same rationale can be applied to square root and reciprocal square root. For reciprocal square root, the number of leading zeros of fractional part is half of the integer bitwidth, thus the highest precision of the final results is $2^{-(WF/2+WL-1)}$ as shown in Fig. 3.3. For the square root operation when $x > 1$, the $WL$ bits ensures $WF/2+1$ bits for the integer part and $WL-WF/2-1$ bits for the fractional part.

When $x < 1$, for the reciprocal operation, the highest precision of the result is $2^{-WI}$ in the $(WF, WI)$ format, for the reciprocal square root operation the highest precision of the result is $2^{-(WF/2+WL-1)}$ in the $(1+WF/2, WL-1-(WF/2))$ format, and for the square root operation, the highest precision of the result is $2^{-(WF/2+WL-1)}$ in the $(1, WF/2+WL-1)$ format. Note that for reciprocal operation and reciprocal square root operation, the result’s fractional bitwidth depends on the input’s integer bitwidth while the result’s integer bitwidth depends on the input fractional bitwidth. The user can define the output bitwidth with arbitrary precisions. Hence, the default output precision can be adjusted based on use’s specifications.

### 3.1.3 Hardware Implementation Results

All three designs in our arbitrary-precision floating-point library are written in Verilog HDL. Every module described by HDL accepts the parameters $WI$, $WF$, $WIO$, $WFO$, $LUT_bits$, and $LUT_addWidth$, where $WI$ denotes the bitwidth of the input integer part, $WF$ denotes the bitwidth of the fractional part, $WIO$ denotes the bitwidth of the output integer part, $WFO$ denotes the bitwidth of the output fractional part, $LUT_addWidth$ denotes the bitwidth of the LUT’s address bus, and $LUT_bits$ denotes the bitwidth of LUT’s entries. The designs will be synthesized based on the user-defined parameters to create customized datapaths. Fig. 3.5

![Figure 3.5. Fixed-point arbitrary-precision module.](image)
shows the top-level input/output signals and parameters for the elementary functions calculations in fixed-point format with arbitrary precisions. Each module in the library has a CE signal and a respectively nRST signal. The nRST signal is an active-low reset signal for the registers. When CE is high, the module starts processing the input data. The din and dout are data input and data output signals. The design also supports exception handling. We synthesized the datapaths for reciprocal, square root, and reciprocal square root using different bitwidths, 8 bits, 10 bits, 12 bits, 16 bits, 20 bits, and 24 bits. Table 3.1, 3.2, and 3.3 present the detailed implementation results for each operations, respectively.

<table>
<thead>
<tr>
<th>$k$</th>
<th>8 bits</th>
<th>10 bits</th>
<th>12 bits</th>
<th>16 bits</th>
<th>20 bits</th>
<th>24 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT size</td>
<td>$2^4 \times 8$</td>
<td>$2^5 \times 10$</td>
<td>$2^6 \times 13$</td>
<td>$2^8 \times 16$</td>
<td>$2^{10} \times 21$</td>
<td>$2^{12} \times 24$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(8, 7)</td>
<td>(10, 9)</td>
<td>(13, 12)</td>
<td>(16, 15)</td>
<td>(21, 20)</td>
<td>(24, 23)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(8, 4)</td>
<td>(10, 5)</td>
<td>(12, 6)</td>
<td>(16, 8)</td>
<td>(20, 10)</td>
<td>(24, 12)</td>
</tr>
<tr>
<td>Latency</td>
<td>$\tau_m + \tau_a$</td>
<td>$\tau_m + \tau_a$</td>
<td>$\tau_m + \tau_a$</td>
<td>$\tau_m + \tau_a$</td>
<td>$\tau_m + \tau_a$</td>
<td>$\tau_m + \tau_a$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>277</td>
<td>277</td>
<td>277</td>
<td>201</td>
<td>201</td>
<td>201</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>101</td>
<td>141</td>
<td>209</td>
<td>247</td>
<td>329</td>
<td>415</td>
</tr>
<tr>
<td>DSP48</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.1. Implementation characteristics of the fixed-point reciprocal with arbitrary precision

<table>
<thead>
<tr>
<th>$k$</th>
<th>8 bits</th>
<th>10 bits</th>
<th>12 bits</th>
<th>16 bits</th>
<th>20 bits</th>
<th>24 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT size</td>
<td>$2^4 \times 10$</td>
<td>$2^5 \times 12$</td>
<td>$2^6 \times 13$</td>
<td>$2^8 \times 16$</td>
<td>$2^{10} \times 23$</td>
<td>$2^{12} \times 27$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(8, 4)</td>
<td>(10, 5)</td>
<td>(12, 6)</td>
<td>(16, 8)</td>
<td>(20, 10)</td>
<td>(24, 12)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(11, 10)</td>
<td>(13, 12)</td>
<td>(15, 14)</td>
<td>(19, 18)</td>
<td>(23, 22)</td>
<td>(27, 26)</td>
</tr>
<tr>
<td>Latency</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>122</td>
<td>121</td>
<td>119</td>
<td>118</td>
<td>93</td>
<td>92</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>138</td>
<td>218</td>
<td>279</td>
<td>318</td>
<td>414</td>
<td>512</td>
</tr>
<tr>
<td>DSP48</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3.2. Implementation characteristics of the fixed-point square root operation with arbitrary precision
Table 3.3. Implementation characteristics of the fixed-point reciprocal square root with arbitrary precision

<table>
<thead>
<tr>
<th>$k$</th>
<th>8 bits</th>
<th>10 bits</th>
<th>12 bits</th>
<th>16 bits</th>
<th>20 bits</th>
<th>24 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT size</td>
<td>$2^4 \times 8$</td>
<td>$2^5 \times 10$</td>
<td>$2^6 \times 12$</td>
<td>$2^8 \times 16$</td>
<td>$2^{10} \times 23$</td>
<td>$2^{12} \times 24$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(8, 7)</td>
<td>(10, 9)</td>
<td>(12, 11)</td>
<td>(16, 15)</td>
<td>(23, 22)</td>
<td>(24, 23)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(8, 4)</td>
<td>(10, 5)</td>
<td>(12, 6)</td>
<td>(16, 8)</td>
<td>(20, 10)</td>
<td>(24, 12)</td>
</tr>
<tr>
<td>Latency</td>
<td>$4\tau_m + \tau_a$</td>
<td>$4\tau_m + \tau_a$</td>
<td>$4\tau_m + \tau_a$</td>
<td>$4\tau_m + \tau_a$</td>
<td>$4\tau_m + \tau_a$</td>
<td>$4\tau_m + \tau_a$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>210</td>
<td>205</td>
<td>204</td>
<td>195</td>
<td>194</td>
<td>162</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>140</td>
<td>226</td>
<td>279</td>
<td>326</td>
<td>423</td>
<td>529</td>
</tr>
<tr>
<td>DSP48</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

3.2 Validation of Hardware Implementations

The MATLAB modules for calculating the three elementary functions with arbitrary precisions are created first based on the algorithms’ characteristics. The MATLAB modules also generate the initialization LUTs based on the parameter settings. After verifying the MATLAB models using exhaustive bit-true simulations, the algorithms are implemented on an FPGA.

We performed exhaustive bit-true simulations for 8, 10, 12, 16, 20 bits wordlengths. They all use the same error checking schemes as the one used for the single-precision described in the Chapter 2. Also, the optimized datapath bitwidths and LUT sizes can be obtained via bit-true simulations.

After exhaustive error verification of MATLAB modules, the content of LUTs are generated and stored into memory block on an FPGA at initialization. The validating process is shown in Fig. 3.6.
3.3 SUMMARY

This chapter presented the fixed-point calculation of reciprocal, square root, and reciprocal square root supporting arbitrary precisions. The implementations on a Xilinx Virtex-7 FPGA with 8, 10, 12, 16, 20, and 24 bits wordlengths are presented. Using exhaustive bit-true simulations, the datapaths parameters, such as the size of the LUTs and precision of intermediate signals, were obtained. The hardware implementation results were verified using the bit-true MATLAB simulation results. Resource requirements, clock frequency, and the latency of different designs were presented and compared.
CHAPTER 4
ARBITRARY-PRECISION SCHEMES FOR FLOATING-POINT RECIPROCAL, SQUARE ROOT, AND RECIPROCAL SQUARE ROOT

This chapter presents the design and implementation of reciprocal, square root, and reciprocal square root in floating-point representation supporting arbitrary precisions. We explore the design and implementation of reciprocal, square root, and reciprocal square root in floating-point format with an arbitrary precision. Table 4.1 gives the specifications of the two IEEE formats, i.e., 32-bit single-precision and 64-bit double-precision, along with the arbitrary-precision format.

<table>
<thead>
<tr>
<th>Format</th>
<th>Sign</th>
<th>Biased Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-precision</td>
<td>1 bit</td>
<td>8 bit</td>
<td>23 bit</td>
</tr>
<tr>
<td>Double-precision</td>
<td>1 bit</td>
<td>11 bit</td>
<td>52 bit</td>
</tr>
<tr>
<td>Arbitrary precision</td>
<td>1 bit</td>
<td>variable</td>
<td>variable</td>
</tr>
</tbody>
</table>

Table 4.1. Various floating-point formats

4.1 ARBITRARY-PRECISION FLOATING-POINT LIBRARY DESIGN

Fig. 4.1 shows the block diagram of the arbitrary-precision floating-point operations of interest. There are three key components of the floating-point operations that should be considered when using the normalized single-precision calculation block: sign, biased exponent, and mantissa.

The one bit sign of the results for the three operations reciprocal, square root and reciprocal square root are the same as the inputs. A flag $\text{sign}$ will be used to store the sign of input $x$. Fraction and exponent will be obtained from the input by bit selection. The exponent can be obtained from the second most significant bit (MSB) to the $W_e + 1$-th bit of the input where $W_e$ is the bitwidth of the exponent field. The fraction can be selected from the right most $W_f$ bits of the input where $W_f$ is the bitwidth of fraction. The implied bit one will be appended to the MSB of the fraction to generate the $x$, which is the input to the normalized calculation block. After calculating $f(x)$ for $x \in [1, 2)$, the output $y_{\text{rec}}, y_{\text{sqrt}}$, and $y_{\text{isqrt}}$ will be within $[0.5, 1), [1, \sqrt{2})$, and $[1, \frac{1}{\sqrt{2}})$ for reciprocal, square root, and reciprocal square root,
Figure 4.1. Implementation of the three elementary functions in floating-point format supporting arbitrary precisions.

respectively. Then, the calculated normalized result of the fraction and exponent will be passed into the \textit{collaborated norm} module. The \textit{collaborated norm} block operates differently for each of the three operations.

Algorithm 5 presents the steps for reciprocal calculation of \( x \) in floating-point format with arbitrary precisions. In the floating-point format, the output’s exponent of the reciprocal operation can be obtained by inverting the sign of the exponent. Hence:

\[
x = (-1)^s x_0 2^{E-B}
\]  

(4.1)

\[
\text{out} = \frac{1}{(-1)^s x_0 2^{E-B}} = (-1)^{-s} y_{\text{rec}} 2^{-(E-B)}
\]  

(4.2)

Due to the adoption of biasing, the exponent is signed and symmetrical. Thus the result of the exponent can be calculated by mirroring to the bias, as shown by the Fig. 4.2. Notice that the result from the normalized calculation module \( y_{\text{rec}} \in [0.5, 1) \). Thus, to generate the fraction of the final result, the normalization process should be done by shifting \( y_{\text{rec}} \) one bit to the left, and also the exponent should minus one correspondently to finish the collaborated normalization process.

Algorithm 6 presents the steps for square root calculation of \( x \) in floating-point format with arbitrary precisions. In the floating-point format, the output’s exponent of the square root
Figure 4.2. Exponent result of reciprocal.

Operation is half of the exponent of the input. Hence:

\[ out = \sqrt{(-1)^s x_0 2^{E-B}} = (-1)^{E_{\text{sqrt}}} 2^{E_{\text{sqrt}} - B} \]  

(4.3)

Notice that, when the difference between exponent and bias \( E - B \) is even, the output exponent is half of the input exponent as shown in Fig. 4.3, and the result from the square root normalized calculation module is also normalized \( y_{\text{sqrt}} \in [1, \sqrt{2}) \). Thus, there is no need to normalize it. However, when the difference between exponent and bias \( E - B \) is odd, a scaling factor \( \sqrt{2} \) should be multiplied with the normalized fraction output, and the exponent should be subtracted by one. Hence:

\[ out = (-1)^{\frac{1}{2}} y_{\text{sqrt}} \times \sqrt{2} \times 2^{E_{\text{sqrt}} - B - \frac{3}{2}} \]  

(4.4)

Algorithm 7 presents the steps for reciprocal square root calculation of \( x \) in floating-point format with arbitrary precision. In the floating-point format, the output’s
Algorithm 5 Reciprocal algorithm for floating-point inputs with arbitrary precisions

\[ W_e = \text{bitwidth of exponent}; \]
\[ 2: \quad W_f = \text{bitwidth of mantissa}; \]
\[ WL = W_e + W_f + 1 \text{ is the wordlength of input } x; \]
\[ 4: \quad E = x[W_L - 2 : W_L - 1 - W_e]; \]
\[ B = 2^{W_e - 1} - 1; \]
\[ 6: \quad m = \{1'b1, x[W_m - 1 : 0]\}; \]
\[ y_{rec} = \text{reciprocal-normalized}(m); \]
\[ 8: \quad m_o = y_{rec} \ll 1; \]

Adjust and produce the exponent:
\[ 10: \quad \text{if } E < B \text{ then} \]
\[ \quad E_o = B + (B - E) - 1; \]
\[ 12: \quad \text{else} \]
\[ \quad E_o = B - (E - B) - 1; \]
\[ 14: \quad \text{end if} \]
\[ \text{result} = \{\text{sign}, E_o, m_o\}; \]

exponent of the reciprocal square root operation is half of the input’s exponent with the inverted sign. Hence:
\[ \text{out} = \frac{1}{\sqrt{(-1)^s x_0^2 E - B}} = (-1)^{-\frac{s}{2}} y_{isqrt} 2^{-\frac{(E-B)}{2}} \]  

(4.5)

Notice that, when the difference between exponent and bias \( E - B \) is even, the output exponent is half of the input exponent with inverted sign as shown in Fig. 4.4, and the result from the square root normalized calculation module is \( y_{isqrt} \in [1, \frac{1}{\sqrt{2}}) \). To normalize the result, the fraction \( y_{isqrt} \) is shifted one bit left, and minus one from the exponent. When the

\[ E_o = \frac{E}{2} \]
\[ B \]

\[ \text{Normal:} \quad \begin{array}{c}
\text{Val}(E_o) = \frac{E}{2} \\
\text{Val}(E) = E
\end{array} \]

\[ \begin{array}{c}
\text{Biased:} \quad \begin{array}{c}
\text{Val}(E_o) = \frac{E - B}{2} \\
\text{Val}(E) = E - B
\end{array}
\end{array} \]

Figure 4.4. Exponent result of reciprocal square root.
Algorithm 6 Square root algorithm for floating-point inputs with arbitrary precisions

\[ y_{sqrt} = square\_root\_normalized(m); \]

2: \[ y_{sqrt} = y_{sqrt} \times \sqrt{2}; \]

Adjust collaboratively and produce the exponent and mantissa:

4: if \( E < B \) then

\[ diff = B - E; \]

6: \[ \text{if } (diff \mod 2) == 0 \text{ then} \]

\[ E_o = B - \frac{B-E}{2}; \]

8: \[ m_o = y_{sqrt}[W_m - 1 : 0]; \]

else

10: \[ E_o = B - \frac{B-E}{2} - 1; \]

12: \[ m_o = y_{sqrt}[W_m - 1 : 0]; \]

else

14: \[ diff = E - B; \]

16: \[ \text{if } (diff \mod 2) == 0 \text{ then} \]

18: \[ m_o = y_{sqrt}[W_m - 1 : 0]; \]

else

20: \[ m_o = y_{sqrt}[W_m - 1 : 0]; \]

end if

22: end if

Combine the \textit{sign}, \( E_o \) and \( m_o \) as the output:

24: \[ \text{result} = \{ \text{sign}, E_o, m_o \}; \]
difference between exponent and bias $E - B$ is odd, a scaling factor $\sqrt{2}$ should be multiplied with the normalized mantissa output, the exponent should add one to compensate as shown in Equation (4.6):

$$out = (-1)^{\frac{E}{2}} y_{\text{isqrt}} \times \sqrt{2} \times 2^{-\frac{(E-B+1)}{2}}$$  (4.6)

**Algorithm 7** Reciprocal square root algorithm for floating-point input with arbitrary precisions

1. $y_{\text{isqrt}} = \text{reciprocal\_square\_root}(m)$;
2. $y_{\text{isqrt}} = y_{\text{isqrt}} \times \sqrt{2}$;
   
   Adjust coolaboratly and produce the exponent and mantissa:
3. if $E < B$ then
   
   $diff = B - E$;
4. if $(diff \mod 2) == 0$ then
   
   $E_o = B + \frac{B-E}{2} - 1$;
5. $m_o = y_{\text{isqrt}}[W_m - 1 : 0]$;
6. else
   
   $E_o = B + \frac{B-E}{2} + 1$;
7. $m_o = y_{\text{isqrt}}[W_m : 1]$;
8. end if
9. else
   
   $diff = E - B$;
10. if $(diff \mod 2) == 0$ then
   
   $E_o = B - \frac{E-B}{2} - 1$;
11. $m_o = y_{\text{isqrt}}[W_m - 1 : 0]$;
12. else
   
   $E_o = B - \frac{E-B}{2} + 1$;
13. $m_o = y_{\text{isqrt}}[W_m : 1]$;
14. end if
15. end if
16. $result = \{\text{sign}, E_o, m_o\}$;

### 4.2 Results of the Hardware Implementation

All three designs in our arbitrary-precision floating-point library are written in Verilog HDL. Every module accepts the parameters $W_e, W_f, LUT\_add\_Width$, and $LUT\_bits$. The
$W_e$ is the bitwidth of the exponent part, $W_f$ is the bitwidth of the fraction part, $LUT\_addWidth$ is the bitwidth of the LUT address length, and $LUT\_bits$ is the bitwidth of each element in the LUT. Fig.4.5 presents the top-level block diagram of calculating

![Block Diagram](image)

**Figure 4.5. Floating-point module with arbitrary precision.**

elementary functions over floating-point inputs with arbitrary precisions. Each module in the library has a $CE$ signal and a $nRST$ signal. The $nRST$ signal is an active-low reset for the registers. When $CE$ signal is high, the module processes the input data. The $OP$ and $result$ are data input (operand) and data output (result), respectively. We synthesized the three designs for calculating reciprocal, square root, and reciprocal square root for floating-point inputs with three different bitwidths, 16, 24, and 32 bits. Table 4.2, 4.3 and 4.4 present the synthesis results for these three operations on a Xilinx Virtex-7 XC7V2000T FPGA at a speed grade of -2.

<table>
<thead>
<tr>
<th></th>
<th>16 bits (4,11)</th>
<th>24 bits (6,17)</th>
<th>32 bits (8,23)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>6</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>LUT size</td>
<td>$2^6 \times 13$</td>
<td>$2^9 \times 19$</td>
<td>$2^{12} \times 15$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>(13, 12)</td>
<td>(19, 18)</td>
<td>(24, 23)</td>
</tr>
<tr>
<td>Datapath format</td>
<td>(15, 14)</td>
<td>(21, 20)</td>
<td>(29, 27)</td>
</tr>
<tr>
<td>Latency</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>267</td>
<td>222</td>
<td>201</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>64</td>
<td>112</td>
<td>131</td>
</tr>
<tr>
<td>DSP48</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 4.2. Implementation characteristics of reciprocal for floating-point inputs format with arbitrary precisions.**

### 4.3 Library Validation and Using Method

For validating the algorithms and hardware implementations of the elementary functions for floating-point with arbitrary precision, exhaustive simulation of operations over normalized inputs with different bitwidths are performed and the absolute errors were
measured. The values of LUTs were generated based on the target precision. Users can choose any algorithms operating on normalized input $x \in [1, 2)$ described in Chapter 2 to implement the operations over inputs in floating-point format with arbitrary precisions. After verifying the algorithms via bit-true exhaustive simulations, hardware description of the algorithms were developed. The FPGA implementation results were compared with the bit-true MATLAB simulation results.

### 4.4 SUMMARY

This chapter presented the procedure and hardware architectures for calculating reciprocal, square root, and reciprocal square-root in floating-point format with arbitrary precisions. These procedures utilize the approaches used for calculating $\sqrt{x}$, $\frac{1}{x}$, and $\frac{1}{\sqrt{x}}$, where $x$ is a normalized number within [1,2), to implement these functions over $x$ with an arbitrary precisions. The FPGA implementation results show that the calculation of these three elementary functions in floating-point format with arbitrary precision requires relatively the same hardware resources and operates at approximately similar frequencies as the operations over the fixed-point format with arbitrary precisions.
CHAPTER 5
ARBITRARY-PRECISION SCHEMES FOR DIVISION

This chapter presents the design and implementation of division over fixed-point and floating-point formats with arbitrary precisions. We compare our implementation results in various bitwidths with other published results.

5.1 ARITHMETIC DESIGN

The divider in fixed-point and floating-point format follows the same rationale. Utilizing the reciprocal module, we are able to implement for division as follows:

\[
\text{quotient} = \text{dividend} \div \text{divisor} = \text{dividend} \times \frac{1}{\text{divisor}}
\]  

(5.1)

Therefore, for the floating-point format, we need to design the multiplier with arbitrary-precision. Fig. 5.1 presents the floating-point multiplier block diagram [35].

The following steps should be followed for the floating-point multiplication:

- First, add two operands’ exponents together: \( E_o = E_1 + E_2 \).
- This addition will add the bias in the exponents twice, thus, a subtraction of the bias from the sum needs to be applied.
- Then add the implied ‘1’ to the MSBs of the fractions.
- Multiply the two unsigned fractions to generate the product.
- Normalize the product.
- Post-process of the mantissa: Use rounding or truncation to fit the product into \( WFO \) bits, where \( WFO \) is the bitwidth of the output’s fraction.
- Exclusive OR the signs of the operands to generate the result’s sign \( \text{sign} \).
- \( y_{\text{div}} = \{ \text{sign}, E_o, f_o \} \)

As the rounding process will affect the units of shifting in the normalization process, the rounding module should be designed with the normalization module together. We use round-to-nearest scheme for the rounding, which is the default rounding mode in the IEEE 754 standard. The details of round-to-nearest scheme is shown in Algorithm 8. Also, our arbitrary-precision floating-point multiplier supports exception handling. It has exception in \( \text{exce}_{\text{in}} \) signal and exception out \( \text{exce}_{\text{out}} \) signal. The \( \text{exce}_{\text{in}} \) is 1 if any of the inputs are invalid, and \( \text{exce}_{\text{out}} \) is asserted to notify an invalid output. This might be caused by an
overflow or an underflow when calculating the exponent or executing the normalization process.

The arbitrary-precision floating-point divider can be implemented using the arbitrary-precision floating-point reciprocal and a multiplier. Fig. 5.2 presents the divider block diagram. One of the operands (divisor) $OP_2$ is passed into the arbitrary-precision floating-point reciprocal block $flotRec$ with the exception control signal $exce_{in}$. The results of the reciprocal block $result$, $exce_{out}$, and the dividend are passed into the floating-point multiplier unit $flotMul$. The output of the multiplier is the result of the division.

Similarly, we modeled and implemented the arbitrary-precision fixed-point divider using arbitrary-precision reciprocal and a fixed-point multiplier as shown in Fig. 5.3 presents. Notice that there is no need for the exception control signal for the fixed-point format design. This is because we have the specific protection mechanism in the datapath to avoid the overflow and underflow. If the result is out of the upper bound, the output will saturate at the upper level of the bound and if the result is less than the lower bound, the output will saturate at the lower level of the bound.

### 5.2 Hardware Implementation Results

The designs of arbitrary-precision floating-point and fixed-point dividers are written in Verilog HDL and are synthesized using Xilinx ISE 14.7. Fig. 5.4 shows the block diagram of the floating-point divider. The floating-point module receives $OP_1$, $OP_2$ as the two input operands, $nRST$ as a reset signal, $CE$ as a clock enable signal, $CLK$ as a system clock, $exce_{in}$ as an input exception notification signal, and produces $result$ as the result of the module and
Algorithm 8 Round-to-nearest process

Input $x$ is a $WL$-bit input including $RL$-bit for the rounding part;

2: $y$ is the output with with format \{WordLength-1:RoundLength\};

\begin{algorithmic}
\State \textbf{if} $x[RL-1] == 1'b1$ \textbf{then} \quad // If the first bit for rounding is ’1’
\State \hspace{1em} \textbf{if} $x[RL-2 : 0] > 0$ \textbf{then} \quad // If the following part is not equal to zero
\State \hspace{2em} $y <= x[WL - 1 : RL] + 1'b1$;
\State \hspace{1em} \textbf{else}
\State \hspace{2em} $y <= x[WL - 1 : RL]$;
\State \hspace{1em} \textbf{end if}
\State \textbf{else} \quad // If the first bit used for rounding is not ’1’
\State \hspace{1em} $y <= x[WL - 1 : RL]$;
\State \textbf{end if}
\end{algorithmic}

Figure 5.2. Arbitrary-precision floating-point block diagram for division.

the \textit{exce\textunderscore out} as the exception out notification signal. The module also accepts the same parameters as the arbitrary-precision floating-point reciprocal module $W_e$, $W_f$, $LUT\textunderscore addWidth$, and $LUT\textunderscore bits$. The implementations are synthesized on a Xilinx Virtex-7 XC7V2000T FPGA with the speed grade of -2. Table 5.1 presents the implementation
Figure 5.3. Arbitrary-precision fixed-point block diagram for division.

Figure 5.4. Arbitrary-precision fixed-point module for division.

characteristics of the divider for different wordlengths. Fig. 5.5 shows the top-level block diagram of the fixed-point divider. The fixed-point module receives \( OP1, OP2 \) as the two input operands, \( nRST \) as a negative reset signal, \( CE \) as a clock enable signal, \( CLK \) as the system clock, and produces \( \text{result} \) as the result of the module. The module also accepts the same parameters as the arbitrary-precision fixed-point modules introduced in Chapters 3 \( WI, WF, WIO, WFO, LUT\_addWidth, \) and \( LUT\_bits \). Table 5.2 presents implementation characteristics of the fixed-point divider for different wordlengths.
<table>
<thead>
<tr>
<th>$k$</th>
<th>16 bits (4,11)</th>
<th>24 bits (6,17)</th>
<th>32 bits (8,23)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT size</td>
<td>$2^6 \times 13$</td>
<td>$2^9 \times 19$</td>
<td>$2^{12} \times 24$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>$(13, 12)$</td>
<td>$(19, 18)$</td>
<td>$(24, 23)$</td>
</tr>
<tr>
<td>Datapath format</td>
<td>$(4, 11)$</td>
<td>$(6, 17)$</td>
<td>$(8, 23)$</td>
</tr>
<tr>
<td>Latency</td>
<td>$3\tau_m + \tau_a$</td>
<td>$3\tau_m + \tau_a$</td>
<td>$3\tau_m + \tau_a$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>267</td>
<td>222</td>
<td>201</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>111</td>
<td>199</td>
<td>221</td>
</tr>
<tr>
<td>DSP48</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.1. Implementation characteristics of division in floating-point format with arbitrary-precision

![Arbitrary-precision fixed-point module for division.](image)

**Figure 5.5.** Arbitrary-precision fixed-point module for division.

<table>
<thead>
<tr>
<th>$k$</th>
<th>8 bits</th>
<th>10 bits</th>
<th>12 bits</th>
<th>16 bits</th>
<th>20 bits</th>
<th>24 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT size</td>
<td>$2^4 \times 8$</td>
<td>$2^9 \times 10$</td>
<td>$2^6 \times 13$</td>
<td>$2^8 \times 16$</td>
<td>$2^{10} \times 21$</td>
<td>$2^{12} \times 24$</td>
</tr>
<tr>
<td>LUT entries format</td>
<td>$(8, 7)$</td>
<td>$(10, 9)$</td>
<td>$(13, 12)$</td>
<td>$(16, 15)$</td>
<td>$(21, 20)$</td>
<td>$(24, 23)$</td>
</tr>
<tr>
<td>Datapath format</td>
<td>$(8, 4)$</td>
<td>$(10, 5)$</td>
<td>$(12, 6)$</td>
<td>$(16, 8)$</td>
<td>$(20, 10)$</td>
<td>$(24, 12)$</td>
</tr>
<tr>
<td>Latency (ns)</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
<td>$2\tau_m + \tau_a$</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>277</td>
<td>277</td>
<td>277</td>
<td>276</td>
<td>201</td>
<td>201</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>153</td>
<td>220</td>
<td>287</td>
<td>386</td>
<td>506</td>
<td>583</td>
</tr>
<tr>
<td>DSP48</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.2. Implementation characteristics of the division in fixed-point format with arbitrary-precision

## 5.3 SUMMARY

This chapter presented the designs of division operation using multiplicative approach in fixed-point and floating-point formats with arbitrary precisions. The implementations on Xilinx Virtex-7 XC7V2000T FPGA at speed grade -2 with different datapath bitwidth are presented.
CHAPTER 6
OPTIMIZATION AND COMPARISON

This chapter presents different techniques for increasing the performance and decreasing the hardware resources of our designs.

6.1 Pipelining Technique

The maximum clock frequency $F_{\text{max}}$ depends on the minimum clock period $T$. The clock period is directly defined by the critical path delay $T_{\text{critical}}$, which can be described as follows:

$$F_{\text{max}} = \frac{1}{T}, \text{ where } T > T_{\text{critical}}$$

(6.1)

Thus, decreasing the critical path delay is the key to improve the maximal working frequency. Fig. 6.2 shows the critical path of the datapath for NR reciprocal.

Pipelining is a well-known technique for reducing the critical path delay by adding registers to the critical path of the datapath. Thus, pipelining is able to increase the clock rate and throughput by decreasing the critical path delay. While pipelining increases the performance and utilization of computational blocks, it increases the latency of the design. Latency can be written as:

$$\text{Latency} = T \times (\text{Number of Stages})$$

(6.2)
For example, in a two-stage pipelined design, the output will be generated every two clock cycles after the input is applied in. The datapaths with one iteration are feed-forward

![Datapath Diagram](image)

**Figure 6.2. The datapath for pipelined NR reciprocal (direct implementation).**

datapaths, which means that the data moves in the forward direction on all the paths. In [10], an optimized three-stage multiplier and adder were presented. The work in [8] presented a 12-stage pipelined implementation of the double-precision divider utilizing GS method on an FPGA. Fig. 6.2 shows the datapath of NR reciprocal that is pipelined using 6 cutsets. The design is for a Xilinx Virtex-7 XC7V2000T at speed grade of -2. The pipelined datapath for single-precision NR reciprocal can run at 368 MHz, which is more than two times faster than the performance of the original datapath (145 MHz). For the arbitrary-precision floating-point

![Datapath Diagram](image)

**Figure 6.3. The datapath for the pipelined arbitrary-precision floating-point mutiplier.**
multiplier, we can pipeline it into six stages as shown in Fig. 6.3. The delay of the arbitrary-precision fixed-point multiplier is about two times of the delay of the adder and the delay of the Normalization & Rounding module is about 3 times of the adder’s delay. Thus, the fixed-point multiplier is pipelined with two stages and the Normalization & Rounding module is pipelined with three stages.

![Figure 6.4. The datapath for the pipelined normalized square root.](image)

We also pipelined the normalized square root module discussed in the Chapter 2. Fig. 6.4 presents the pipelined datapath for this module. Pipelining the Hung’s datapath using 5 stages allows the normalized square root module to run at 368 MHz on a Xilinx Virtex-7 FPGA instead of 92 MHz using the unpipelined datapath.

Fig. 6.5 shows the block diagram of the pipelined elementary function calculation in arbitrary-precision floating-point format.

### 6.2 Small LUT and Multiplier Technique

One disadvantage of the traditional multiplicative approaches for reciprocal and square root calculations, such as NR, GS and Hung’s method is that the size of the LUT increases exponentially with the bitwidth of the input. Although the performance of using these algorithms is relatively high, it’s not practical to use them in high-precision calculations, such as in double-precision reciprocal, square root, or division as the size of LUTs would be very large. Therefore, for high-precision calculations, we should use techniques with relatively small LUTs and multipliers.

The work in [31] presents the comparison between different approaches using small multipliers and LUTs. The results show that Ito’s method has the minimum total delay while utilizing a relatively small LUT. As we discussed in the Chapter 2, Ito’s approach uses linear initial approximation and accelerated convergence methods. Therefore, when the input
operand’s bitwidth is relatively large or when the required precision is relatively high, the LUT used in Ito’s method will be significantly smaller than those used in NR, GS, or Hung’s method. For the reciprocal operation, the work in [26] enhances the performance of the NR approach by modifying the recurrence so that additional accuracy is achieved by each multiply-accumulate and by adding a cubic term. The work in [31] uses 56-bit inputs, which is also the bitwidth of the LUT for initial approximations. Three $56 \times 56 - \text{bit}$ multipliers are used. However, using our exhaustive error checking, as described in Algorithm 1, we found that we can use only one $27 \times 53 - \text{bit}$ multiplier and two $30 \times 56 - \text{bit}$ multipliers instead of using the three $56 \times 56 - \text{bit}$ multipliers. Fig. 6.6 shows the absolute error plots of the reciprocal utilizing Ito’s approximation for double-precision accuracy with the LUT and multipliers mentioned above.

While Ito’s approach uses three multipliers compared to two and only one used in NR and Hung’s approaches, respectively, we can use pipelining to improve the throughput.
6.3 Unfolding Technique

Unfolding is a transformation technique that can be applied to a DSP program to create a new scheme describing more than one iteration of the scheme. Hence, unfolding is also called loop unrolling and has been widely used by many compilers for reducing the loop overhead in code written in high-level languages.

We can apply the same rationale of software unfolding technique into our iterative hardware designs to improve the throughput and decrease the size of LUTs. For example, Fig. 6.7 presents the datapath for NR reciprocal utilizing Ito’s initial approximation approach.
Utilizing more than one iteration enable us to use a even smaller LUT, but the throughput decreases. In order to achieve higher throughput, we can use the unfolding technique to transfer the iteration into direct implementation, as shown in Fig. 6.8.

Figure 6.8. The datapath for unfolded NR reciprocal utilizing Ito’s initial approximation approach.

Exhaustive error checking shows that to achieve double-precision accuracy, using Ito’s initial approximation with one iteration requires $2^{13} \times 27 = 221184$-bit LUT, but only requires $2^6 \times 15 = 960$-bit LUT if using two iterations. Fig. 6.9 shows the absolute error plots of the reciprocal utilizing Ito’s approximation for double-precision accuracy with two iterations. Notice that unfolding the datapath does not affect the maximum frequency. Because the multiplier of the last stage has the same size of the folded datapath’s, which leads the largest combinational delay of the datapath remains the same.
In this section, we will compare our optimized designs with the original designs presented in the previous chapters and the latest published designs in both single-precision and double-precision floating-point formats. We choose to compare our implementation results with Xilinx’s floating-point IP core [36], Altera’s floating-point IP core [37], the PolyGS algorithm [8], the FloPoCo algorithm [38], and the VFLOAT library from Northeastern University [13], [14] and [39].

Our designs are written in Verilog HDL and synthesized using Xilinx ISE 14.7 on a Virtex-7 XC7V2000T FPGA and a Virtex-6 XC6VLX75T FPGA.

Table 6.1 presents the implementation results of different work for single-precision reciprocal on a Virtex-7 FPGA. The results show that our pipelined designs are the fastest among others. Our implementation utilizing Hung’s method is faster than the implementation utilizing Ito’s method. However, Ito’s method only requires \(2^6 \times 13 = 832\) bits LUT, which is much smaller than the LUT used in Hung’s method \((2^{12} \times 26 = 106496\) bits). The results in Table 6.2 show that our optimized reciprocal utilizing Ito’s method are the fastest among others with double-precision format also.

Table 6.3 presents the implementation results of 32-bit square root operation. With pipelining, the frequency increases about two times of the original design as shown in Table 2.5. The results show that our implementation is faster than the Xilinx IP core with the same number of pipeline stages. Notice that our implementation without DSP48 is faster than the one with DSP48, while in Table 6.1, the implementation with DSP48 is faster. This is because

---

**Figure 6.9.** The reciprocal error plot utilizing Ito’s approach for double-precision accuracy with two iterations.

6.4 **IMPLEMENTATION RESULTS AND COMPARISON**

In this section, we will compare our optimized designs with the original designs presented in the previous chapters and the latest published designs in both single-precision and double-precision floating-point formats. We choose to compare our implementation results with Xilinx’s floating-point IP core [36], Altera’s floating-point IP core [37], the PolyGS algorithm [8], the FloPoCo algorithm [38], and the VFLOAT library from Northeastern University [13], [14] and [39].

Our designs are written in Verilog HDL and synthesized using Xilinx ISE 14.7 on a Virtex-7 XC7V2000T FPGA and a Virtex-6 XC6VLX75T FPGA.

Table 6.1 presents the implementation results of different work for single-precision reciprocal on a Virtex-7 FPGA. The results show that our pipelined designs are the fastest among others. Our implementation utilizing Hung’s method is faster than the implementation utilizing Ito’s method. However, Ito’s method only requires \(2^6 \times 13 = 832\) bits LUT, which is much smaller than the LUT used in Hung’s method \((2^{12} \times 26 = 106496\) bits). The results in Table 6.2 show that our optimized reciprocal utilizing Ito’s method are the fastest among others with double-precision format also.

Table 6.3 presents the implementation results of 32-bit square root operation. With pipelining, the frequency increases about two times of the original design as shown in Table 2.5. The results show that our implementation is faster than the Xilinx IP core with the same number of pipeline stages. Notice that our implementation without DSP48 is faster than the one with DSP48, while in Table 6.1, the implementation with DSP48 is faster. This is because
Table 6.1. Implementation characteristics of different single-precision reciprocal realizations.

<table>
<thead>
<tr>
<th>Device</th>
<th>Ours1 (Hung)</th>
<th>Ours2 (Ito)</th>
<th>Ours3 (Ito unfold)</th>
<th>Xilinx IP [36]</th>
<th>Xilinx IP2 [36]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency (MHz)</td>
<td>606</td>
<td>591</td>
<td>310</td>
<td>191</td>
<td>473</td>
</tr>
<tr>
<td>Latency (Clock cycles)</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td>Slice registers</td>
<td>162 (&lt; 1%)</td>
<td>295 (&lt; 1%)</td>
<td>230 (&lt; 1%)</td>
<td>174 (&lt; 1%)</td>
<td>256 (&lt; 1%)</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>133 (&lt; 1%)</td>
<td>215 (&lt; 1%)</td>
<td>199 (&lt; 1%)</td>
<td>147 (&lt; 1%)</td>
<td>228 (&lt; 1%)</td>
</tr>
<tr>
<td>DSP48</td>
<td>2 (&lt; 1%)</td>
<td>3 (&lt; 1%)</td>
<td>6 (&lt; 1%)</td>
<td>8 (&lt; 1%)</td>
<td>8 (&lt; 1%)</td>
</tr>
</tbody>
</table>

Table 6.2. Implementation characteristics of different double-precision reciprocal realizations.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency (MHz)</td>
<td>126</td>
<td>145</td>
<td>111</td>
<td>438</td>
<td>203</td>
</tr>
<tr>
<td>Latency (Clock cycles)</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>36</td>
<td>27</td>
</tr>
<tr>
<td>Slice registers</td>
<td>550 (&lt; 1%)</td>
<td>809 (&lt; 1%)</td>
<td>344 (&lt; 1%)</td>
<td>657 (&lt; 1%)</td>
<td>N/A</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>542 (&lt; 1%)</td>
<td>5583 (&lt; 1%)</td>
<td>296 (&lt; 1%)</td>
<td>314 (&lt; 1%)</td>
<td>N/A</td>
</tr>
<tr>
<td>DSP48</td>
<td>18 (&lt; 1%)</td>
<td>0</td>
<td>14 (&lt; 1%)</td>
<td>14 (&lt; 1%)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6.3. Implementation characteristics of different single-precision square root realizations.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency (MHz)</td>
<td>210</td>
<td>543</td>
<td>151</td>
<td>521</td>
<td>472</td>
</tr>
<tr>
<td>Latency (Clock cycles)</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>25</td>
<td>28</td>
</tr>
<tr>
<td>Slice registers</td>
<td>222 (&lt; 1%)</td>
<td>989 (1%)</td>
<td>360 (&lt; 1%)</td>
<td>865 (1%)</td>
<td>N/A</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>232 (&lt; 1%)</td>
<td>87 (&lt; 1%)</td>
<td>546 (1%)</td>
<td>525 (1%)</td>
<td>N/A</td>
</tr>
<tr>
<td>DSP48</td>
<td>6 (&lt; 1%)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6.4 presents the implementation results of different double-precision square root modules. To compare with the VFLOAT library, we synthesized our designs on a Virtex-6 XC6VLX75T FPGA. Notice from the result, our design can work faster than the VFLOAT library and Xilinx IP core using the same number of pipeline stages at the cost of larger area.

Table 6.5 presents the implementation results of different single-precision division modules. The latency of our pipelined division is 14 clock cycles. The results show that both
Table 6.4. Implementation characteristics of different double-precision square root realizations.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency (MHz)</td>
<td>155</td>
<td>107</td>
<td>69</td>
<td>375</td>
<td>366</td>
</tr>
<tr>
<td>Latency (Clock cycles)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>57</td>
<td>57</td>
</tr>
<tr>
<td>Slice registers</td>
<td>1075 (1%)</td>
<td>989 (1%)</td>
<td>546 (&lt; 1%)</td>
<td>3232 (3%)</td>
<td>N/A</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>6826 (14%)</td>
<td>6117 (13%)</td>
<td>1839 (4%)</td>
<td>1903 (4%)</td>
<td>N/A</td>
</tr>
<tr>
<td>DSP48</td>
<td>15 (2%)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6.5. Implementation characteristics of different single-precision division realizations.

<table>
<thead>
<tr>
<th>Device</th>
<th>Ours1 (Hung)</th>
<th>Ours2 (Ito)</th>
<th>Out3 (Ito unfold)</th>
<th>Xilinx IP [36]</th>
<th>Altera IP [37]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency (MHz)</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>288</td>
<td>358</td>
</tr>
<tr>
<td>Latency (Clock cycles)</td>
<td>14</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Slice registers</td>
<td>507 (&lt; 1%)</td>
<td>645 (&lt; 1%)</td>
<td>626 (&lt; 1%)</td>
<td>1374 (&lt; 1%)</td>
<td>N/A</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>444 (&lt; 1%)</td>
<td>520 (&lt; 1%)</td>
<td>486 (&lt; 1%)</td>
<td>800 (&lt; 1%)</td>
<td>N/A</td>
</tr>
<tr>
<td>DSP48</td>
<td>4 (&lt; 1%)</td>
<td>5 (&lt; 1%)</td>
<td>8 (&lt; 1%)</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6.6 presents the implementation results of different double-precision division modules. The results show that with the same number of pipeline stages, our designs are faster than the VFLOAT’s division. Notice that for division, both designs use reciprocal and multiplication for implementing division [39] and both reciprocal designs use the multiplicative approach. The division design of VFLOAT library is using a size of $2^{14} \times 16 = 262144$ bits LUT [13], while our optimized LUT for one iteration has the size of $2^{13} \times 26 = 212992$ bits, and for two iterations has the size of $2^6 \times 15 = 960$ bits, which are much smaller. Using the same device and hardware resources configuration, our unfolded design is slightly slower than the unfolded algorithm PolyGS, but the LUT size of our design (960-bit) is smaller than the PolyGS’s (21504-bit) [8]. Moreover, our design is faster than the FloPoCo floating-point library’s which uses a Radix-4 digit-recurrence method [38].
<table>
<thead>
<tr>
<th>Device</th>
<th>Ours1 (Ito)</th>
<th>Ours2 (Ito unfold)</th>
<th>VFLOAT [14]</th>
<th>PolyGS [8]</th>
<th>FloPoCo [38]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency (MHz)</td>
<td>168</td>
<td>166</td>
<td>148</td>
<td>230</td>
<td>136</td>
</tr>
<tr>
<td>Latency (Clock cycles)</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>LUT size (bit)</td>
<td>212992</td>
<td>960</td>
<td>262144</td>
<td>21504</td>
<td>N/A</td>
</tr>
<tr>
<td>Slice registers</td>
<td>1267 (1%)</td>
<td>1754 (2%)</td>
<td>934 (1%)</td>
<td>1244 (1%)</td>
<td>2509 (2%)</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>4960 (11%)</td>
<td>10024 (22%)</td>
<td>6957 (15%)</td>
<td>1297 (3%)</td>
<td>4419 (10%)</td>
</tr>
<tr>
<td>DSP48</td>
<td>6 (2%)</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.6. Implementation characteristics of different double-precision division realizations.

6.5 SUMMARY

This chapter first discussed the pipeline architectures utilizing relatively small LUTs. Then, we implemented various pipelined arbitrary-precision schemes with the optimized datapath in both single-precision and double-precision formats, and compared our implementation results with other published designs providing a comprehensive comparative analysis on performance and hardware costs.
CHAPTER 7
CONCLUSION

Choosing numerically stable algorithms and efficient implementations of elementary functions requires careful consideration of various parameters, such as the convergence rate of the algorithm, resource constraints, including the available on-chip memory, logical resources, dedicated multipliers and latency, and required performance. This thesis explored the fundamental trade-offs between alternative high-throughput techniques for compact implementation of the three key elementary operations $\frac{1}{x}$, $\sqrt{x}$ and $\frac{1}{\sqrt{x}}$.

To permit the quantitative analysis of the design trade-offs, we developed a parameterizable set of fixed-point and floating-point routines in MATLAB that permitted us to minimize the precision of datapath variables and the size of the look-up tables (LUTs). We performed exhaustive bit-true simulations across the approximation interval $[1, 2)$ and determined the errors to ensure that the proposed precisions provide the 24-bit accuracy required for single-precision floating-point arithmetic, while minimizing the size of the required hardware. We investigated different multiplicative methods and compared their performance and resource requirements.

Then, we designed arbitrary-precision fixed-point and floating-point schemes for $\frac{1}{x}$, $\sqrt{x}$ and $\frac{1}{\sqrt{x}}$. Also, an arbitrary-precision design for the division operation using the multiplicative approach was designed and implemented in both fixed-point and floating-point format. Proposed designs were pipelined and unfolded to improve their performance.

Finally, we compared the implementations results of our arbitrary-precision floating-point reciprocal, square root, and division with state-of-the-art published work at both single-precision and double-precision accuracy. It was shown that our optimized designs for arbitrary-precision floating-point reciprocal, square root, and division can achieve higher throughputs.

7.1 SUMMARY OF THE THESIS ACHIEVEMENTS

The highlights of the major tasks accomplished in the thesis are as follows:

- Completely analyzed and validated several multiplicative methods for calculating three basic arithmetic operations, reciprocal, square root, and reciprocal square root.
• Exhausively simulated and implemented the multiplicative algorithms and compared their performances under normalized single-precision format.

• Designed and implemented the arbitrary-precision scheme for the three basic arithmetic operations, reciprocal, square root, and reciprocal square root in both fixed-point and floating-point format.

• Designed and implemented the arbitrary-precision division scheme in fixed-point and floating-point format.

• Applied pipelining and unfolding techniques to improve the performance of our designs.

• Compared and analyzed the implementation results between our original and optimized designs with other published work and IP cores made available by industry.
BIBLIOGRAPHY


