768735

Design and Implementation of Reciprocal Unit for Interval and Floating-Point Arithmetic

by

Umut Küçükkabak

A Thesis Submitted to the
Graduate School of Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Electrical & Computer Engineering

Koç University

February, 2005

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Umut Küçükkabak

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

[illegible]

Asst. Prof. Ahmet Akkaş (Advisor)

[illegible]

Prof. Reha Civanlar

[illegible]

Asst. Prof. Alper Demir

Date: February 4, 2005

Peace at home, peace in the world.

Yurtta sulh, cihanda sulh.

Mustafa Kemal Atatürk, 1931

To my family and my granddad, Süleyman.

ABSTRACT

Floating-point computations suffer from undetected errors due to rounding and catastrophic cancellation. Fast computers let programmers write numerically intensive programs, but computed results can be far from the true results due to the accumulation of errors in arithmetic operations. Interval arithmetic provides an efficient method for monitoring and controlling errors in numerical computations. With interval arithmetic, each data value is represented by two floating-point numbers which correspond to the endpoints of an interval, such that the true result is guaranteed to lie on this interval. To support interval arithmetic, several software tools have been developed including interval arithmetic libraries, extended scientific programming languages, and interval enhanced compilers. The main disadvantage of these software tools is their speed, since interval operations are implemented using function calls. To speed up interval arithmetic, hardware support for interval arithmetic operations (addition/subtraction, multiplication, division, and comparison/selection) has been developed.

In this research, hardware support for interval reciprocal operation is investigated. The combined interval and floating-point reciprocal unit is designed to support both interval and floating-point reciprocal operations. The unit that supports reciprocal operation for interval and floating-point arithmetic is implemented at gate level and tested. Furthermore, the unit is synthesized to estimate the number of gates and delays.

ÖZETÇE

Kayan noktalı hesaplamalar, yuvarlama ve iptallenmeler sonucu ortaya çıkan hatalardan ötürü olumsuz etkilenmektedir. Hızlı bilgisayarlar programcılara yoğun sayısal hesaplamalar içeren programlar yazmalarına müsade etmekte, fakat aritmetik işlemlerdeki hataların toplanması sonucu hesaplanan sonuçlar doğru sonuçlardan çok farklı olabilmektedir. Aralıklı aritmetik, sayısal hesaplamalarda ortaya çıkan hataların izlenmesi ve kontrol edilmesi için etkin bir yöntemdir. Aralıklı aritmetik ile her veri değeri, aralığın son noktalarını oluşturan iki kayan noktalı sayıdan oluşur ve doğru sonucun bu aralıkta bulunacağı garanti edilir. Aralıklı aritmetiği destekleyen birçok yazılım aracı geliştirilmiştir. Bunların başlıcaları; aralıklı aritmetik program kitaplıkları , genişletilmiş bilimsel programlama dilleri ve aralıklı aritmetiği destekleyen derleyicilerdir. Aralıklı aritmetik işlemleri işlevsel (fonksiyonel) çağrılar ile gerçekleştirildiği için, bu yazılım araçlarının temel götürüsü yavaş olmalarıdır. Aralıklı aritmetiği hızlandırmak için, aralıklı aritmetik işlemleri (toplama/çıkarma, çarpma, bölme ve karşılaştırma/seçme) için donanımsal destekler geliştirilmiştir.

Bu araştırmada, aralıklı bir sayının tersini almak için donanımsal destek incelenmiştir. Aralıklı ve kayan nokta sayılarının her ikisinin de tersini hesaplayabilen birleşik bir ünite tasarlanmıştır. Aralıklı ve kayan noktalı sayılarının her ikisinin de tersini hesaplayabilen bu ünite, geçit (kapı) seviyesinde gerçekleştirilmiş ve test edilmiştir. Daha sonra, yaklaşık olarak geçit sayısını ve gecikmeyi tahmin edebilmek için unite sentez edilmiştir.

ACKNOWLEDGMENTS

I would like to sincerely thank my advisor Dr. Ahmet Akkaş, for his belief in me and my thesis, endless support and encouragement since the beginning of my graduate studies. I am very glad to get the chance of experiencing his kindness. His attitude has never given me any hesitation to communicate with him, both as an advisor and as a friend.

I would also like to thank Prof. Dr. Reha Civanlar, Dr. Alper Demir, Dr. Hakan Ürey, Dr. Attila Gürsoy, Dr. Yücel Yemez and Dr. Öznur Özkasap for their contributions to my profession during my graduate studies.

I would like to specially thank Mr. Demir for his helps in my technical questions. I am thankful to Mr. Civanlar and Mr. Demir for serving on my advisory committee.

My graduate study has not only advanced my profession, but I have earned life-time friends here at Koç University. I would like to thank Mustafa Can Filibeli, Ferit Ozan Akgül, Can James Wetherilt, Mehmet Emre Yavuz, Tanır Özçelebi, Can Kızılkale, Işıl Yıldırım, Burcu Sağlam, Müge Pirtini, Kıvılcım Büyükhatipoğlu, Tayyar Önal and many others for wonderful times and memories. My sincere gratitude goes to Ali Selim Aytuna. We have graduated from Middle East Technical University together with Ali Selim and in the past six years, I have earned a life-long brother. Also, I am very glad to call Mustafa Can and Ferit Ozan as my new brothers.

Finally, I would like to state my everlasting gratitude to my family. I would like to thank my father, **Mehmet**, my mother, **Berna**, my beloved sister, ***Nazlı***, my American host-mother, Karen Saldana, my lovely cousins, Ilgın, Beylûn, Pelin and Gülbin, my aunts, Beyza, Verda, Zehra and Şenay, my uncles Yılmaz, Cengiz and Serdoğan, my dearest grandmothers, Pakize and Şükriye, and all other family members for teaching me the life itself and being the reason of my presence.

TABLE OF CONTENTS

List of Tables x

List of Figures xi

Nomenclature xii

Chapter 1: Inroduction 1

1.1 The Need for Reliable Computing 1

1.2 Motivation for Reciprocal Operation 3

1.3 Reciprocal Unit 3

1.4 Contribution 4

1.5 Outline 5

Chapter 2: Background 6

2.1 Floating-Point 6

2.1.1 Floating-Point Formats 7

2.1.2 Rounding Errors in Floating-Point Arithmetic 10

2.1.3 Rounding Modes in IEEE-754 Floting-Point Arithmetic 11

2.2 Interval Arithmetic 12

2.2.1 Interval Arithmetic Applications 13

2.2.2 Interval Arithmetic Operations 13

2.3 The Taylor Series 18

2.4 Newton-Raphson Method 19

Chapter 3: Floating-Point Reciprocal Unit 22

3.1 An Overview to Compute Reciprocal 22
3.2 Computing Power of an Operand 23
3.3 Floating-Point Reciprocal Unit 26
3.3.1 Computing Reciprocal of an Operand 27
3.3.2 Determining the Table Look-up Values 27
3.3.3 Operand Modification Phase 28
3.3.4 Computation and Accuracy of the Initial Reciprocal Approximation 30
3.3.5 Newton-Raphson Iteration 30
3.3.6 Implementation of Floating-Point Reciprocal Unit 33
3.3.7 Area and Delay Estimates 39
3.4 Comparisons 41
3.5 Conclusions 43

Chapter 4: A Combined Interval and Floating-Point Reciprocal Unit 45
4.1 An Overview to the Combined Reciprocal Unit 46
4.2 Implementation of the Combined Reciprocal Unit 47
4.2.1 Reciprocation of a Double Precision Floating-Point Number . 47
4.2.2 Reciprocation of an Interval Number 48
4.3 Area and Delay Estimates 51
4.4 Conclusions 53

Chapter 5: Conclusions 56

Appendix A: Refinement of the Table Look-up Coefficient 58

Bibliography 61

Vita 70

LIST OF TABLES

2.1 Maximum Rounding Errors with Rounding to Nearest(Even) Mode . 11

3.1 Selection Bits of the Multiplexors Utilized in the Reciprocal Unit . . 35
3.2 WRITE_ENABLE Signal for REG5 37
3.3 Area Estimates of the Components in the Reciprocal Unit 40
3.4 Delay Estimates of the Stages in the Reciprocal Unit 40

4.1 CONTROL Signal for MUX4 49
4.2 WRITE_ENABLE2 Signal for REG6 50
4.3 WRITE_ENABLE3 Signal for REG7 50
4.4 Area Estimates of the Components in the Combined Reciprocal Unit 52
4.5 Delay Estimates of the Stages in the Combined Reciprocal Unit . . . 53

LIST OF FIGURES

2.1 IEEE-754 Single Precision Floating-Point Representation 7

2.2 IEEE-754 Double Precision Floating-Point Representation 9

2.3 Representation of 0.1 by Single Precision Floating-Point Representation 11

2.4 Interval Addition and a Numerical Example 14

2.5 Interval Subtraction and a Numerical Example 15

2.6 Interval Multiplication and a Numerical Example 15

2.7 Interval Division and a Numerical Example 16

2.8 Interval Reciprocal and a Numerical Example 17

2.9 Interval Reciprocal Square-root and a Numerical Example 18

2.10 Graphical Interpretation of Newton-Raphson Method 20

3.1 Reciprocal Unit Implementation 44

4.1 Combined Reciprocal Unit Implementation 55

NOMENCLATURE

VLSI	Very Large-Scale Integration
NR	Newton Raphson
ulp	Unit in the Last Place
SRT	Sweeney, Robertson, Tocher
IEEE	Institute of Electrical and Electronics Engineers
NaN	Not a Number
FPU	Floating-Point Unit
LSB	Least Significant Bit
AC	Alternating Current
ROM	Read Only Memory
CLA_53	Carry Lookahead Adder of 53-bits
CLA_64	Carry Lookahead Adder of 64-bits
DTM_53	Dadda Tree Multiplier of 53-bits
MUX	Multiplexor
REG	Register
OpMod	Operand Modifier
INV_51	Inverters of 51-bits
RND_UP	Round toward $+\infty$ Unit
RND_DOWN	Round toward $-\infty$ Unit
hex	in Hexadecimal Representation
DSP	Digital Signal Processing

Op	Operand
KB	KiloByte
MB	MegaByte
Hz	Hertz
MHz	Megahertz
ns	nanoseconds

Chapter 1

INRODUCTION

1.1 The Need for Reliable Computing

VLSI technology advances as parallel to the statement of Gordon Moore, known as *Moore's Law*, [1]. The number of transistors on an integrated circuit still continues to double its predecessor within 18 to 24 months, as stated. Beyond semiconductor technology scaling, the innovative design approach on computer hardware boosts microprocessor performance, hence originates a great impact on computer speed [2]. Improvements on computer speed provides the ability to perform trillions of arithmetic operation per second.

Despite the improvement on the speed of the arithmetic computations, the precision obtained has not altered over the past two decades. Today, most modern processors support IEEE double and/or double-extended precision floating-point arithmetic, which was defined in 1985. The accuracy of double precision number is about *fifteen* decimal digits. IEEE floating-point standard is being modified to support quadruple precision arithmetic to increase the precision in numerical computations.

Quadruple precision numbers are often 128-bit and consist of a sign bit, 15-bit exponent and 112-bit fraction. The accuracy of quadruple precision floating-point numbers is about *thirty-three* decimal digits. Even though the accuracy of computations is improved using quadruple precision numbers, floating-point computations do not provide a direct method to determine the accuracy of the result. In other words, a floating-point number contains no accuracy information and moreover, it is impossible to represent most real numbers using finite precision floating-point format.

Since floating-point arithmetic provides limited accuracy, it can lead to large errors after a few consecutive operations. As an example, the multiplication of the following matrices

$$A = \begin{bmatrix} -10^{18} \\ 2246 \\ 10^{27} \\ 10^{25} \\ 22 \\ 10^{5} \end{bmatrix} \quad \textit{and} \quad B^T = \begin{bmatrix} 10^{38} \\ 33 \\ 10^{29} \\ -10^{22} \\ 1044 \\ 10^{42} \end{bmatrix}$$

yields the result

$$A \cdot B = 0$$

using IEEE-754 double precision floating-point arithmetic, where the matrix B is given in its transpose form. However, the correct multiplication yields the following result:

$$A \cdot B = 97,086$$

Catastrophic cancellation and round-off error introduced by floating-point arithmetic may cause such inaccurate results. In today's world, a wide range of researches -from astronomy to genetics- contain mass computations with long computation times and as a consequence *reliable computing* is extremely necessary in sequential arithmetic operations.

Interval arithmetic provides an efficient method for monitoring and controlling errors in floating-point computations. Round-off behavior, uncertainty in input data and nonlinear problems can easily be dealt with interval arithmetic. As a definition, an interval is the set of all real numbers between and including the interval's lower and upper bounds. Interval arithmetic is used to evaluate arithmetic expressions over sets of numbers contained in intervals. Hence, any interval arithmetic result is a new interval that is guaranteed to contain the set of all possible resulting values. For example, the interval X = [2.34, 2.35] has a lower interval endpoint 2.34 and an upper interval endpoint 2.35. These two endpoints bound the true result, which is greater than or equal to 2.34 and less than or equal to 2.35. When performing interval arithmetic on computers, one or both of the endpoints may not be representable as a floating-point number. In this case, the interval endpoints are computed by *outward rounding.* Outward rounding requires that the lower endpoint is rounded downward towards negative infinity, and the upper endpoint is rounded upward towards positive infinity.

1.2 Motivation for Reciprocal Operation

The reciprocal operation is important in scientific computing, digital signal processing, and multimedia applications [3], [4].

Besides unique utilization of reciprocal, division operation can also be implemented by reciprocation. Reciprocal of the divisor and a subsequent multiplication of the reciprocal result with the dividend yields division. Division is the most time consuming arithmetic operation and it is difficult to pipeline. As a result, in mass computations, a replacement to the division may yield a significant decrease in the computation time.

1.3 Reciprocal Unit

The reciprocal unit presented in this thesis supports floating-point and interval arithmetic. The proposed method to compute the reciprocal has two stages:

- Computing Initial Approximation
  - Table Look-up and Operand Modification
  - Multiplication
- Newton-Raphson Iteration

Necessity and implementation of the stages will be given in details in the following chapters. Table look-up, operand modification, and multiplication are required in order to compute the initial approximation for the reciprocal. On the other side, Newton-Raphson Method is used to increase the accuracy of the initial approximation result and applied twice in the proposed implementation.

1.4 Contribution

In this thesis, my contributions are as follows:

- The theory of reciprocal operation using table look-up and multiplication proposed by Takagi [5] is investigated in details and is used to compute the initial approximation of reciprocal.
- The Newton-Raphson algorithm is configured in order to be utilized for reciprocal operation.
- The computation of the ROM values are carried out and the refinement of the coefficient is conducted using Maple.
- The number of bits, which are used to index the table, is optimized and set to 12 (this number will be denoted as m in Chapter 3).
- First the reciprocal unit is implemented for floating-point arithmetic.
- Then, the floating-point reciprocal unit is extended to support interval arithmetic. The extended version of the floating-point reciprocal unit is called the combined interval and floating-point arithmetic reciprocal unit.
- Both units are implemented and simulated in VHDL. Units are synthesized to determine the clock cycle time and to estimate the number of gates.
- Replacement of the division is suggested by using the implemented reciprocal unit with an additional multiplication operation.

1.5 Outline

The outline for this thesis is as follows: Chapter 2 presents the required background for floating-point and interval arithmetic, Taylor Series, and Newton-Raphson Method. Chapter 3 proposes the method utilized for reciprocal operation and gives the reciprocal unit implementation for floating-point. Furthermore, the area and delay estimates of the implementation and the comparisons are also discussed in here. Chapter 4 presents the combined reciprocal unit implementation which supports floating-point and interval arithmetic. Finally, Chapter 5 gives our conclusions.

Chapter 2

BACKGROUND

In this chapter, floating-point, interval arithmetic, Taylor Series, and Newton-Raphson method are briefly described. IEEE-754 Standard for floating-point number is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and the most Unix platforms. On the other hand, *interval arithmetic* is being employed within a wide range of scientific applications, which is associated with elaborate mathematics.

In the last two sections of this chapter, an overview of the Taylor Series and Newton-Raphson Method can be found.

2.1 Floating-Point

Among several methods, *floating-point* is the most commonly utilized representation to approximate real numbers on computers. Floating-point representation basically represents reals in a scientific notation.

In order to compare with fixed point representation, floating-point employs a sort of sliding window of precision, appropriate to the scale of the number, which allows it to span numbers approximately in a range of 1.17 x 10^{-38} to 3.40 x 10^{38} for single-precision and 2.22 x 10^{-308} to 1.79 x 10^{308} for double-precision standard representation, excluding infinite values [6], [7], [8], [9].

On the other hand, fixed point representation has a fixed window of precision, which limits it from representing very large or very small numbers. Also, loss of precision is unavoidable when two large numbers are divided using fixed-point representation.

2.1.1 Floating-Point Formats

After a history of confusing and complex representations of numbers in computers [10], both as in terms of hardware and software; nowadays, most of the general purpose computer architectures are based on IEEE-754 Standard [9] and support *single, double* and *double-extended* floating-point numbers.

The IEEE-754 single precision floating-point number requires a 32-bit word, which may be represented as numbered from 31 to 0, left to right, as shown in Figure 2.1. The first bit from left, which is annotated as the 31st bit, is the sign bit, *S*. The next 8-bit represent the exponent bits, *Exponent*, and the remaining 23-bit are the fraction bits, *Mantissa.*

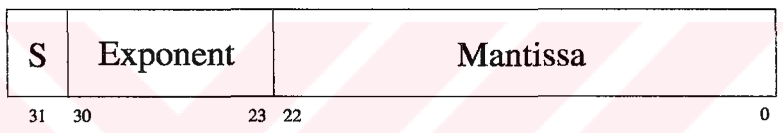

Figure 2.1: IEEE-754 Single Precision Floating-Point Representation

The *sign bit*, which serves to identify whether the represented real number to be negative or positive, is set to logical one (1) or zero (0), respectively. The *biased exponent* has implicitly determined base set to *two*, which is not explicitly stored in the representation. The exponent part is utilized in order to determine the value of the represented number and the related information can be viewed in the following rules. The *mantissa* of the represented real number is composed of the fraction part with an implicit leading (hidden) digit, for which the details can be found below.

The following rules are defined in IEEE-754 Standard [9] for single precision floating-point representation in order to determine the value, *V*, represented by a 32-bit word:

- If $E = 255$ and F is nonzero, then $V =$ NaN (Not a Number)
- If $E = 255$ and F is zero and $S = 1$, then $V = -\infty$

- If $E = 255$ and F is zero and $S = 0$, then $V = \infty$

- If $0 < E < 255$ then $V = (-1)^S \text{ x } 2^{E-127} \text{ x } (1.F)$,
  where; $1.F$ is intended to represent the binary number created by prefixing the fraction part with an implicit leading 1 (hidden one), and a binary point (.), as well the exponent bias being set to 127.

- If $E = 0$ and F is nonzero, $V = (-1)^S \text{ x } 2^{-126} \text{ x } (0.F)$,
  where these are defined as *denormalized* numbers.

- If $E = 0$ and F is zero and $S = 1$, then $V = -0$

- If $E = 0$ and F is zero and $S = 0$, then $V = 0$

Not a number, denormalized numbers, infinity, and zero are the special cases of the IEEE-754 Standard. Briefly, *Zero* is either negative zero or positive zero, which are distinct values but they both compare as equal. *Infinity* value represent either negative or positive infinity and is very useful in operations where overflow case occurs. Operations with infinite values are well defined in IEEE-754 Standard. Numbers with non-zero fraction part but having all zeros in exponent part are called *denormalized* numbers and zero can be interpreted as a special form of denormalized number. Special handling methods are employed in arithmetical operations for NaN (*Not a Number*) values, which are used to represent values those do not represent a real number. Arithmetical operations on special numbers are well defined by IEEE-754 Standard. For example, multiplication of zero and infinity yields a NAN and any operation with a NaN results to NaN. Other operations on special numbers can be found in [6].

As shown in Figure 2.2, the IEEE-754 double precision floating-point number requires a 64-bit word, where the first bit is the sign bit, *S*, the next 11-bit are the exponent bits, *Exponent*, and the remaining 52-bit are the fraction bits, *Mantissa*.

Since the number of bits in the fraction part determines the precision and since the number of bits in the exponent part determines the range of representable numbers,

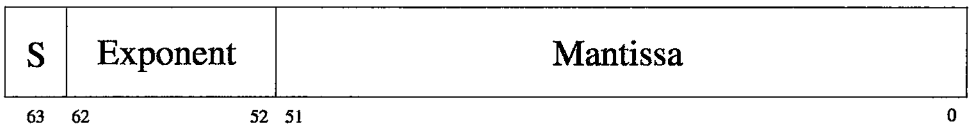

Figure 2.2: IEEE-754 Double Precision Floating-Point Representation

IEEE-754 double precision floating-point numbers have greater range and are more precise than single precision floating-point numbers [9].

The rules defined in IEEE-754 Standard [9] for double precision floating-point representation in order to determine the value, V, represented by a 64-bit word, are detailed below:

- If $E = 2047$ and F is nonzero, then V = NaN (Not a Number)
- If $E = 2047$ and F is zero and $S = 1$, then $V = -\infty$
- If $E = 2047$ and F is zero and $S = 0$, then $V = \infty$
- If $0 < E < 2047$ then $V = (-1)^S$ x 2^{E-1023} x $(1.F)$,
  where; $1.F$ is intended to represent the binary number created by prefixing the fraction part with an implicit leading 1 (hidden one), and a binary point (.), as well the exponent bias being set to 1023.
- If $E = 0$ and F is nonzero, $V = (-1)^S$ x 2^{-1022} x $(0.F)$,
  where these are defined as *denormalized* numbers.
- If $E = 0$ and F is zero and $S = 1$, then $V = -0$
- If $E = 0$ and F is zero and $S = 0$, then $V = 0$

Extensions to well-known single and double precision floating-point, namely single-extended and double-extended floating-point, are available in IEEE-754 Standard, as specified in [9]. The single-extended floating-point word is defined to have more than 43-bit. Corresponding lower limit is 79-bit for double-extended floating-point. However, many scientific computing applications, such as computational geometry, computational physics and climate modelling need more precise arithmetical operations [11]. 128-bit *quadruple precision* arithmetic significantly improves the numerical stability of those scientific applications as stated in [12]. IBM S/390 G5 FPU (floating-point unit) fully supports the quadruple precision arithmetic in hardware [13]. But, even with full hardware support for quadruple precision, the arithmetical operations are *four* times slower than double precision in hardware [14]. Much faster quadruple precision floating-point multiplier is presented in [15], which requires more hardware than IBM S/390 G5 FPU, as a trade-off. IEEE-754 Standard is also being revised to support quadruple precision floating-point [16].

2.1.2 Rounding Errors in Floating-Point Arithmetic

It is impossible to represent some real numbers by IEEE-754 floating-point due to two reasons [17]:

- Real number to be represented may have a finite decimal representation, but in binary, it may have an infinite repeating representation.
- Real number to be represented can be out of range, i.e., the number exceeds the upper and lower representable limits of the corresponding floating-point number.

The *rounding error* is introduced as a consequent of the first reason. One quick example to this situation is to represent 0.1. Actually, 0.1 lies in between two representable floating-point numbers, but none of those floating-point numbers can exactly represent the real number 0.1, because of the infinite repeating sequence in its fraction part, as seen in Figure 2.3.

$$1.10011001100110011001100 \times 2^{-4} = 0.0999999403953552$$

Figure 2.3: Representation of 0.1 by Single Precision Floating-Point Representation

Well-known measure of rounding error is *Unit in the Last Place*, and it is abbreviated as *ulp*. For example, if the infinitely precise real number 0.61271 is represented as 0.6127 in floating-point representation, than the error is stated as 0.1 ulp. Another measure of a rounding error is *relative error* and for the above case, the relative error is approximated as $0.00001/0.61271 \approx 0.00001632$. In this work, ulp will be considered as the measure of rounding error.

Utilizing the rounding to nearest(even) mode, the nearest floating-point number will correspond to a rounding error of less than or equal to 0.5 ulp, as the proof to this statement can be found in [17]. The least rounding error is obtained when the rounding mode is set to rounding to nearest(even) and Table 2.1 shows the maximum possible rounding errors for different precision types. The required accuracy is 1 *ulp* in the remaining rounding modes [18]. The rounding modes are described in the next section.

Precision Type	**Max. Rounding Error (0.5 ulp)**
Single Precision	0.596046447754e-07
Double Precision	0.111022302462515654e-15
Quadruple Precision	0.481482486096808963263994445e-34

Table 2.1: Maximum Rounding Errors with Rounding to Nearest(Even) Mode

2.1.3 Rounding Modes in IEEE-754 Floting-Point Arithmetic

Four rounding modes are defined in IEEE-754 Floating-Point Arithmetic Standard [9]:

- Round to Nearest(Even): The infinitely precise real number is represented with the nearest representable floating-point number. If the two representable values are at the same distance, then the even one (which has a zero in its least significand bit, LSB) is selected.

- Round toward $+\infty$: The infinitely precise real number is represented with the nearest greater representable floating-point number.
- Round toward $-\infty$: The infinitely precise real number is represented with the nearest smaller representable floating-point number.
- Round toward 0 (zero) : The infinitely precise real number is represented with the nearest and smaller in magnitude representable floating-point number.

The IEEE-754 Standard specifies round to nearest(even) as the default rounding mode. However, the rest three rounding modes should be implemented in all IEEE-754 compliant FPU's. In other words, all four rounding modes should be available to be utilized in user-selectable manner.

In order to obtain the most accurate true results in floating-point operations, such as multiplication, many different hardware design approaches are available in the literature for the implementation of the rounding algorithms, that conforms to the IEEE-754 rounding modes. These approaches are considering criterions such as the speed of the rounding and the required hardware complexity. Among such works are [19], [20], [21], [22], [23].

2.2 Interval Arithmetic

In 1962, Ramon E. Moore's dissertation has initiated the modern development of interval arithmetic [24]. *Interval arithmetic* is defined on sets of intervals and an interval $X = [x_l, x_u]$ consists of two real numbers; a lower endpoint x_l and an upper endpoint x_u, such that $x_l \leq x_u$. The interval X is the set of real numbers x, satisfying $x_l \leq x \leq x_u$. In the rest of this chapter, the lowercase letters will denote real numbers and the uppercase letters will denote intervals.

2.2.1 Interval Arithmetic Applications

Today, since interval arithmetic is extremely powerful to bound the range of functions, it is widely used in various computations and numerical applications in fields of science and engineering. Global optimization algorithms, bounding the error term in Taylor's Theorem, bounding the effects of roundoff errors and solving nonlinear systems are the major topics where interval arithmetic is frequently utilized. Specific numerical applications of interval arithmetic are solving linear systems [25], [26], nonlinear systems [27], [28], [29], [30], adaptive quadrature [31], initial value problems and error bounds for ordinary differential equations [32], boundary value problems [33] and integral equations [34]. Many scientific and engineering applications, such as verification of chaotic behaviors of dynamical systems [35], process design [36], computational geometry [37], fluid mechanics [38], [39], computer graphics [40], solving AC network equations [41], quality control in the manufacture of radio-electronic devices [42], Hurwitz stability in control theory [43], medical diagnosis systems [44], remote sensing [45], examination of uncertainty effects in economics [46] and quality control in manufacturing processes [47] employ interval arithmetic.

2.2.2 Interval Arithmetic Operations

Interval arithmetic operations are designed to produce interval results such that true result is guaranteed to lie within the interval endpoints. Operations on interval endpoints may yield real numbers which can not be represented as a floating-point number. In order to contain the true result in the resulting interval, *outward rounding* is performed to compute the lower and upper endpoints of the resulting interval. Outward rounding proceeds as follows. The resultant lower endpoint is rounded towards negative infinity and the resultant upper endpoint is rounded towards positive infinity. In the following, ∇ will denote rounding towards negative infinity and Δ will denote rounding towards positive infinity.

To provide closure over the interval operations, *special intervals* including empty interval, the entire interval, and intervals with infinite endpoints are defined [48]. The *empty interval* does not contain any real numbers and is represented by setting each endpoint to Not a Number (NaN). $\oslash$ denotes an empty interval. The *entire interval* contains all real numbers and is denoted as $[-\infty, +\infty]$. Finally, there are two *intervals with infinite endpoints*:

- $[-\infty, x]$, which represents all real numbers less than or equal to x.
- $[x, +\infty]$, which represents all real numbers greater than or equal to x.

2.2.2.1 Addition and Subtraction

When adding two intervals, the lower endpoint of the resulting interval is computed by adding the operands' lower endpoints with rounding towards negative infinity. The upper endpoint of the resulting interval is obtained by adding the operands' upper endpoints with rounding towards positive infinity. An addition of two intervals and a numerical example is given in Figure 2.4.

$$Z = X + Y$$
$$[z_l, z_u] = [\nabla x_l + y_l, \Delta x_u + y_u]$$

$$[3.68, 3.73] = [1.42, 1.45] + [2.26, 2.28]$$

Figure 2.4: Interval Addition and a Numerical Example

Interval subtraction is similar to interval addition, as seen in Figure 2.5. However, the upper endpoint of the subtrahend is subtracted from the lower endpoint of the minuend with rounding towards negative infinity. Accordingly, the lower endpoint of the subtrahend is subtracted from the upper endpoint of the minuend with rounding towards positive infinity.

2.2.2.2 Multiplication

Multiplication of two interval numbers is more complex than addition or subtraction. The upper endpoint of the product is computed by the maximum of each endpoints

$$Z = X - Y$$
$$[z_l, z_u] = [\nabla x_l - y_u, \Delta x_u - y_l]$$
$$[-0,86, -0.81] = [1.42, 1.45] - [2.26, 2.28]$$

Figure 2.5: Interval Subtraction and a Numerical Example

multiplication. Similarly, the lower endpoint of the product is computed by the minimum of each endpoints multiplications as observed in Figure 2.6. In straightforward manner, the result can be obtained by four multiplications and three comparisons.

$$Z = X \cdot Y$$
$$[z_l, z_u] = [\nabla min(x_l y_l, x_l y_u, x_u y_l, x_l y_l), \Delta max(x_l y_l, x_l y_u, x_u y_l, x_u y_u)]$$
$$[3.2092, 3.3060] = [1.42, 1.45] \cdot [2.26, 2.28]$$

Figure 2.6: Interval Multiplication and a Numerical Example

In order to perform multiplication of intervals faster and make it worth to replace floating-point arithmetic, it is required to examine the sign bits of endpoints of the multiplier and the multiplicand. In that case, it is possible to describe nine separate cases for different sign combinations of the operands:

1. If $x_l \geq 0$ and $y_l \geq 0$, then $X \cdot Y = [\nabla x_l y_l, \Delta x_u y_u]$

2. If $x_l \geq 0$ and $y_l \leq 0 \leq y_u$, then $X \cdot Y = [\nabla x_u y_l, \Delta x_u y_u]$

3. If $x_l \geq 0$ and $y_u \leq 0$, then $X \cdot Y = [\nabla x_u y_l, \Delta x_l y_u]$

4. If $x_l \leq 0 \leq x_u$ and $y_l \geq 0$, then $X \cdot Y = [\nabla x_l y_u, \Delta x_u y_u]$

5. If $x_l \leq 0 \leq x_u$ and $y_u \leq 0$, then $X \cdot Y = [\nabla x_u y_l, \Delta x_l y_l]$

6. If $x_u \leq 0$ and $y_l \geq 0$, then $X \cdot Y = [\nabla x_l y_u, \Delta x_u y_l]$

7. If $x_u \leq 0$ and $y_l \leq 0 \leq y_u$, then $X \cdot Y = [\bigtriangledown x_l y_u, \triangle x_l y_l]$

8. If $x_u \leq 0$ and $y_u \leq 0$, then $X \cdot Y = [\bigtriangledown x_u y_u, \triangle x_l y_l]$

9. If $x_l \leq 0 \leq x_u$ and $y_l \leq 0 \leq y_u$, then $X \cdot Y = [\bigtriangledown min(x_u y_l, x_l y_u), \triangle max(x_l y_l, x_u y_u)]$

In each combination, the resulting interval can be computed by two multiplications, except case 9. In case 9, four multiplications and two comparisons should be performed. However, since multiplication of floating-point numbers takes considerable amount of time, even case 9 can be performed in shorter. In other words, three multiplications can be sufficient to obtain the result for case 9 by the fast algorithm of Heindl [49] and as well by [50], [51]. Another study predicts an average time of two multiplications in order to perform interval multiplication [52].

2.2.2.3 Division

Division of two intervals is similar to multiplication of two intervals, except the case where zero lies in between the lower and upper endpoints of the divisor. In that case, the quotient interval will be the whole representable real numbers, i.e., $[-\infty, +\infty]$. Interval division is shown in Figure 2.7.

$$Z = \frac{X}{Y}$$

$$[z_l, z_u] = \left[\bigtriangledown min(\tfrac{x_l}{y_l}, \tfrac{x_l}{y_u}, \tfrac{x_u}{y_l}, \tfrac{x_u}{y_u}), \triangle max(\tfrac{x_l}{y_l}, \tfrac{x_l}{y_u}, \tfrac{x_u}{y_l}, \tfrac{x_u}{y_u})\right]$$

$$[0.5, 0.8] = \frac{[1.0,1.2]}{[1.5,2.0]}$$

Figure 2.7: Interval Division and a Numerical Example

It is possible to extract seven different cases for the interval division if the sign bits of endpoints of the dividend and the divisor are examined initially:

1. If $x_l > 0$ and $y_l > 0$, then $\frac{X}{Y} = \left[\bigtriangledown \frac{x_l}{y_u}, \triangle \frac{x_u}{y_l}\right]$

2. If $x_l > 0$ and $y_u < 0$, then $\frac{X}{Y} = \left[\bigtriangledown \frac{x_u}{y_u}, \triangle \frac{x_l}{y_l}\right]$

3. If $x_u < 0$ and $y_l > 0$, then $\frac{X}{Y} = \left[\nabla\frac{x_l}{y_l}, \Delta\frac{x_u}{y_u}\right]$

4. If $x_u < 0$ and $y_u < 0$, then $\frac{X}{Y} = \left[\nabla\frac{x_u}{y_l}, \Delta\frac{x_l}{y_u}\right]$

5. If $x_l < 0 < x_u$ and $y_l > 0$, then $\frac{X}{Y} = \left[\nabla\frac{x_l}{y_l}, \Delta\frac{x_u}{y_l}\right]$

6. If $x_l < 0 < x_u$ and $y_u < 0$, then $\frac{X}{Y} = \left[\nabla\frac{x_u}{y_u}, \Delta\frac{x_l}{y_u}\right]$

7. If $y_l < 0 < y_u$, then $\frac{X}{Y} = [-\infty, +\infty]$

2.2.2.4 Reciprocal of an Interval

Reciprocal of an interval is straight forward and obeys interval division rules in which the dividend is unity, as seen in Figure 2.8. However, there are two special cases. If the interval, for which the reciprocal will be computed, is an empty interval; then the resulting interval is an empty interval as well. Also, if the divisor interval contains zero, then the resulting interval is the entire interval.

$$\frac{1}{X} = \begin{cases} \oslash & : \text{if interval } X \text{ is empty} \\ [-\infty, +\infty] & : \text{if } x_l \leq 0 \leq x_u \\ [\ \nabla(\frac{1}{x_u}), \Delta(\frac{1}{x_l})] & : \text{elsewhere} \end{cases}$$

$$\frac{1}{[1.25, 1.60]} = [0.625, 0.800]$$

Figure 2.8: Interval Reciprocal and a Numerical Example

2.2.2.5 Reciprocal Square-root of an Interval

Reciprocal square-root of an interval is similar to reciprocal of an interval, except the special cases. When the upper endpoint of the divisor interval is less than zero or when the divisor interval is an empty interval, then the resulting interval is also empty. Finally, when the lower endpoint of the divisor interval is less than zero and the upper endpoint of the divisor interval is greater than zero, then the resulting interval is an interval with an infinite endpoint, as shown in Figure 2.9.

$$\frac{1}{\sqrt{X}} = \begin{cases} \emptyset & : \text{if } x_u < 0 \text{ or interval } X \text{ is empty} \\ [\nabla(\frac{1}{\sqrt{x_u}}), +\infty] & : \text{if } x_l < 0 \text{ and } x_u > 0 \\ [\nabla(\frac{1}{\sqrt{x_u}}), \Delta(\frac{1}{\sqrt{x_l}})] & : \text{elsewhere} \end{cases}$$

$$\frac{1}{\sqrt{([-2,4])}} = \frac{1}{\sqrt{([0,4])}} = [0.5, +\infty]$$

Figure 2.9: Interval Reciprocal Square-root and a Numerical Example

2.3 The Taylor Series

Taylor series is widely employed in prediction of the function value at one point. Moreover, the Taylor series representation can approximate any smooth function as a polynomial [53].

Taylor's Theorem states that if the function f and its first $n+1$ derivatives are continuous on an interval containing a and x, then the value of the function at x is given by Equation (2.1), which is known as *Taylor's formula.*

$$\begin{aligned} f(x) &= f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 \\ &+\frac{f'''(a)}{3!}(x-a)^3 + \cdots \\ &+\frac{f^{(n)}(a)}{n!}(x-a)^n + R_n \end{aligned} \tag{2.1}$$

where the remainder R_n is defined as follows:

$$R_n = \int_a^x \frac{(x-t)^n}{n!} f^{(n+1)}(t)dt \tag{2.2}$$

where $t = a$ is a dummy variable.

For functions whose values alter on the interval of interest, the more terms included in the Taylor series representation, the better estimate of the function is provided. *Zero*, *first* and *second-order* Taylor series are listed below, respectively:

$$f(x_{i+1}) \simeq f(x_i) \tag{2.3}$$

$$f(x_{i+1}) \simeq f(x_i) + f'(x_i)(x_{i+1} - x_i) \tag{2.4}$$

$$f(x_{i+1}) \simeq f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2 \tag{2.5}$$

In this research, first-order Taylor series approximation, given in Equation (2.4), is utilized in both determining the constant values, which constitutes the required table for the reciprocal unit, and computing of the power of an operand, as well. The details on how Taylor series is employed will be given in Chapter 3.

2.4 Newton-Raphson Method

Newton-Raphson method is the most widely used of all root-locating formulas [53]. It can be investigated in Figure 2.10 that a tangent line to a function will cross the x-axis at a point which is an approximate to the root of the function. As the Newton-Raphson method is applied consecutively, a closer estimate to the root is obtained.

The slope of the tangent line in Figure 2.10 is denoted by $f'(x_i)$ and can be calculated by geometrical interpretation such that:

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}} \tag{2.6}$$

Newton-Raphson formula given below is obtained by re-arranging the Equation (2.6):

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2.7}$$

Once the Newton-Raphson method is applied, the error becomes roughly propor-

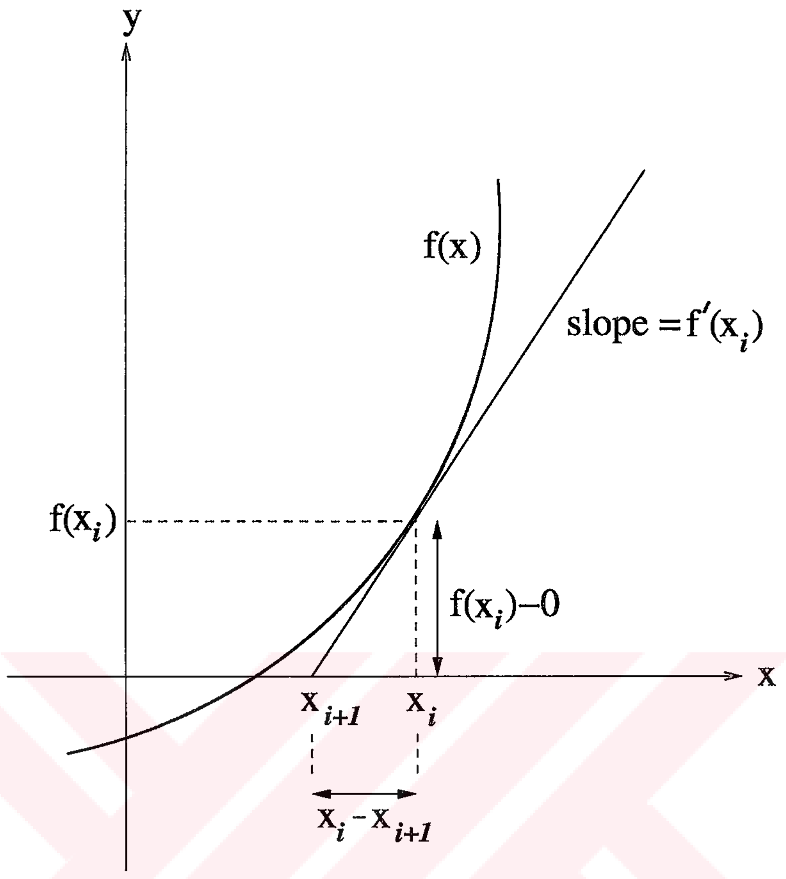

Figure 2.10: Graphical Interpretation of Newton-Raphson Method

tional to the square of the previous error. Hence, the number of significant figures of accuracy approximately doubles itself in each Newton-Raphson iteration, which provides the property of *quadratically convergence* to the method.

The derivation of the Newton-Raphson method can also be carried out by Taylor series. Since the location of the root is being considered, the obtained result should be zero in the first-order Taylor series given in Equation (2.4):

$$0 = f(x_i) + f'(x_i)(x_{i+1} - x_i) \tag{2.8}$$

which can be rewritten as;

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2.9}$$

Equation (2.9) is derived from Taylor series expansion and equal to Newton-Raphson formula given in Equation (2.7).

Quadratically convergence property of Newton-Raphson method is the reason that the method is employed to improve the accuracy of the initially obtained reciprocal approximation in this work. The details on how Newton-Raphson method is applied in the presented reciprocal unit will be given in Chapter 3.

Chapter 3

FLOATING-POINT RECIPROCAL UNIT

Among the arithmetic operations, the division operation is the most time consuming operation because the number of cycles used to determine the quotient is proportional to the number of bits of the dividend. It is also difficult to pipeline the division operation due to the dependencies between the iterations [54], [55].

As shown in Equation (3.1), division can be implemented by computing the reciprocal of the divisor. In order to finalize the division operation, the result of reciprocal operation is utilized as the multiplier in a subsequent multiply operation, where the dividend is being the multiplicand.

$$\frac{a}{x} = a \cdot \frac{1}{x} \tag{3.1}$$

In this chapter, an implementation of reciprocal unit designed for IEEE-754 double precision floating-point number is presented. The design utilizes a table look-up and Newton-Raphson iteration. The theory for computing the initial approximation of the reciprocal is by Takagi [5].

3.1 An Overview to Compute Reciprocal

A method used to compute reciprocal of an operand consists of two basic steps:

- Computation of the initial reciprocal approximation,
- Two consecutive Newton-Raphson iterations.

The piecewise linear approximation, based on the first-order Taylor expansion [56] [5], is used to compute the reciprocal of the operand approximately. This approximate result will be mentioned as *initial approximation* throughout this work.

In order to compute the initial approximation, a constant, C, is read through a table look-up first. As well, a modification to the operand, whose details will be given in the next section, is applied. Finally, multiplication and addition follow the table look-up and operand modification procedure to obtain the initial approximation.

The Newton-Raphson iterations construct the second phase of the method. Each iteration of Newton-Raphson yields a quadratic convergence to the true result. In other words, error of the previous step squares itself in each Newton-Raphson iteration, which in turn results in doubled accuracy as compared to the previous approximation [53]. This ensures that, if more than thirteen correct bits are acquired in the mantissa part of the initial approximation, it is guaranteed to obtain the correct reciprocal result after two consecutive Newton-Raphson iterations as seen in Equation (3.2):

$$14\ bits \rightarrow 28\ bits \rightarrow 56\ bits \tag{3.2}$$

3.2 Computing Power of an Operand

In this section, a method for generating the *p-th* power of an operand, X_{op}, is given. The power, p, is in the form of -2^k (where, k is 0 for reciprocal and k is -1 for reciprocal square-root operation). The operand, X_{op}, is an IEEE compliant 64-bit normalized double precision floating-point number as shown in Figure 2.2 and it is in the decimal range of $1 \le X_{op} < 2$. The hidden-one and the least significand 52-bit of X_{op} altogether represents the mantissa part and will be called as *operand*, X, throughout this work. Operand X is an *(n+1)*-bit binary number and n is 52 for double precision number.

The binary representation of the operand, X, is shown in Equation (3.3):

$$X = [1.x_1x_2x_3...x_{52}] \tag{3.3}$$

where;

$$x_i \epsilon \{0,1\} \tag{3.4}$$

The modification of the operand and the determination of the ROM values are carried out based on Taylor series expansion. In order to represent the power of the operand by Taylor series expansion, the operand X is separated into two parts from 1^{st} to m^{th} and from $(m+1)^{th}$ to n^{th} bit, as proposed by Takagi [5]. Below, the operand, X, is apparently the sum of its two parts:

$$X_1 = [1.x_1 x_2 x_3 ... x_m] \tag{3.5}$$

$$X_2 = [0.x_{m+1} x_{m+2} ... x_n] \times 2^{-m} \tag{3.6}$$

where;

$$X = X_1 + X_2 \tag{3.7}$$

Given in Equation (2.1), an approximated version of the Taylor series expansion by truncating the series after the first derivative term can be represented as first-order Taylor series:

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) \tag{3.8}$$

Power of an operand, which is denoted by X^p, is approximated based on first-order Taylor series as in Equation (3.8):

$$X^p \simeq (X_1 + 2^{-m-1})^p + p \cdot (X_1 + 2^{-m-1})^{p-1} \cdot (X_2 - 2^{-m-1}) \tag{3.9}$$

where the function shown in Equation (3.10);

$$f(x) = x^p \tag{3.10}$$

and below equalities are considered:

$$\begin{aligned}
x_i &= (X_1 + 2^{-m-1}) && (3.11)\\
x_{i+1} &= X && (3.12)\\
f(x_i) &= (X_1 + 2^{-m-1})^p && (3.13)\\
f'(x_i) &= p \cdot (X_1 + 2^{-m-1})^{p-1} && (3.14)\\
f(x_{i+1}) &= X^p && (3.15)\\
x_{i+1} - x_i &= X - (X_1 + 2^{-m-1}) && \\
&= X - X_1 - 2^{-m-1} = (X_2 - 2^{-m-1}) && (3.16)
\end{aligned}$$

The Taylor series linear approximation for a power of an operand adopts a *linear function* as described in [57] and can be represented such that:

$$X^p \simeq C_0 + C_1 X_2 \qquad (3.17)$$

where;

$$\begin{aligned}
C_0 &= (X_1 + 2^{-m-1})^p - p \cdot (X_1 + 2^{-m-1})^{p-1} \cdot (2^{-m-1}) && (3.18)\\
C_1 &= p \cdot (X_1 + 2^{-m-1})^{p-1} && (3.19)
\end{aligned}$$

The conventional linear approximation representation given above in Equation (3.17) will require a great size of table since two coefficients, C_0 and C_1, are necessary. One multiplication, C_1X_2, and one addition operation, $C_0+C_1X_2$ are required in order to obtain the initial approximation.

A new method which significantly reduces the size of the table by providing only one coefficient, and also which eliminates the addition operation is proposed by Takagi [5].

Rewriting Equation (3.9), power of the operand, X^p, can also be expressed as shown in Equation (3.20):

$$X^p \simeq (X_1 + 2^{-m-1})^{p-1}[(X_1 + 2^{-m-1}) + p \cdot (X_2 - 2^{-m-1})] \qquad (3.20)$$

Above arrangement constructs the new method proposed by Takagi. Finally, the below representation given in Equation (3.21) can be employed in a way that only *one* multiplication operation will be sufficient in order to compute the initial approximation with an easy modification of the operand, X'. As well, the utilized table (ROM) size is reduced, since only one coefficient, C, is required.

$$X^p \simeq C \cdot X' \qquad (3.21)$$

where;

$$C = (X_1 + 2^{-m-1})^{p-1} \qquad (3.22)$$

$$X' = [(X_1 + 2^{-m-1}) + p \cdot (X_2 - 2^{-m-1})] \qquad (3.23)$$

This manipulation yields the same accuracy in bits for the initial approximation compared to the conventional linear approximation. Moreover, as previously mentioned, the required ROM size is reduced significantly and the entire reciprocal computation time is shortened by one clock cycle since the addition is removed.

3.3 Floating-Point Reciprocal Unit

This section is dedicated for computing the reciprocal of an operand. First, the theory on how to reciprocate an operand by table look-up and multiplication method will be introduced. Later, the utilization of Newton-Raphson method will be detailed, which provides more accurate reciprocal result. Finally, the implemented unit will be presented and obtained area-delay estimates as a result of synthesizing the unit will be given.

3.3.1 Computing Reciprocal of an Operand

The proposed unit in this chapter computes the reciprocal of an operand. Hence, related power, $p = -1$ is considered. The initial reciprocal approximation, X^{-1}, is calculated by referring to Equation (3.9) such that:

$$X^{-1} \simeq (X_1 + 2^{-m-1})^{-1} - (X_1 + 2^{-m-1})^{-2}(X_2 - 2^{-m-1}) \tag{3.24}$$

where;

$$f(x_i) = (X_1 + 2^{-m-1})^{-1} \tag{3.25}$$
$$f'(x_i) = -(X_1 + 2^{-m-1})^{-2} \tag{3.26}$$
$$f(x_{i+1}) = X^{-1} \tag{3.27}$$
$$x_{i+1} - x_i = (X_2 - 2^{-m-1}) \tag{3.28}$$

By rewriting Equation (3.24), reciprocal of X can also be expressed as shown in Equation (3.29), in order to employ the proposed form of $C \cdot X'$ mentioned above:

$$X^{-1} \simeq (X_1 + 2^{-m-1})^{-2}[(X_1 + 2^{-m-1}) - (X_2 - 2^{-m-1})] \tag{3.29}$$

where;

$$C = (X_1 + 2^{-m-1})^{-2} \tag{3.30}$$
$$X' = [(X_1 + 2^{-m-1}) - (X_2 - 2^{-m-1})] \tag{3.31}$$

3.3.2 Determining the Table Look-up Values

The constant, C, given in Equation (3.30) can be refined to C' in order to reduce the error in the initial approximation as proposed by Takagi [5]. The refinement of the coefficient is performed based on the formula given in Equation (3.32). The mathematical computation which clarifies the refinement process is studied in this thesis. The details on refinement of the coefficient are given in Appendix A.

$$\epsilon = C \cdot X' - X^{-1} \tag{3.32}$$

where, ϵ denotes the *error* in $C \cdot X'$. Note that, $C \cdot X'$ is proposed to yield the initial reciprocal approximation.

The refined coefficient is computed as shown below:

$$C' = (X_1)^{-2} - (X_1)^{-3} \cdot 2^{-m} + (X_1)^{-4} \cdot 7 \cdot 2^{-2m-3} \tag{3.33}$$

In the rest of the study, C' is the constant value read from a ROM and the utilized ROM will store words which are pre-determined by Equation (3.33).

As given in Equation (3.5), X_1 is constructed by the first m-bit of the operand. Hence, for the phase of table look-up, the first m-bit of the mantissa of the operand X_{op} will constitute the *index bits* as inputs to the ROM. In other words, the entire X_1 without hidden-one is used to index the ROM.

The size of the values stored in the ROM are determinant on the accuracy of the initial approximation and these values will be called as *ROM words.* As the size of the words stored in the table increases, the initial approximation becomes closer to the true reciprocal result and hence has better accuracy. However, the trade-off here is that, the number of index bits (m-bit) directly determines the size of the ROM. In the presented reciprocal unit, the ROM words are of size $2m$, and with this selection, the utilized ROM size will be:

$$2^m \times 2m \; bits \tag{3.34}$$

3.3.3 Operand Modification Phase

The remaining term in Equation (3.20), which is $[(X_1 + 2^{-m-1}) + p \cdot (X_2 - 2^{-m-1})]$, has to be formed and provided by the operand modifier. For the reciprocal case, the term becomes $[(X_1 + 2^{-m-1}) - (X_2 - 2^{-m-1})]$, which is denoted by X' in Equation

(3.31).

Rearranging the above term, X' can be expressed as:

$$X' = (X_1 - X_2 + 2 \cdot 2^{-m-1}) \tag{3.35}$$

$$= (X_1 - X_2 + 2^{-m}) \tag{3.36}$$

Keeping the bits of X_1 fixed and appending bitwise inverted version of X_2, as shown in Equation (3.37), yields a very precise approximation to exact computation of the modified operand given in Equation (3.36):

$$X' \simeq [1.x_1 x_2 x_3 ... x_m \tilde{x}_{m+1} \tilde{x}_{m+2} ... \tilde{x}_n] \tag{3.37}$$

where; complement of x_i is denoted by $\tilde{x}_i$.

In fact, exact X' would be obtained with an additional term, 2^{-n}, but in this work the last term in Equation (3.38) is omitted while modifying the operand.

$$X' = [1.x_1 x_2 x_3 ... x_m \tilde{x}_{m+1} \tilde{x}_{m+2} ... \tilde{x}_n] + 2^{-n} \tag{3.38}$$

This disregard is apprehensible since the purpose of the first stage is to obtain an initial reciprocal approximation and in any case, the following Newton-Raphson stages will converge on the true reciprocal. As well, the omission is advantageous since an addition operation is saved during the operand modification, which would introduce an extra delay to the initial approximation process.

3.3.4 Computation and Accuracy of the Initial Reciprocal Approximation

The initial reciprocal approximation is computed by multiplication of the term C' (read from ROM) with the modified operand X'.

$$X^{-1} \simeq C' \cdot X' \tag{3.39}$$

The multiplication of $C' \cdot X'$ is conducted by a 53×53 Dadda Tree Multiplier, which will be utilized for all multiplication operations throughout the complete procedure, covering initial approximation stage and Newton-Raphson stages. Moreover, the utilization of the existing multiplier on the floating-point unit of the processor is also feasible, in order reduce hardware cost significantly. This multiplier is abbreviated as DTM_53 in next sections.

Among a set of numbers, trials show that the obtained accuracy of the initial approximation is about $2m \pm 3$ bits.

3.3.5 Newton-Raphson Iteration

Newton-Raphson method is employed in the process of increasing, in fact, doubling the accuracy of the previously obtained initial approximation of reciprocal.

An overview to the method and its mathematical proof can be found in the last section of Chapter 2.

As detailed explicitly below, Newton-Raphson method is applied twice in an iterative manner in this work, in order to increase the accuracy of the initial approximation of reciprocal obtained by table look-up. In each Newton-Raphson iteration, the number of true bits in the mantissa approximately doubles itself. The proof to the quadratically convergence of the Newton-Raphson method can be viewed in the next section.

Below assignments will construct the form of employed Newton-Raphson method in the presented unit:

- The operand is denoted by X,
- The initial reciprocal approximation of the operand is denoted by x_i,
- The reciprocal result after first Newton-Raphson iteration applied to initial approximation is denoted by x_{i+1}.

The function $f(x)$, given in Equation (3.40), should vanish for the specific x value which is selected as the reciprocal of X.

$$f(x) = X - \frac{1}{x} \tag{3.40}$$

where;

$$x = \frac{1}{X} \tag{3.41}$$

Inserting the function given in Equation (3.40) into the general Newton-Raphson formula given in Equation(3.42), the mathematical expression for the utilized Newton-Raphson iteration form is obtained, as shown in Equation (3.44).

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{3.42}$$

$$x_{i+1} = x_i - \frac{X - \frac{1}{x_i}}{\frac{1}{{x_i}^2}} \tag{3.43}$$

$$x_{i+1} = x_i(2 - Xx_i) \tag{3.44}$$

where;

$$f'(x) = \frac{1}{x^2} \tag{3.45}$$

In order to double the accuracy of the previous reciprocal result, the operation given in Equation (3.44) is conducted by means of hardware and called as Newton-Raphson iteration throughout this thesis.

The Newton-Raphson iteration phase needs two multiplication and one subtraction operations to yield x_{i+1}. The multiplications will be performed with the previously utilized 53 $\times$ 53 Dadda Tree Multiplier, DTM_53. On the other hand, the subtraction operation is a subtraction from 2, so the hardware equivalent is apparently the *two's complement* of Xx_i. While computing the two's complement of Xx_i, first the bitwise inversion of all bits of Xx_i is carried out. Finally, a logical one is added to the least significand bit position of the inverted result, using the 53-bit Carry Lookahead Adder. The mentioned adder will be denoted as CLA_53 in the rest of this work.

3.3.5.1 Error Analysis of Newton-Raphson Implementation

The error of initial reciprocal approximation is assumed to be E_{x_i}, such that:

$$E_{x_i} = x_i - \frac{1}{X} \tag{3.46}$$

which can be rearranged as shown in Equation (3.47);

$$x_i = E_{x_i} + \frac{1}{X} \tag{3.47}$$

Inserting x_i into the Newton-Raphson form, which is specifically derived for reciprocation in Equation (3.44), will yield the following Equation (3.48):

$$x_{i+1} = \frac{1}{X} - XE_{x_i}^2 \tag{3.48}$$

Similar to Equation (3.47), the error of reciprocal result maintained after the first Newton-Raphson iteration can be defined as:

$$x_{i+1} = E_{x_{i+1}} + \frac{1}{X} \tag{3.49}$$

Equations (3.48) and (3.49) are both equal to x_{i+1}, so the right hand side of these equations are also equal to each other.

$$\frac{1}{X} - XE_{x_i}^2 = E_{x_{i+1}} + \frac{1}{X} \tag{3.50}$$

$$E_{x_{i+1}} = -XE_{x_i}^2 \tag{3.51}$$

Equation (3.51) proves that the absolute error degrades quadratically in each Newton-Raphson iteration as it is proportional to the square of previous error.

3.3.6 Implementation of Floating-Point Reciprocal Unit

The hardware design that supports reciprocal operation is shown in Figure 3.1. The implemented design consists of three pipeline stages. The first pipeline stage constructs the table look-up and operand modification phase. The second pipeline stage performs the multiplication. The last pipeline stage manipulates the results by shift, bitwise inversion or buffer operations.

The reciprocal operation takes total of eleven clock cycles. The first three clock cycles yields the initial reciprocal approximation. Each Newton-Raphson iteration, in order to obtain more accurate reciprocal result, is completed in four clock cycles. Eight clock cycles are dedicated for Newton-Raphson iterations, since it is applied twice.

The first pipeline stage, consisting of ROM and the operand modifier (OpMod), is only used in the first clock cycle of the reciprocal operation to obtain the constant, C' (Constant), and the modified operand, X' (Modified_Op). Therefore, pipeline registers are not used between the first and the second pipeline stages. In the following, the components of the stages and their respective operations will be explained in details.

The computations of the implemented unit are based on the 52-bit mantissa, with given constraints below:

- X is a 64-bit normalized double precision floating-point number in the decimal range of $1 \leq X < 2$. As a result, the first 12-bit of X, i.e., the sign and the exponent bits of X, is equal to $(3FF)_{hex}$.

- Given the constraint on the operand X, the exact reciprocal, X^{-1}, is supposed to be in the decimal range of $0.5 < X^{-1} \leq 1$. The most significand 12-bit of X^{-1} should have $(3FE)_{hex}$.

Hence, the sign bit and exponent bits will not be considered in the rest of this study.

As to speak over the implemented unit, the most significand m-bit of the mantissa of the operand forms the index bits of the table look-up procedure, whereas the entire mantissa is provided to the OpMod unit initially.

As previously mentioned, a trade-off appears while selecting the number m, as it is determinant on both the ROM size and the accuracy of the initial approximation. Utmost care is taken to keep the ROM size minimum in order to prevent the cost of hardware. On the other hand, the accuracy obtained by initial approximation has to be sufficient to yield the correct result after two Newton-Raphson iterations. Among many experiments performed on the implemented unit, the most effective m, which yields exactly correct results at all trials is chosen, such that;

$$m = 12 \tag{3.52}$$

With this selection, the employed ROM should have size of 12KB, referring to Equation (3.34):

$$2^{12} \times 24 \; bits = 2^{10} \times 96 \; bits = 96 \; Kbits = 12 \; KB \tag{3.53}$$

In the 1st clock cycle, the ROM takes as inputs the most significand 12-bit of the mantissa from REG1 register, in which the input operand is saved in 64-bit IEEE compliant format. The ROM is constructed with the values that are computed as described in section 3.3.2. The procedure for forming a specific 24-bit ROM word is as follows: 23-bit of the mantissa of the floating-point number computed by Equation (3.33) is taken. This 23-bit constructs the least significand 23-bit of the ROM word. On the other side, the least significand bit of the exponent of the floating-point number computed by Equation (3.33) constructs the most significand bit of the ROM word.

As a result, a 24-bit ROM word is obtained.

The least significand bit of the exponent of the ROM value computed by Equation (3.33), is determinant on the flow of the presented unit. The initial approximation result is normalized whether by *shifting left 1-bit* or *2-bit*, depending on the *odd* or *even* exponent of the ROM value, respectively. The least significand bit of the exponent is sufficient to choose the right operation and is supplied from the ROM to the multiplexor MUX3 as the most significand selecting input, S2. Details on the selection bits of all of the multiplexors utilized in the unit are summarized in Table 3.1.

Clock Cycle	**Select Bits**			**Selected Input**		
	S2	**S1**	**S0**	**MUX1**	**MUX2**	**MUX3**
2nd, 3rd	0	0	0	Constant	Modified_Op	L_1_SHIFT
4th, 5th, 8th, 9th	0	0	1	Mantissa	REG5	INV_51
7th	0	1	0	X	X	L_2_SHIFT
6th, 10th, 11th	0	1	1	REG4	REG5	CLA64_Out
2nd	1	0	0	Constant	Modified_Op	X
4th, 5th, 8th, 9th	1	0	1	Mantissa	REG5	INV_51
3rd, 7th	1	1	0	X	X	L_2_SHIFT
6th, 10th, 11th	1	1	1	REG4	REG5	CLA64_Out
Note: S2 = First Bit of ROM, X = Don't Care						

Table 3.1: Selection Bits of the Multiplexors Utilized in the Reciprocal Unit

A logical one is concatenated to the front of the least significand 23-bit of the read value from the ROM and 29 zeros are appended to the right of the word in order to fulfill the 53-bit.

On the other side, the mantissa of the operand is supplied to the OpMod unit. The OpMod unit is constructed with forty inverters, in which the most significand 12-bit of the mantissa stays the same, whereas the least significand 40-bit are bitwise inverted, as explained in Equation (3.37). Within the OpMod unit, a logical one is appended as a most significand bit of the 52-bit word in order that the output of 53-bit is suitable for multiplication.

In the 2nd clock cycle, these two 53-bit values are selected by MUX1 and MUX2 multiplexors. The selected ROM output constitutes the refined constant, C' (Constant), and the selected OpMod unit output is the modified operand, X' (Modified_Op), referring to Equation (3.39). As a final step of the 2nd clock cycle, the outputs of MUX1 and MUX2 are multiplied by the 53×53 Dadda Tree Multiplier, DTM_53, and results in carry-save format are stored in the registers REG2 and REG3. These results are kept in carry-save format to avoid carry-propagate addition, which would increase the delay.

In the 3rd clock cycle, the contents of registers REG2 and REG3 are added by 64-bit Carry Lookahead Adder, CLA_64. Depending on the most significand bit of the ROM, either *shift left 1-bit* (L_1_SHIFT) or *shift left 2-bit* (L_2_SHIFT) output is selected by MUX3. L_1_SHIFT unit takes as inputs the most significand 52-bit of the CLA_64 output and shifts it to the left 1-bit by inserting a zero at the end and truncates the most significand bit. Finally, a hidden-one is appended to the left of the word for the oncoming multiplication in the L_1_SHIFT unit. L_2_SHIFT is conducted in similar manner to L_1_SHIFT, for which two logical zeros are inserted to the end of the CLA_64 output, where two most significand bits are truncated. The hidden-one is as well appended as the most significand bit of the word.

The MUX3 output is stored into both registers REG4 and REG5, at the end of the 3rd clock cycle. Note that, REG5 is a register with WRITE_ENABLE signal, where in some cases, the register value will not be updated in order to keep the previous result for future operations. This memory behavior is the requirement of the Newton-Raphson iteration stages. Table 3.2 summarizes the clock cycles in which REG5 is enabled. At the end of the 3rd clock cycle, the initial reciprocal approximation, x_i, is obtained with an accuracy of $2m \pm 3$ bits and it is stored into registers REG4 and REG5. The computed initial approximation will be iterated twice by Newton-Raphson method in the next eight clock cycles.

Note that, each Newton-Raphson iteration takes four clock cycles. Second and third pipeline stages are used twice in each Newton-Raphson iteration. In the follow-

Effective Clock Cycle	WRITE_ENABLE
3^{rd}, 7^{th}, 11^{th}	1
else	0

Table 3.2: WRITE_ENABLE Signal for REG5

ing, implementation details of the Newton-Raphson iteration is given.

At the beginning of the 4^{th} clock cycle, 53-bit mantissa of the original operand, Mantissa, is selected by MUX1. MUX2 selects the input coming from register REG5, which contains the initial reciprocal approximation. These two values are multiplied in the 4^{th} clock cycle. The multiplication of these two values will yield Xx_i, in the Newton-Raphson formula given in Equation (3.44).

In the 5^{th} clock cycle, the multiplication results are saved in REG2 and REG3 and the register values are added by CLA_64. In order to obtain the mathematical expression, $2 - Xx_i$, the required operation is to take *two's complement* of Xx_i, provided by CLA_64. Excluding the hidden-one, 51-bit of the multiplication result is bitwise inverted by INV_51. Later, two bits, a hidden-one and a consequent zero is appended as the most significand 2-bit of the inverted word. Inverted result is incremented by *one*, using CLA_53. One of the operand in CLA_53 is constant and equal to one. *One* is formed by 51-bit logical zeros and a logical one as the least significand bit. Finally, $2 - Xx_i$ is obtained and saved in register REG4. However, in this clock cycle, WRITE_ENABLE signal is set to zero and register REG5 is not updated. Hence, register REG5 still preserves the initial approximation result, x_i.

Note that, the adder CLA_53 is used to obtain two's complement of Xx_i, to compute $2-Xx_i$. In the first draft design of the reciprocal unit, instead of taking two's complement of Xx_i, one's complement is used as was done in the IBM 360/91 division algorithm [58]. Even though the small error introduced using one's complement is 1 ulp compared to two's complement, simulation results show that the final reciprocal results may be 2 ulp different than the results obtained using IEEE compliant floating-point unit. Therefore, the extra adder, CLA_53, is used instead of only bit inversion, to reduce the error from 2 ulp to 1 ulp in the final reciprocal result. Although using CLA_53 slightly increases the hardware, it does not effect the clock cycle time of the reciprocal unit, because the delay of the third pipeline stage is still less than the second pipeline stage.

During the 6th clock cycle, MUX1 selects the value in REG4, which is $2 - Xx_i$, and MUX2 selects the value of REG5, in which the initial approximation, x_i, is saved. Their multiplication is performed in the same clock cycle to yield $x_i(2 - Xx_i)$.

In the 7th clock cycle, products in carry-save format stored in registers REG2 and REG3 are added by CLA_64. Later, MUX3 selects the output of L_2_SHIFT unit and provides the result to REG4 and REG5, for which WRITE_ENABLE signal is set to one. At the end of the 7th clock cycle, first iteration of the Newton-Raphson, x_{i+1}, is available in both registers REG4 and REG5. This concludes the first Newton-Raphson iteration and the number of accurate bits is doubled compared to the initial reciprocal approximation.

The 8th, 9th, 10th and 11th clock cycles are dedicated to conduct the second Newton-Raphson iteration. These are obviously the repetition of the 4th, 5th, 6th and 7th clock cycles, respectively. The exception here is that, in the last clock cycle, MUX3 selects the exact multiplication result, which is shown as CLA64_Out. The most significand 53-bit of CLA_64 result is denoted as CLA64_Out. The final reciprocal result is saved in both registers REG4 and REG5.

In this implementation, m is 12 and two Newton-Raphson iterations are applied to obtain more accurate result. Simulation results, among hundreds of normalized floating point numbers, show that 52-bit accuracy including the hidden-one is obtained. However, since an infinite precision to compute the final result is not available, the proposed design does not guarantee the correctness of the least significand bit of mantissa, i.e., the 52th bit, compared to IEEE compliant reciprocal operation. More clearly, in most of the cases investigated, the obtained reciprocal result is same as the reciprocal result provided by an IEEE compliant floating-point unit (FPU) with the default round to nearest(even) mode. There are also a few cases in which the com-

Component	**# of Gates**	**Reference in the Unit**
ROM	207	ROM
REG (64-bit)	521	REG1, REG2, REG3,
REG (53-bit)	536	REG4, REG5
MUX (3-to-1, 53-bit)	518	MUX1
MUX (2-to-1, 53-bit)	445	MUX2
MUX (4-to-1, 53-bit)	479	MUX3
CLA (53-bit)	1011	CLA_53
CLA (64-bit)	1154	CLA_64
INVERTERS	91	INV_51, OpMod
Dadda Tree Multiplier (53-bit)	40811	DTM_53
Total Area	**47144**	

Table 3.3: Area Estimates of the Components in the Reciprocal Unit

puted reciprocal result is 1 ulp more than the results obtained by IEEE compliant FPU. This may not be a problem especially in the area of DSP applications.

In order to summarize the operation of the reciprocal unit, computation of the initial reciprocal approximation takes three clock cycles, which requires table look-up, multiplication, and addition operations. While iterating the initial approximation, each Newton-Raphson iteration takes four clock cycles, and the second and third pipeline stages are used twice in each iteration. Finally, reciprocal operation takes total of eleven clock cycles, excluding the rounding.

3.3.7 Area and Delay Estimates

In order to estimate the area and the worst-case delay, the reciprocal unit is implemented and simulated in VHDL. For the procedure of synthesizing the unit, *LeonardoSpectrum* synthesis tool by Mentor Graphics and TSMC 0.25 micron CMOS standard cell library are employed. The design is optimized for speed with a greater frequency of 250 MHz. Table 3.3 gives area estimates for the components of the whole unit, based on the synthesis results, which were obtained using an operating voltage of 2.5 Volts and a temperature of 25 degrees centigrade.

Note that, the column *# of Gates* in Table 3.3 refers to the number of equiva-

Stage	**Delay (ns)**	**Determinant Components on the Delay of the Stage**
1	1.95	REG1, ROM
2	5.62	REG5, MUX2, DTM_53
3	4.36	REG3, CLA_64, INV_51, CLA_53, MUX3, REG4
Clock Cycle	**5.62**	

Table 3.4: Delay Estimates of the Stages in the Reciprocal Unit

lent gates for each type of components and the results are normalized such that an equivalent gate corresponds to the area of a single minimum-size inverter.

The total area given in Table 3.3 is calculated by adding the referred *# of Gates* values, taking into account that the 64-bit register is used three times and the 53-bit register is used two times in the reciprocal unit.

In order to determine the clock cycle time of the unit, the three pipeline stages are synthesized separately. The components of each pipeline stages and their respective worst case delays are given in Table 3.4.

The worst case delay path occurs in the second pipeline stage where the multiplication operation is performed. Hence, the clock cycle time of the unit is determined on the worst case delay path, which is obtained by adding the delay of REG5, MUX1, and DTM_53, as shown in Equation (3.54).

$$\begin{aligned} t_{clk} &= t_{REG5} + t_{MUX2} + t_{DTM\_53} \\ &= 0.67 + 0.29 + 4.66 \\ &= 5.62ns. \end{aligned} \tag{3.54}$$

As a conclusion, the reciprocal unit takes eleven clock cycles with a clock cycle time of 5.62 nanoseconds to perform reciprocal operation. During the last two clock cycles of reciprocal operation, only the second and the third pipeline stages are used. Therefore, it is possible to start a new reciprocal operation every nine cycles.

3.4 Comparisons

A smaller size of multiplier would be sufficient in order to compute the reciprocal of an operand with the conventional linear approximation. However, the conventional linear approximation requires a table of size $2^m \times 3m$ *bits* and an extra addition operation, given in Equation (3.17).

It is possible to share the existing multiplier in the floating-point unit in order to replace the 53-bit Dadda Tree Multiplier used in the reciprocal unit. This will lead to a significant decrease in the area occupied by the reciprocal unit. As well, the proposed implementation has two thirds ROM size, $2^m \times 2m$ *bits*, which is another advantage in terms of area.

SRT division is the most commonly implemented division algorithm in modern microprocessors. It is based on the *digit recurrence*. Sweeney, Robertson, and Tocher, each developed the algorithm approximately the same time and their initials form the name of the algorithm, SRT.

SRT algorithm uses normalized operands. It allows redundant representations of partial remainders and quotient digits [59]. Basically, the key to the algorithm is to select the quotient digits by comparing the most significand bits of partial remainder with the divisor.

Time complexity of SRT algorithm depends on the length of the divisor and radix. Radix-4 SRT division is frequently used algorithm for division operation, where 2-bit of the quotient bits are determined in each iteration. If the operand is a double precision number, more than 53-bit of quotient is required in order to perform rounding. Hence, Radix-4 SRT division takes approximately 30 clock cycles including initial setup and final rounding.

There are many different approaches to the hardware design of SRT method, in order to implement the algorithm with faster clock cycle times and/or to occupy the minimum areas. Among such works are [59], [60], [61], [62].

Another well-known division algorithms to mention are very high radix and variable latency algorithms [63]. Very high radix algorithms are attractive means of achieving low latency while also providing a true remainder. Division unit of DEC Alpha 21164 processor is implemented using the variable latency algorithm [64]. Whenever a consecutive sequence of zeros or ones is detected in the partial remainder, a similar number of quotient bits are also set to zero, all within the same clock cycle. It is reported that an average of 2.4 quotient bits are retired per cycle using variable latency algorithm algorithm [64]. However, because of the duplication of the quotient-selection logic for speculation, the required area is about 44% more than a conventional radix implementation.

A design similar to the floating-point reciprocal unit presented in Figure 3.1 has been developed by Fowler [65]. The theory for computing the initial approximation is different in [65]; however, it employs Newton-Raphson iteration twice in order to increase the accuracy of the initial approximation. In the design of Fowler, the error introduced for the reciprocal result is investigated based on each rounding mode. However, the area and delay estimates are not presented and the least significand bit is still not guaranteed for reciprocal operation.

The reciprocal unit presented in this thesis may replace the division unit. Even if the Radix-8 SRT division is used, it takes roughly 20 clock cycles for 53-bit mantissa, whereas the reciprocal operation in our design takes 11 clock cycles (excluding the final rounding cycle). A subsequent multiplication, which can be implemented in at most 3 clock cycles, would be sufficient to complete the division operation. The disadvantage of our design is the non-guaranteed accuracy of the least significand bit. This would cause a problem to have an IEEE compliant floating-point unit. However, an error, one unit in the last place (*ulp*), may not be a problem for DSP applications. Another area that the reciprocal through a table look-up and Newton-Raphson iteration can be used is interval arithmetic where the containment of the true result is more important [66]. The modified version of floating-point reciprocal unit which is able to handle both floating-point and interval number will be introduced in Chapter 4.

3.5 Conclusions

The reciprocal unit presented in this chapter takes eleven cycles to compute the reciprocal of a double precision floating-point number (excluding the final rounding cycle). The implemented and proposed design to compute the reciprocal is utilizing table look-up and multiplication in order to obtain an initial approximation and two Newton-Raphson iterations. An accuracy of 52-bit is obtained in the experiments conducted, but the least significand bit is not guaranteed to be same for all cases compared to IEEE compliant floating-point unit.

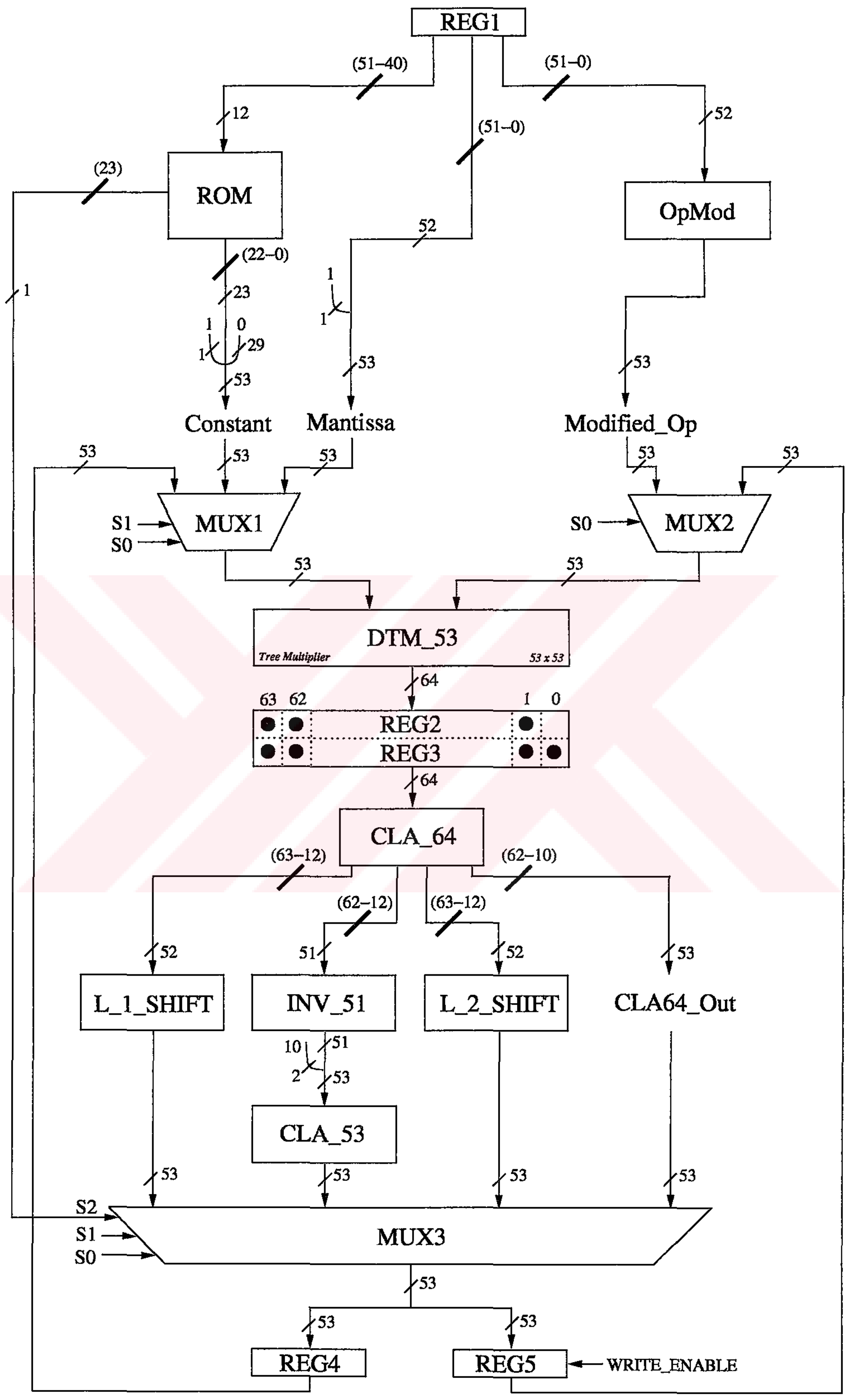

Figure 3.1: Reciprocal Unit Implementation

Chapter 4

A COMBINED INTERVAL AND FLOATING-POINT RECIPROCAL UNIT

Interval arithmetic provides an efficient method for monitoring errors in numerical computations and for solving problems that cannot be efficiently solved with floating-point arithmetic. To provide software support for fixed-precision interval arithmetic, interval arithmetic libraries have been developed. INTLIB is an interval arithmetic library for FORTRAN 77 [67]. Another fixed-precision interval arithmetic library is the Programmer's Runtime Optimized Fast Interval Library (PROFIL) [68]. Profile is a C++ class library that uses a set of Basic Interval Arithmetic Subroutines (BIAS). InC++ is another library for interval computation [69]. Unlike previous libraries, InC++ supports operations on open-ended intervals (e.g. (1.23, 1.24), infinite intervals (e.g. [1.23,+∞]), and discontinuous intervals (e.g. [1.23, 1.24] U [2.14,2.47]).

Techniques for variable-precision arithmetic and interval arithmetic have been combined in the extended scientific programming languages PASCAL-XSC [70], C-XSC [71], ACRITH-XSC [72], and VPI [73]. These languages are extensions to existing programming languages, which allow the programmer to define abstract data types, overload operators, and create dynamic arrays. To support accurate, self-validating computation, these languages provide variable-precision data types for intervals, vectors, matrices, and complex numbers.

Recently, compiler support for interval arithmetic has been developed. The GNU Fortran Compiler has been modified to provide support for single and double precision interval data types [74]. These modifications are based on the Interval Arithmetic Specification [48]. Although the interval-enhanced GNU Fortran compiler provides support for interval arithmetic, it suffers from function call overhead, because all interval operations are implemented by calls to runtime library routines. Sun Microsystems enhanced their Fortran 95 compiler to support intervals as intrinsic data types [75]. Sun's compiler has full support for extended interval data types. It also provides some compiler optimizations, which substantially improve the performance of interval code.

The main disadvantage of software tools for interval arithmetic is their speed. Since the arithmetic operations are simulated in software, tremendous overhead occurs due to function calls, memory management, error checking, changing rounding modes, and exception handing. It is expected that interval arithmetic will become generally accepted when its performance is within a factor of two of floating-point arithmetic. Therefore, hardware support for interval arithmetic is required.

To provide hardware support for interval arithmetic, a coprocessor and several hardware units have been designed. Variable-precision, interval arithmetic coprocessors [76] allow the programmer to specify the precision of the computation, determine the accuracy of the results, and recompute results with higher precision when necessary. The combined interval and floating-point adder/subtracter [77], multiplier [78], divider [79], comparator/selector [80] have also been designed. These combined units support both floating-point and interval operations. In this chapter, hardware support for interval reciprocal operation is investigated.

4.1 An Overview to the Combined Reciprocal Unit

The combined interval and floating-point reciprocal unit computes the reciprocal of either an interval or a floating-point number. The interval endpoints and the floating-point number are all double precision floating-point numbers. When the combined interval and floating-point reciprocal unit is used to compute the reciprocal of an interval, it guarantees that the computed result interval contains the true result. However, for the floating-point reciprocation, the reciprocal result may differ from the true value based on an infinite precision result. When the result do differ, the difference is only 1 ulp more than the true value as it is detailed in Chapter 3.

4.2 Implementation of the Combined Reciprocal Unit

The previously implemented floating-point reciprocal unit shown in Figure 3.1 is modified in order to support both interval and floating-point number. Additional *two* 53-bit registers, REG6, REG7, *two* rounding units, RND_UP, RND_DOWN, a *single* 2-to-1 multiplexor, MUX4, and a *single* 64-bit register, REG8, are used in the combined interval and floating-point reciprocal unit. The hardware design for the combined reciprocal unit is shown in Figure 4.1.

The combined reciprocal unit consists of four pipeline stages. The first pipeline constructs the table look-up and operand modification phase. The second pipeline performs the multiplication. The third pipeline manipulates the results by shift, bitwise inversion or buffer operations. The last pipeline performs rounding.

All of the components and their abbreviations which are used for the floating-point reciprocal unit are also valid in the combined reciprocal unit. However, the additional components used in the combined reciprocal unit will be detailed in the following sections.

4.2.1 Reciprocation of a Double Precision Floating-Point Number

In the combined reciprocal unit, reciprocal operation for a double precision floating-point number is conducted in a similar manner compared to the floating-point reciprocal unit. The operand, X, saved in 64-bit register REG1, is reciprocated in eleven clock cycles, containing the initial approximation phase and two Newton-Raphson iterations.

The CONTROL signal is the selection bit of multiplexor MUX4. In the 1st clock cycle, CONTROL signal is set to zero in order to select the 52-bit mantissa of the operand saved in register REG1. The value of CONTROL signal for different clock cycles is given in Table 4.1.

As the operand is selected by MUX4, the rest of the initial approximation and Newton-Raphson stages are similar to the previously introduced floating-point reciprocal unit. At the end of the 11th clock cycle, the reciprocal result of the double precision floating-point operand is ready and saved in register REG4.

The reciprocal result for floating-point may rarely 1 ulp more than the true value based on an infinite precision result as specified by IEEE-754 Standard. In most of the cases investigated, the obtained reciprocal result is same as the reciprocal result provided by SRT division, performed on an IEEE compliant floating-point unit. However, for some of the cases, whose frequency is extremely low, the obtained reciprocal result is 1 ulp more than the reciprocal result provided by IEEE compliant floating-point unit. Actually, it is safe to say that the obtained reciprocal result is either same or 1 ulp less/more than the true rounded result computed with one of the four rounding modes based on an infinite precision. Therefore, it is not efficient to round the result when the combined unit is used to compute reciprocal of a floating-point number.

4.2.2 Reciprocation of an Interval Number

Reciprocal of an interval number, formed by two double precision floating-point numbers constituting its endpoints, is computed by the combined reciprocal unit.

The operand interval is represented as below:

$$X = [X_l, X_u] \tag{4.1}$$

where; X_l is the lower endpoint of the interval and X_u is the upper endpoint of the interval.

Reciprocation of the interval operand is performed as follows. X_l is saved in register REG1 and X_u is saved in register REG8. The CONTROL signal, whose value for different clock cycles is given in Table 4.1, is set to zero in the 1^{st} clock cycle. As the multiplexor MUX4 selects the mantissa of the lower endpoint of the operand, the reciprocal operation follows the same procedure of the floating-point reciprocation. At the end of the 11^{th} clock cycle, the unrounded reciprocal of X_l is ready and saved in REG4.

Effective Clock Cycle	**CONTROL**
1st	0
10th	1
other	X
Note: **X = Don't Care**	

Table 4.1: CONTROL Signal for MUX4

The reciprocal of the lower endpoint of the operand, $(X_l)^{-1}$ constitutes the upper endpoint for the resultant reciprocal interval and the reciprocal of the upper endpoint of the operand, $(X_u)^{-1}$, formes the lower endpoint for the resultant interval, given in Equation (4.2).

$$X^{-1} = \left[(X_u)^{-1}, (X_l)^{-1}\right] \tag{4.2}$$

In the 12th clock cycle, unrounded reciprocal of the lower endpoint of operand stored in register REG4 is rounded towards positive infinity by RND_UP unit. The upwards rounded result constitutes the upper endpoint of reciprocal interval and guarantees the containment. To round positive infinity, the RND_UP unit adds *one* to the least significand bit of the result saved in register REG4. Rounded result, which forms the upper endpoint of the reciprocal interval result, is saved in register REG6, where WRITE_ENABLE2 signal is set in the 12th clock cycle. Table 4.2 summarizes the clock cycles in which REG6 is loaded.

Since the combined unit is pipelined, it is possible to initiate the computation for the reciprocal of upper endpoint in the 10th clock cycle. In the same clock cycle, the second Newton-Raphson iteration for the reciprocal computation of the lower endpoint is performed using the second pipeline stage. Therefore, at the end of the 20th clock cycle, unrounded reciprocal of the upper endpoint is ready and the result is saved in register REG5.

In the 21st clock cycle, available result in REG5 is read and RND_DOWN unit subtracts *one* from the result in order to round the result towards negative infinity.

Effective Clock Cycle	WRITE_ENABLE2
12th	1
else	0

Table 4.2: WRITE_ENABLE2 Signal for REG6

This will ensure that the result is decreased by 1 ulp. Note that, rounding units, RND_UP and RND_DOWN does not increase the clock cycle time and performs their dedicated operations in one clock cycle. Details of area and delay estimates of the combined reciprocal unit will be given in the following section. Within the same clock cycle, WRITE_ENABLE3 signal, whose details is available in Table 4.3, is set to logical one, and the result, which forms the lower endpoint of the reciprocal interval result, is saved in REG7.

In the case that both endpoints of the interval operand are negative, the rounding units operate in an opposite manner. For that case, the upper endpoint of the result is rounded towards positive infinity by decreasing the result by 1 ulp, and the lower endpoint of the result is rounded towards negative infinity by increasing the result by 1 ulp.

Two special cases as a definition of interval reciprocal operation are given in Figure 2.8. First, the interval operand may contain zero. If it does, then the entire interval is returned as a reciprocal interval result. Second, the interval operand may be empty interval. In this case, the empty interval is returned as a reciprocal interval result.

Effective Clock Cycle	WRITE_ENABLE3
21th	1
else	0

Table 4.3: WRITE_ENABLE3 Signal for REG7

As a conclusion, in twenty-one clock cycles, the reciprocal of an interval is computed and the results are available in the registers REG6 and REG7, which constitute the upper and lower endpoint of the reciprocal result, respectively. Thus, the latency of interval reciprocation is twenty-one clock cycles. Since initiation interval is the number of clock cycles that must elapse between issuing two operations of a given type, the initiation interval is eighteen clock cycles for interval reciprocal operation. The computed interval result is given in Equation (4.3).

$$X^{-1} = \left[\nabla(X_u)^{-1}, \Delta(X_l)^{-1}\right] \tag{4.3}$$

When the combined unit is used to compute the reciprocal of a double precision floating-point number, the latency is eleven clock cycles and the initiation interval is nine clock cycles.

The main advantage of the proposed design is that the true result is guaranteed to lie in the computed reciprocal interval. Other noteworthy advantages of the design are its speed and its ability to replace the division operation for interval arithmetic. The unit would be inexpensive in terms of hardware, if the existing multiplier on the processor is utilized as to replace the multiplier employed in our design. However, the drawback of the hardware support offered for reciprocal of an interval is that the possibility of additional 1 ulp error in each endpoint computation. For some cases, whose frequency is extremely low, if both endpoints are computed with 1 ulp error, then the range of the result interval would be 2 ulp wider compared to the interval result obtained using an infinite precision.

4.3 Area and Delay Estimates

In order to estimate the area and the worst-case delay, the combined reciprocal unit is implemented and simulated in VHDL. For the procedure of synthesizing the unit, *LeonardoSpectrum* synthesis tool by Mentor Graphics and TSMC 0.25 micron CMOS standard cell library are employed. The design is optimized for speed with a greater frequency of 250 MHz. Table 4.4 gives area estimates for the components of the whole unit, based on the synthesis results, which were obtained using an operating voltage of 2.5 Volts and a temperature of 25 degrees centigrade.

Component	**# of Gates**	**Reference in the Unit**
ROM	207	ROM
REG (64-bit)	521	REG1, REG2, REG3, REG8
REG (53-bit)	536	REG4, REG5, REG6, REG7
MUX (3-to-1, 53-bit)	518	MUX1
MUX (2-to-1, 53-bit)	445	MUX2
MUX (4-to-1, 53-bit)	479	MUX3
MUX (2-to-1, 52-bit)	210	MUX4
CLA (53-bit)	1011	CLA_53
CLA (64-bit)	1154	CLA_64
INVERTERS	91	INV_51, OpMod
Dadda Tree Multiplier (53-bit)	40811	DTM_53
Total Area	**50969**	

Table 4.4: Area Estimates of the Components in the Combined Reciprocal Unit

Note that, the column *# of Gates* in Table 4.4 refers to the number of equivalent gates for each type of components and the results are normalized such that an equivalent gate corresponds to the area of a single minimum-size inverter.

The total area given in Table 4.4 is calculated by adding the referred *# of Gates* values, taking into account that the register of 53-bit is utilized four times, the register of 64-bit is utilized four times and Carry Lookahead Adder of 53-bit is utilized three times in the combined reciprocal unit.

The total area of the combined implementation is only 8% more than the floating-point reciprocal unit. However, the clock cycle time remains the same and it is determined by the second pipeline stage.

In order to determine the clock cycle time of the combined reciprocal unit, the four pipeline stages of the combined unit are synthesized separately. The components of the stages and their respective worst case delays are given in Table 4.5. The worst case delay path occurs in the second stage where the multiplication operations are performed. Hence, the clock cycle time of the combined reciprocal unit is determined on the worst case delay path, which is obtained by adding the delay of REG5, MUX2

Stage	**Delay (ns)**	**Determinant Components on the Delay of the Stage**
1	2.62	REG1, MUX4, ROM
2	5.62	REG5, MUX2, DTM_53
3	4.34	REG3, CLA_64, INV_51, CLA_53, MUX3
4	2.23	REG4, RND_UP, REG6
Clock Cycle	**5.62**	

Table 4.5: Delay Estimates of the Stages in the Combined Reciprocal Unit

and DTM_53, as shown in Equation (4.4).

$$\begin{aligned} t_{clk} &= t_{REG5} + t_{MUX2} + t_{DTM\_53} \\ &= 0.67 + 0.29 + 4.66 \\ &= 5.62ns. \end{aligned} \tag{4.4}$$

4.4 Conclusions

The combined reciprocal unit presented in this chapter is able to perform reciprocation of a double precision floating-point number or an interval number formed by two double precision floating-point numbers. It takes eleven clock cycles to compute the reciprocal of a double precision floating-point number and the initiation interval is nine clock cycles. The least significand bit is still not guaranteed for all cases for the floating-point reciprocation.

However, the implemented and proposed combined reciprocal unit is offering an interval reciprocation for which the exact result is guaranteed to be contained in the final reciprocal interval result, which takes twenty-one clock cycles (including the rounding cycle). Interval division operation takes fifty-seven clock cycles using the combined interval and floating-point divider given in [79], whereas if the reciprocal result is used in the subsequent multiplication, it is possible to perform interval division operation in total of twenty-five clock cycles, where the interval multiplication takes four clock cycles [78]. Thus, the combined reciprocal unit also reduces the number of clock cycles required for interval division operation from fifty-seven clock cycles to twenty-five clock cycles. This is a significant improvement.

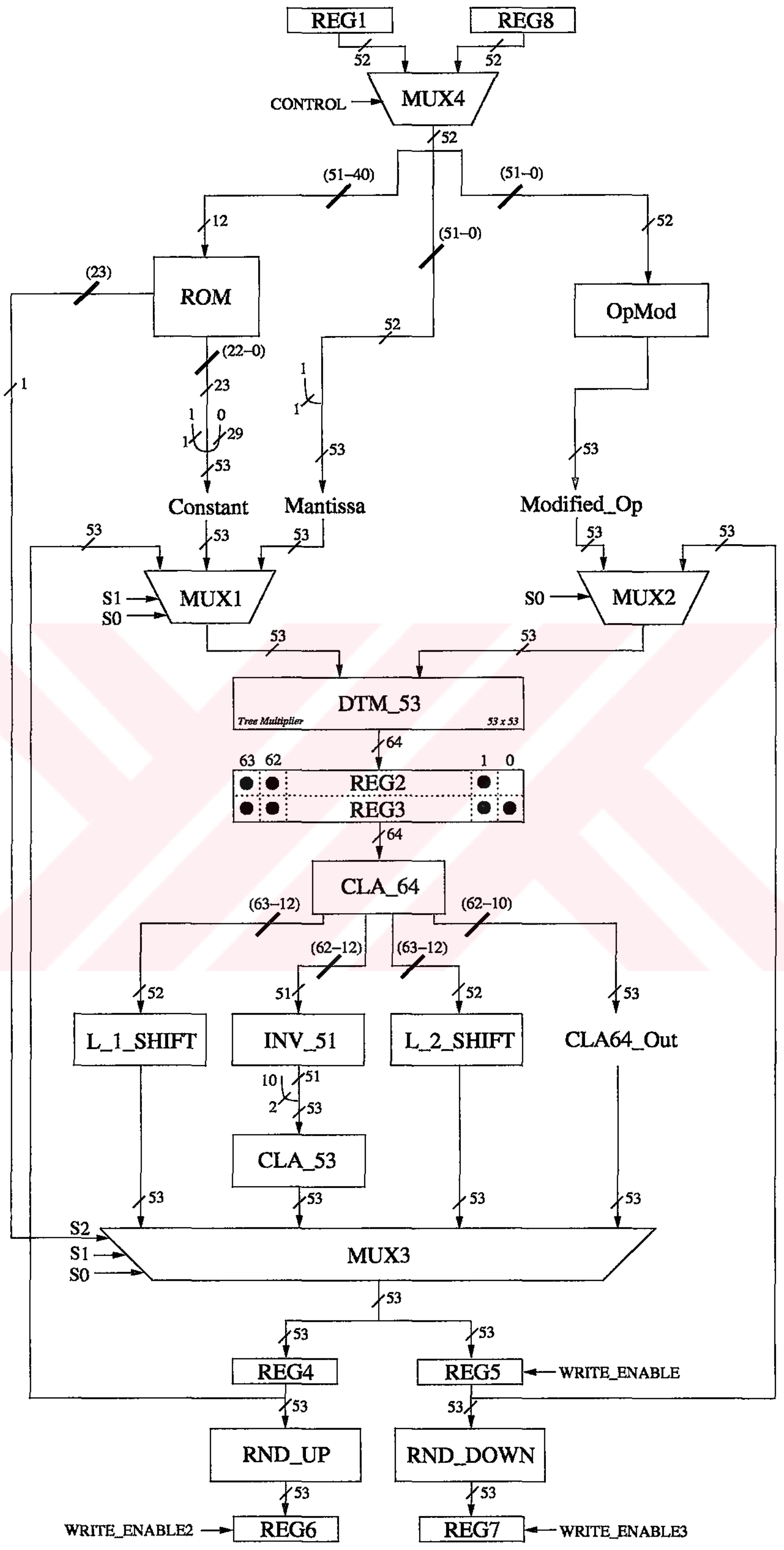

Figure 4.1: Combined Reciprocal Unit Implementation

Chapter 5

CONCLUSIONS

This thesis introduces an implementation of a reciprocal unit for floating-point and interval arithmetic. The method to perform reciprocal operation depends on table look-up and multiplication, basically. Moreover, in order to obtain more precise result, Newton-Raphson Method is used.

At first, reciprocal unit is designed, tested and synthesized for double precision floating-point. Reciprocal of a double precision floating-point number is computed in eleven clock cycles, including phases such that modification of the operand, table look-up, multiplication, and two consecutive Newton-Raphson iterations.

The floating-point reciprocal unit yields a result in which the least significand bit is not guaranteed to be true based on an infinite precision. In fact, for most of the cases investigated, the obtained reciprocal result is same as the reciprocal result provided by an IEEE compliant floating-point unit. However, for some of the cases, whose frequency is extremely low, the obtained reciprocal result is 1 ulp more than the reciprocal result provided by an IEEE compliant floating-point unit. As a result, rounding is not applied.

The combined interval and floating-point reciprocal unit is designed, tested and synthesized. The combined unit computes the reciprocal of a double precision floating-point or an interval number. Hardware support for interval arithmetic is aimed in order to speed up interval operations. For floating-point reciprocal computation performed by the combined unit, rounding is not applied due to the same reasons explained previously.

In interval reciprocal computation, the operand and resultant intervals are both formed by two double precision endpoints. The main advantage of the design is to maintain the containment of the true result. The reciprocal result is obtained in twenty-one clock cycles, including the rounding. The *Round towards Positive Infinity* ($+\infty$) is applied to compute the upper endpoint and the *Round towards Negative Infinity* ($-\infty$) is applied to compute the lower endpoint. After rounding, it is guaranteed that the result interval contains the correct reciprocal for the interval operand. However, the disadvantage of the hardware support offered for reciprocal of an interval is that the possibility of 1 ulp error in each endpoint computation. For some cases, whose frequency is extremely low, if both endpoints are computed with 1 ulp error, then the range of the result interval would be 2 ulp wider compared to the interval result obtained using an infinite precision.

Interval arithmetic is extremely powerful to bound the range of functions. It is widely used in various computations and numerical applications in fields of science and engineering. The most time consuming operation is division in such applications. For example, the interval division operation takes fifty-seven clock cycles with the combined interval and floating-point divider. As a future direction, the presented reciprocal operation may perform division operation as well and may replace the division unit. It is possible to complete interval division operation in twenty-five clock cycles if the reciprocal result obtained with the proposed combined reciprocal unit is used in the subsequent multiplication operation, where the interval multiplication takes four clock cycles. With the possibility of utilizing the existent multiplier on the processor as to replace the multiplier employed in our design, the proposed design is inexpensive in terms of hardware.

Appendix A

REFINEMENT OF THE TABLE LOOK-UP COEFFICIENT

Minimization of the maximum absolute error is performed below for the computed power of an operand.

$$\epsilon = C \cdot X' - X^p \tag{A.1}$$

where, ϵ denotes the *error* in $C \cdot X'$ and X^p denotes the correct result for power of the operand.

Left hand side of Equation (A.1) can be rewritten as:

$$\epsilon = X'[C - X^p(X')^{-1}] \tag{A.2}$$

where;

$$C = (X_1 + 2^{-m-1})^{p-1} \tag{A.3}$$

$$X' = [X_1 + 2^{-m-1} + p \cdot (X_2 - 2^{-m-1})] \tag{A.4}$$

$$X = X_1 + X_2 \tag{A.5}$$

Rearranging Equation (A.2):

$$\begin{aligned} \epsilon = {} & X'[(X_1 + 2^{-m-1})^{p-1} \\ & -(X_1 + X_2)^p (X_1 + 2^{-m-1} + p \cdot (X_2 - 2^{-m-1}))^{-1}] \end{aligned} \tag{A.6}$$

General Binomial Theorem states that [81]:

$$(x+a)^r = \sum_{k=0}^{r} \binom{r}{k} x^k a^{r-k} \tag{A.7}$$

where;

$$\binom{r}{k} = \begin{cases} \frac{r(r-1)(r-2)...(r-k+1)}{k(k-1)(k-2)...1} & \text{, integer } k \geq 0 \\ 0 & \text{, integer } k < 0 \end{cases}$$

Applying Binomial Theorem to Equation (A.6):

$$\begin{aligned} \epsilon \;=\; & X'[((X_1)^{p-1}+(p-1)(X_1)^{p-2}(2)^{-m-1}+\tfrac{(p-1)(p-2)}{2!}(X_1)^{p-3}((2)^{-m-1})^2+...) \\ & -((X_1)^p+(p)(X_1)^{p-1}X_2+\tfrac{(p)(p-1)}{2!}(X_1)^{p-2}(X_2)^2+...) \\ & \cdot((X_1)^{-1}-(X_1)^{-2}(pX_2-(p-1)(2)^{-m-1})+(X_1)^{-3}(pX_2-(p-1)(2)^{-m-1})^2-...)] \end{aligned} \tag{A.8}$$

Rewriting Equation (A.8):

$$\begin{aligned} \epsilon \;=\; & X'[((X_1)^{p-1}+(p-1)(X_1)^{p-2}(2)^{-m-1}+(p-1)(p-2)(X_1)^{p-3}((2)^{-2m-3})+...) \\ & -((X_1)^p+(p)(X_1)^{p-1}X_2+\tfrac{(p)(p-1)}{2}(X_1)^{p-2}(X_2)^2+...) \\ & \cdot((X_1)^{-1}-(X_1)^{-2}pX_2+(X_1)^{-2}(p-1)(2)^{-m-1}+(X_1)^{-3}p^2X_2^2-(X_1)^{-3}p(p-1)X_2(2)^{-m}+(X_1)^{-3}(p-1)^2(2)^{-2m-2}...)] \end{aligned}$$

$$\begin{aligned} =\; & X'[(X_1)^{p-1}+(p-1)(X_1)^{p-2}(2)^{-m-1}+(p-1)(p-2)(X_1)^{p-3}((2)^{-2m-3})-(X_1)^{p-1}+(X_1)^{p-2}pX_2 \\ & -(X_1)^{p-2}(p-1)(2)^{-m-1}-(X_1)^{p-3}p^2(X_2)^2+(X_1)^{p-3}p(p-1)X_2(2)^{-m}-(X_1)^{p-3}(p-1)^2(2)^{-2m-2} \\ & -(X_1)^{p-2}(p)X_2+(X_1)^{p-3}p^2(X_2)^2-(X_1)^{p-3}p(p-1)X_2(2)^{-m-1}-\tfrac{p(p-1)}{2}(X_1)^{p-3}(X_2)^2+...] \end{aligned} \tag{A.9}$$

Many terms in Equation (A.9) cancel out. Neglecting the rest of the non-considered terms; simplified version of the error is obtained:

$$\epsilon \;\simeq\; X'\left[-(X_1)^{p-3}(p)(p-1)\cdot\left(\frac{(X_2)^2}{2}-(X_2)2^{-m-1}+2^{-2m-3}\right)\right] \tag{A.10}$$

Rearranging Equation (A.10):

$$\begin{aligned} \epsilon &\simeq X' \left[-(p)(p-1)(X_1)^{p-3}2^{-1} \cdot \left((X_2)^2 - (X_2)2^{-m} + 2^{-2m-2}\right)\right] \\ &\simeq X' \left[-(p)(p-1)(X_1)^{p-3}2^{-1} \cdot \left((X_2) - 2^{-m-1}\right)^2\right] \end{aligned} \tag{A.11}$$

Since X_2 is the rest of the mantissa, beginning from the $(m+1)^{th}$ bit, the above error can be approximated as:

$$\epsilon \simeq X' \left[-(p)(p-1)(X_1)^{p-3}(2)^{-1}(2)^{-2m-3}\right] \tag{A.12}$$

Finally, refinement of the table look-up coefficient with the complementary of the term in Equation (A.12), excluding the modified operand term, is performed:

$$\begin{aligned} C' &= C+(p)(p-1)(X_1)^{p-3}2^{-2m-4} \\ &= (X_1+2^{-m-1})^{p-1}+(X_1+2^{-m-1})^{p-1} \\ &= (X_1)^{p-1}+(p-1)(X_1)^{p-2}2^{-m-1}+\frac{(p-1)(p-2)}{2!}(X_1)^{p-3}2^{-2m-2}+...+(p)(p-1)(X_1)^{p-3}(2)^{-2m-4} \\ &\simeq (X_1)^{p-1}+(p-1)(X_1)^{p-2}(2)^{-m-1}+(X_1)^{p-3}(2)^{-2m-4}[p(p-1)+2(p-1)(p-2)] \end{aligned} \tag{A.13}$$

The obtained refined coefficient given in Equation (A.14) yields *one* or *two* more bit/bits accuracy for the initial approximation result:

$$C' = (X_1)^{p-1}+(p-1)(X_1)^{p-2}(2)^{-m-1}+(X_1)^{p-3}(2)^{-2m-4}[(p-1)(3p-4)] \tag{A.14}$$

Table look-up coefficient for reciprocation operation is provided by the general refinement formula, where $p = -1$,

$$C' = (X_1)^{-2} - (X_1)^{-3} \cdot 2^{-m} + (X_1)^{-4} \cdot 7 \cdot 2^{-2m-3} \tag{A.15}$$

BIBLIOGRAPHY

[1] [Online]. Available: http://www.hyperdictionary.com/computing/moore's+law

[2] A. Moshovos and G. Sohi, "Microarchitectural innovations: boosting microprocessor performance beyond semiconductor technology scaling," in *Proceedings of the* IEEE, vol. 89, no. 11, November 2001, pp. 1560–1575.

[3] R. W. Steward, R. Chapman, and T. Durrani, "The square root in signal processing," in *Proceedings of Real Time Signal Processing XII*, 1989, pp. 89–100.

[4] V. K. Jain, G. E. Perez, and J. M. Wills, "Novel reciprocal and square-root VLSI cell: Architecture and application to signal processing," in *International Conference on Acoustics, Speech and Signal Processing*, vol. 2, 1991, pp. 1201–1204.

[5] N. Takagi, "Generating a power of an operand by a table look-up and a multiplication," in *Proceedings of the 13th Symposium on Computer Arithmetic*, July 1997, pp. 126–131.

[6] S. Hollasch, IEEE *Standard 754 Floating Point Numbers.* [Online]. Available: http://stevehollasch.com/cgindex/coding/ieeefloat.html

[7] W. Kahan, *Lecture Notes on the Status of* IEEE *Standard 754 for Binary Floating-Point Arithmetic*, Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1996. [Online]. Available: http://www.cs.berkeley.edu/˜ wkahan

[8] *Floating-Point Numbers on* OS*/400,* IBM *Software Technical Document*, IBM Corp. [Online]. Available: http://www.ibm.com

[9] ANSI/IEEE *754-1985 Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers, New York, NY, 1985.

[10] C. Severance, "An interview with the old man of floating-point," February 1998. [Online]. Available: http://cch.loria.fr/documentation/IEEE754/wkahan/754story.html

[11] A. Akkas, "Instruction set enhancements for reliable computations," Ph.D. dissertation, Graduate Department of Computer Science, Lehigh University, August 2001.

[12] Y. He and C. Ding, "Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications." *Journal of Supercomputing*, vol. 18, pp. 259–277, 2001.

[13] E. M. Schwarz and C. A. Krygowski, "The S/390 G5 floating-point unit," *IBM J. RES. DEVELOPMENT*, vol. 43, no. 5/6, September/November 1999. [Online]. Available: http://www.research.ibm.com/journal/rd/435/schwarz.pdf

[14] J. Demmel, G. Henry, and W. Kahan, *Document for the Basic Linear Algebra Subprograms (*BLAS*) Standard*, Basic Linear Algebra Subprograms Technical (BLAST) Forum, December 1997.

[15] A. Akkas and M. J. Schulte, "A quadruple precision and dual double precision floating-point multiplier," in *Euromicro Symposium on Digital System Design*, 2003, pp. 76–81.

[16] 754R Working Group, *Some Proposals for Revising* ANSI/IEEE *Std 754-1985*, IEEE. [Online]. Available: http://754r.ucbtest.org

[17] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *Computing Surveys*, March 1991.

[18] J. A. Tupper, "Graphing equations with generalized interval arithmetic," Master's thesis, Graduate Department of Computer Science, University of Toronto, 1996.

[19] M. R. Santoro, G. Bewick, and M. A. Horowitz, "Rounding algorithms for IEEE multipliers," *Proc. of 9th Symposium on Computer Arithmetic*, pp. 176–183, 1989.

[20] N. Quach, N. Takagi, and M. Flynn, "On fast IEEE rounding," Stanford University, Computer Systems Laboratory, Dept. of Electrical Eng. and Computer Science, Tech. Rep. CSL-TR-91-459, January 1991.

[21] G. Even and W. J. Paul, "On the design of IEEE compliant floting point units," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 398–413, May 2000.

[22] G. Even and P.-M. Seidel, "A comparison of three rounding algorithms for IEEE floating-point multiplication," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 638–650, July 2000.

[23] W. Park, T. Han, S. Kim, and S. Yang, "A floating point multiplier performing IEEE rounding and addition in parallel," *Journal of Systems Architecture*, vol. 45, no. 14, pp. 1195–1207, July 1999.

[24] R. E. Moore, "Interval arithmetic and automatic error analysis in digital computing," Ph.D. dissertation, Stanford University, October 1962.

[25] C. F. Korn and C. Ullrich, "Verified solution of linear systems based on common software libraries," *Interval Computations*, vol. 3, pp. 116–132, 1993.

[26] C. F.Korn and C. Ullrich, "Extending linpack by verification routines for linear systems," *Mathematics and Computers in Simulation*, vol. 39, no. 1-2, pp. 21–37, 1995.

[27] H. Schwandt, "An interval arithmetic method for the solution of nonlinear systems of equations on a vector computer," *Parallel Computing*, vol. 4, no. 3, pp. 323–337, 1987.

[28] E. R. Hansen, *Global Optimization Using Interval Analysis.* Marcel Dekker, Inc., New York, 1992.

[29] A. Neumaier, *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, England, 1990.

[30] E. Hyvonen, "Constraint reasoning based on interval arithmetic," in *IJCAI-89 Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, N. Sridharan, Ed., 1989.

[31] G. F. Corliss and L. B. Rall, "Adaptive, self-validating numerical quadrature," *SIAM J. Sci. Statist. Comput.*, vol. 8, no. 5, pp. 831–847, September 1985.

[32] R. Lohner, *Computational Ordinary Differential Equations.* Oxford: Clarendon Press, 1992, ch. Enclosures for the Solutions of Ordinary Initial and Boundary Value Problems, pp. 425–435.

[33] M. Goehlen, M. Plum, and J. Schroder, "A programmed algorithm for existence proofs for two-point boundary value problems," *Computing*, vol. 44, pp. 91–132, 1990.

[34] H. J. Dobner and E. Kaucher, "Inclusion methods for integral equations," *Proceedings of the Third International IMACS-GAMM Symposium (SCAN-91), Amsterdam, Netherlands*, pp. 13–24, 1992.

[35] S. M. Hammel, J. A. Yorke, and C. Grebogi, "Do numerical orbits of chaotic dynamical processes represent true orbits?" *Journal of Complexity*, vol. 3, no. 2, pp. 136–145, 1987.

[36] G. V. Balaji and J. D. Seader, "Application of interval newton methods to chemical engineering problems," *Reliable Computing*, vol. 1, no. 3, pp. 215–223, 1995.

[37] S. P. Mudur and P. A. Koparkar, "Interval methods for processing geometric objects," *IEEE Computer Graphics and Applications*, vol. 4, no. 2, pp. 7–17, February 1984.

[38] R. B. Kearfott, "Interval computations: Introduction, uses, and resources," *Euromath Bull.*, vol. 2, pp. 95–112, 1996.

[39] Y. Watanabe, "Guaranteed error bounds for finite element solutions of the stokes equations," Ph.D. dissertation, Computer Center, Kyushu University, 1996.

[40] R. B. Kearfott and Z. Xing, "Rigorous computation of surface patch intersection curves," 1993, department of Mathematics, University of Southwestern Louisiana, Lafayette, LA.

[41] K. Okumura and S. Higashino, "A method for solving complex linear equation of ac networks by interval computation," in *Proceedings of IEEE International Symposiumon on Circuits and Systems - ISCAS '94*, 1994, pp. 121–124.

[42] V. N. Krishchuk, M. N. Vesilega, and G. L. Kosina, "The experience of applying interval calculation for the analysis of the strength characteristics of radio-electronic devices," 1992, dept. of Radio Designing and Manufacturing, Zaporozhue, Ukraine.

[43] J. Rohn, "An algorithm for checking the stability of symmetric interval matrices," *IEEE Transactions on Automatic Control*, vol. 41, no. 1, pp. 133–136, 1996.

[44] L. J. Kohout and I. Stabile, "Interval-valued inference in medical knowledge-based system CLINAID," *Interval Computations*, vol. 3, pp. 88–115, 1993.

[45] G. D. Hager, "Solving large systems of nonlinear constraints with application to data modeling," *Interval Computations*, vol. 2, pp. 169–200, 1993.

[46] M. E. Jerrell, *Applications of Interval Computations, Applied Optimization.* Dordrecht, Netherlands: Kluwer, 1996, vol. 3, ch. Applications of Interval Computations to Regional Economic Input-Output Models, pp. 133–143.

[47] S. Hadjihassan, E. Walter, and L. Pronzato, *Applications of Interval Computations, Applied Optimization.* Dordrecht, Netherlands: Kluwer, 1996, vol. 3, ch. Quality Improvement via Optimization of Tolerance Intervals During the Design Stage, pp. 91–131.

[48] D. Chiriaev and G. W. Walster, "Interval arithmetic specification," Sun Microsystems, Inc., Tech. Rep., 1998. [Online]. Available: http://www.mscs.mu.edu/˜ globsol/walster-papers.html/

[49] G. Heindl, "An improved algorithm for computing the product of two machine intervals," Department of Mathematics, University of Wuppertal, Germany, Tech. Rep. IAGMPI-9304, 1993.

[50] E. D. Popova, "Multiplication distributivity of proper and improper intervals," *Reliable Computing*, vol. 7, no. 2, pp. 129–140, 2001.

[51] J. W. von Gudenberg, "Hardware support for interval arithmetic - extended version," Information Instutition, University of Wurzburg, Tech. Rep. 125, October 1995.

[52] C. Hamzo and V. Kreinovich, "On average bit complexity of interval arithmetic," *Bulletin of the European Association for Theoretical Computer Science*, vol. 68, pp. 153–156, 1999.

[53] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers.* McGraw-Hill, 1998, vol. 3.

[54] I. Koren, *Computer Arithmetic Algorithms.* Prentice-Hall Inc., 1993, vol. 1.

[55] S. F. Oberman and M. J. Flynn, "An analysis of division algorithms and implementations," Departments of Electrical Engineering and Computer Science, Stanford University, Tech. Rep. CSL-TR-95-675, July 1995.

[56] G. Dahlquist, A. Bjorck, and N. A. eds., *Numerical Methods.* Prentice-Hall Inc., 1974.

[57] M. Ito, N. Takagi, and S. Yajima, "Efficient initial approximation for multiplicative division and square root by a multiplication with operand modification," *IEEE Transactions on Computers*, vol. 46, no. 4, April 1997.

[58] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM system/360 model 91: Floating-point execution unit," IBM *Journal*, pp. 34–53, January 1967.

[59] H. R. Srinivas and K. K. Parhi, "A fast radix 4 division algorithm."

[60] A. Nannarelli and T. Lang, "Power-delay tradeoffs for radix-4 and radix-8 dividers."

[61] D. L. Harris, S. F. Oberman, and M. A. Horowitz, "SRT division architectures and implementations."

[62] P. Montuschi and L. Ciminiera, "Design of a radix 4 division unit with simple selection table," *IEEE Transactions on Computers*, vol. 41, no. 12, pp. 1606–1611, December 1992.

[63] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 833–854, August 1997.

[64] P. Bannon and J. Keller, "Internal architecture of Alpha 21164 microprocessor," *Digest of Papers* COMPCON *'95*, pp. 79–87, March 1995.

[65] D. L. Fowler and J. E. Smith, "An accurate, high speed implementation of division by reciprocal approximation," Astronautics Corporation of America, 5800 Cottage Grove Rd. Madison, Wisconsin, 53716.

[66] R. E. Moore, *Interval Analysis.* Englewood Cliffs, NJ: Prentice-Hall Inc., 1966.

[67] R. B. Kearfott, M. Dawande, K. Du, and C. Hu, "Algorithm 737: INTLIB: A portable FORTRAN 77 interval standard function library," ACM *Trans. Math. Software*, vol. 20, pp. 447–459, 1994.

[68] O. Knuppel, "PROFIL/BIAS - a fast interval library," *Computing*, vol. 53, pp. 277–288, 1994.

[69] E. Hyvonen and S. D. Pascale, "InC++ library family for interval computations," *Supplement to the Internat, Workshop on Applications of Interval Computations, ed. V. Kreinovich, Internat. J. Reliable Comput.*, pp. 85–90, 1995.

[70] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, and C. Ullrich, PASCAL-XSC*: Language Reference with Examples.* Springer-Verlag, 1991.

[71] R. Klatte, U. Kulisch, A. Wiethoff, C. Lawo, and M. Rauch, C-XSC*: A C++ Class Library for Extended Scientific Computing.* Springer-Verlag, 1993.

[72] W. V. Valter, "ACRITH-XSC: A fortran-like language for verified scientific computing," in *Scientific Computing with Automatic Result Verification*, E. Adams and U. Kulisch, Eds. Academic Press, 1993, pp. 45–70.

[73] J. S. Ely, "The VPI software package for variable precision interval arithmetic," *Interval Computations*, vol. 2, pp. 135–153, 1993.

[74] M. Schulte, V. Zelov, A. Akkas, and J. Burley, "The interval-enhanced gnu fortran compiler," *Reliable Comput.*, no. 53, pp. 311–322, 1999.

[75] "Interval arithmetic in the forte^{TM} fortran 95 compiler." [Online]. Available: http://www.sun.com/forte/fortran/interval

[76] M. J. Schulte and E. E. Swartzlander, Jr, "A hardware design and arithmetic algorithms for a variable-precision, interval arithmetic coprocessor," in *Proceedings of the 12th Symposium on Computer Arithmetic*, July 1995, pp. 163–171.

[77] J. E. Stine, "Design issues for accurate and reliable arithmetic," Ph.D. dissertation, Graduate Department of Computer Science, Lehigh University, 2001.

[78] J. E. Stine and M. J.Schulte, "A combined interval and floating-point multiplier," *8th Great Lakes Symposium on VLSI*, February 1998.

[79] J. E. Stine and M. J. Schulte, "A combined interval and floating-point divider," *Thirty-Second Asilomar Conf. on Signals, Systems and Computers*, November 1998.

[80] A. Akkas, "A combined interval and floating-point comparator/selector," *IEEE 13th Internat. Conf. on Application-specific Systems, Architectures, and Processors*, pp. 208–217, 2002.

[81] [Online]. Available: http://mathworld.wolfram.com/BinomialTheorem.html

VITA

Umut Küçükkabak, the son of Mehmet and Berna Küçükkabak, was born in Burdur, Türkiye on May 27th, 1979. Between 1997-1998, he stayed in Ohio, U.S.A., as an American Field Service (AFS) Intercultural Exchange Student. In 2002, he received his Bachelor of Science degree in Electrical and Electronic Engineering from Middle East Technical University (METU - ODTÜ) in Ankara, Turkey. During his undergraduate studies, he completed his summer interns in 2000 and 2001, at Aselsan in Ankara. From September 2002 till June 2004, he worked as a teaching and research assistant at Koç University, İstanbul, Turkey, where he is currently pursuing his Master of Science degree in Electrical and Computer Engineering. During his Master of Science studies, he worked on the implementation and design of reciprocal unit, in which a table look-up procedure and Newton-Raphson iterations are employed.

PUBLICATIONS

- Umut Küçükkabak and Ahmet Akkaş, "Design and Implementation of Reciprocal Unit Using Table Look-up and Newton-Raphson Iteration" *Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04)*, Rennes, France, 31 August - 3 September, 2004, pages: 249 - 253.

CONTACT INFORMATION

- E-mail address : umut@kucukkabak.com