Dual-Mode Quadruple Precision Floating-Point Division and Square-Root Units

by

Aytunç İşseven

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

July, 2007

Koç University

Graduate School of Sciences and Engineering

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

Aytunç İşseven

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:

____

Assistant Prof. Ahmet Akkaş (Advisor)

____

Associate Prof. Atilla Gürsoy

____

Assistant Prof. Engin Erzin

Date: ______________________________

I dedicate this thesis to my mother, Ayşegül İşseven.

ABSTRACT

Many scientific applications require more accurate computations than double precision or double-extended precision floating-point arithmetic. This thesis presents the design of dual-mode quadruple precision floating-point division and square-root units that also supports two parallel double precision operations. A radix-4 SRT division and square-root algorithms are used to design the dual-mode quadruple precision floating-point division and square-root units. The implementation details of the divider presented in this thesis show how a conventional quadruple precision divider is modified and the datapath can be divided into two parts to support both a quadruple precision and two parallel double precision operations. To estimate area and worst case delay, a double, a quadruple, a dual-mode double, and a dual-mode quadruple precision floating-point division units are implemented in VHDL and synthesized. Similar to the dual-mode division unit, it is shown that how the datapath of conventional quadruple precision square-root unit is modified and can be divided into two parts to support both a quadruple precision and two parallel double precision operations. The correctness of all the designs was tested and verified through extensive simulation. The synthesis results show that the dual-mode quadruple precision divider requires 22% more area than the quadruple precision divider and the worst case delay is 1% longer. Also the dual-mode quadruple precision square-root unit requires 22% more area than the conventional quadruple precision square-root unit and the worst case delay is 2% longer. A quadruple precision division and square-root operations take fifty nine cycles and two parallel double precision division and square-root operations take twenty nine cycles.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Ahmet Akkaş for his endless support during the course of my graduate studies. What I grasped from him was not only his academic knowledge. But his way of having a great understanding, strong principles, work ethics and discipline were the things I feel more lucky to have a chance to encounter. Knocking on his door in the mornings is an exciting moment for me as it always used to be since the first day of my studies at KOÇ University.

I would also like to thank my teachers Prof. Murat Tekalp, Prof. Alper Demir, Prof. Serdar Taşıran, Prof. Engin Erzin, Prof. Yücel Yemez, Prof. Öznur Özkasap, Prof. Atilla Gürsoy and many others for their contributions to my profession during my graduate studies. I can not forget to mention my gratitude to my elementary school teacher Savaş Paksoy and my grandfather Erdinç İlkin as I believe they are the reason why I am an engineer.

My studies at Koç University not only helped me get better in my profession, but earned me life-time friends. I would like to thank my officemate Alp Arslan Bayrakçi for always being a brother-like friend in both the good and bad days. Thanks to Mehmet Emre Sargin, Engin Kurutepe, Ulaş Bağcı and Tayyar Önal and many others for wonderful times and memories.

Finally, I would like to thank my family for who I am and where I am. Without my father and mother I can not see myself having a graduate diploma. Since my kindergarten, I can say my mother Ayşegül, had education for the second time in her life by always being next to me. She left us alone in 1998 because of cancer but I always promised myself to continue her ideals in her life for me and my younger brother. Thanks to my father, Tuncay for always being a pushing force in everything I do in my life. My brother Çağrı, grandmother Canan and her mother Şükran, my uncle Can and her wife Yasemin, my cousins Erdinç and Ayşegül and my fiancèe Sena were as eager as me for my graduation and they motivated me a lot. I hope one day, my name will pass in Çağrı's, Erdinç's and Ayşegül's master thesis.

TABLE OF CONTENTS

List of Tables viii

List of Figures ix

Nomenclature x

Chapter 1: Introduction 1

1.1 The Need for Reliable Computing 1

1.2 Motivation 2

1.2.1 Dual-Mode 2

1.2.2 Quadruple Precision Operations 4

1.3 Dual-Mode Quadruple Precision SRT Radix-4 Division Overview 5

1.4 Dual-Mode Quadruple Precision SRT Radix-4 Square-Root Overview 5

1.5 Contribution 6

1.6 Outline 7

Chapter 2: Background 8

2.1 Floating-Point Numbers 8

2.2 Rounding Errors in Floating-Point Arithmetic 12

2.3 Rounding Modes in IEEE-754 Floting-Point Arithmetic 13

2.4 Digit-Recurrence Division Algorithms 14

2.4.1 SRT Division 14

2.5 SRT Square-Root Algorithm 20

Chapter 3: Dual-Mode SRT Radix-4 Division Unit 24

3.1 Conventional SRT Radix-4 Division Unit 24

3.1.1 Quotient Digit Set 24

3.1.2 Quotient Digit Selection Function . . . 26
3.1.3 Main Recurrence and Carry-Save Adder . . . 27
3.1.4 On-the-Fly Conversion and Rounding . . . 33
3.2 Dual-Mode SRT Radix-4 Quadruple Precision Division Unit . . . 34
3.2.1 Main Recurrence . . . 36
3.2.2 Generating Divisor Multiples . . . 39
3.2.3 On-the-Fly Rounding and Conversion . . . 39
3.3 Comparison of Synthesis Results . . . 41

Chapter 4: Dual-Mode SRT Radix-4 Square-Root Unit **42**
4.1 Conventional SRT Radix-4 Square Root Unit . . . 42
4.1.1 Main Recurrence and Carry-Save Adder . . . 42
4.1.2 Result-Digit Selection Function . . . 43
4.1.3 F-Generation Unit . . . 45
4.1.4 On-the-Fly Conversion and Rounding . . . 46
4.2 Dual-Mode Quadruple Precision SRT Radix-4 Square-Root Unit . . . 49
4.2.1 Main Recurrence . . . 50
4.2.2 F-Generation and On-the-Fly Conversion and Rounding . . . 51
4.3 Comparison of Synthesis Results . . . 53

Chapter 5: Conclusion **55**

Chapter 6: Appendix A **56**

Chapter 7: Appendix B **70**

Bibliography **84**

Vita **89**

LIST OF TABLES

2.1 Maximum Rounding Errors with Rounding to Nearest (Even) Mode 13

2.2 Possible Radix-4 Quotient Digit Sets 18

3.1 Area and Delay Estimates. 41

4.1 F Generation Scenarios 48

4.2 Area and Delay Estimates. 53

LIST OF FIGURES

2.1 IEEE 754 Single Precision Floating-Point Number 9
2.2 IEEE 754 Double Precision Floating-Point Number 10
2.3 IEEE 754 Quadruple Precision Floating-Point Number 12
2.4 Derivation of SRT Formula Using Paper-Pencil Division Method 15
2.5 SRT radix-4 a=2 P-D Plot 19

3.1 SRT Radix-4 Divisor Multiple Generation Unit 25
3.2 Sign and Zero Detection Network 28
3.3 PLA used for Quotient Digit Selection 29
3.4 Main Recurrence Unit for SRT radix-4 Division 31
3.5 On-the-Fly Conversion and Rounding Unit 32
3.6 On-the-Fly Conversion's Update Scenarios 34
3.7 Step by step On-the-Fly Conversion Example 35
3.8 Dual-Mode Quadruple Precision Division Unit 37
3.9 Divisor Multiple Generation Unit for Dual-Mode Quadruple Precision Division 38
3.10 OTF Conversion and Rounding Unit for Dual-Mode Quadruple Precision Division 40

4.1 Square-Root Main Unit 44
4.2 Square-Root S_hat Generation Logic 45
4.3 A Combined F generation and OTF Unit for Square-Root 47
4.4 Square-Root Result from the OTF 49
4.5 Dual-Mode Square-Root Residual and Result-Digit Generation Unit 52
4.6 Dual-Mode Square-Root F-Generation Unit Register Extension Principle . . . 54

NOMENCLATURE

CPU	Central Processing Unit
FPU	Floating-Point Unit
FGEN	F-Generation Unit
ILP	Instruction Level Parallelism
OTF	On-the-Fly Conversion and Rounding
CSA	Carry-Save Adder
QDS	Quotient Digit Selection
QUAD	Quadruple Precision

Chapter 1

INTRODUCTION

1.1 The Need for Reliable Computing

The speed of digital computers is increasing as a result of advances in VLSI technology and innovative computer design. Increasing computer speeds provide the ability to perform trillions of arithmetic operations per second [1]. Although there have been significant advances in VLSI technology, the precision and arithmetic used in floating-point operations has not changed over the past two decades. Today, most modern processors support IEEE double and/or double-extended precision floating-point arithmetic [28], which was defined in 1985 [29, 48].

Despite the improvement on the speed of the arithmetic computations, the precision obtained has not improved over the past two decades. Today, most modern processors have hardware support for double precision (64-bit) or double-extended precision (typically 80-bit) floating-point arithmetic operations. However, double and double-extended precision are not enough for many scientific applications including climate modeling [5], computational physics [6], and computational geometry [6].

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 dot product of the following matrices

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

yields the result

$$A \bullet B = 0$$

using IEEE-754 double precision floating-point arithmetic. However, the correct result is as follows:

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

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

The accuracy and reliability of numerical computations can be increased using extended precision arithmetic. For example, quadruple precision arithmetic increases the accuracy and reliability of numerical computations by providing floatingpoint numbers that have more than twice the precision of double precision numbers. To be exact, the accuracy of double precision numbers is roughly fifteen decimal digits. On the other hand, accuracy of quadruple precision floating-point numbers is about thirty-three decimal digits. This comes in handy in applications such as computational fluid dynamics and physical modeling, which require accurate numerical computations. The use of quadruple precision arithmetic can greatly improve the numerical stability and reproducibility of many of these applications.

1.2 Motivation

1.2.1 Dual-Mode

A processor can be preferred by many users to satisfy different needs. For example, as of today, a computer games programmer would most probably be not too much interested in

the precision of the FPU in a processor. On the other hand, a researcher who is trying simulate the behavior of some certain atmospherical event happening hundreds of years in the future, will importance to the FPU precision being high. Having these different needs, one can conclude that even for home PC users, specific purpose processors would be a necessity. Considering the current state of commercial market for processors, such a thought is not too practical. However, a way to address a specific processor to people with totally different precision needs would be to design one with a handle capacity of multiple precision modes [50, 49].

Current processors can handle both single and double precision operations in hardware but they are not considered as *Dual-Mode*. The reason is that such FPU's hardware is designed to operate only in exact double precision. When single precision operation is required, FPU pads twenty-nine zeros at the end of the single precision mantissa and convert the exponent to its double precision equivalent. After computing the result in double precision, the result exponent will be converted back to single precision and twenty-nine bits will be dropped from the least significant bits of the mantissa. A dual-mode double precision FPU supports both one double precision operation or two single precision operations in parallel. When software requests a single precision operation, dual-mode FPU sets the mode flag to single and works just like two conventional single precision FPUs. With a small amount of compromise in area, the throughput can be doubled when working in lower precision mode.

For a summation operation, just like any other basic operations, precision of the floating-point numbers affects the overall delay directly. Consider a very simple addition algorithm where carrying out happens in linear *O(n)* time where n is the number of bits. This algorithm is implemented on a conventional double precision FPU and a conventional single precision FPU. Also consider software requests two single precision numbers to be summed. In this case, lower(single) precision FPU has an advantage since higher precision FPU starts to add padded zeros when single precision FPU is finished with the request. Carrying-out of the lower precision FPU's operands finish half way through which results in nearly doubling the overall speed of the operation.

It is understood from above that, when a low precision operation is requested, lower-precision FPU design overruns the higher precision design. In the same scenario, now lets

compare conventional single precision FPU and a dual-mode double precision FPU. Since dual-mode double precision FPU can be considered as two conventional single precision FPUs, dual-mode design can do twice the amount of single precision FPU design. In these terms, dual-mode double precision FPU, when single precision operations is requested, can roughly be at least two times faster than a conventional double precision FPU. This great delay advantage of dual-mode design over the conventional only brings a twenty percent area increase disadvantage which seems acceptable. In the case where a double precision operation is requested, conventional double precision and dual-mode double precision designs work nearly identical. Basically, "why lose extra time when there is no need for high precision" is the motivation behind the dual-mode design.

1.2.2 Quadruple Precision Operations

Double and double-extended precision are not enough for many scientific applications including climate modeling [5], computational physics [6], and computational geometry [6]. Since floating-point arithmetic provides finite accuracy, it can lead to fatal errors after several consecutive operations. Main components of these errors are catastrophic cancelation and round-off errors. The higher accuracy (precision) of the operands are, the less errors will be in magnitude. Due to the advantages of quadruple precision arithmetic in scientific computing applications, specifications for quadruple precision numbers are being added to the revised version of the IEEE 754 Standard for Floating-Point Arithmetic [7, 31, 33, 47].

Quadruple precision operations are usually supported by software, such as on Sun's Sparc processors. Software support, however, has performance problems for numerically intensive applications. Therefore, hardware support for quadruple precision arithmetic is essential. The most modern microprocessors provide multiple identical functional units to speed up numerical computations [8]. For example, high performance processors Power4 and Sparc have identical two floatingpoint units for addition and multiplication operations [9, 10]. Another trend in microprocessor architectures is to have wide 128-bit internal datapaths [8, 11], which can support quadruple precision operands.

1.3 Dual-Mode Quadruple Precision SRT Radix-4 Division Overview

Although floating-point division is not the most frequent operation among floating-point arithmetic operations, the speed of it is important for the overall speed of the FPU especially because it is the most time consuming floating-point operation [36, 40, 41, 43] and it is difficult to pipeline. Significant research has been done to develop efficient floatingpoint division algorithms [2, 3, 16, 17], but there is very limited contributions to literature for dualmode hardware implementation of extended precision floating-point arithmetic (i.e., quadruple precision). Recently, dual-mode floating-point multipliers [12, 13] and adders [14, 15] have been designed.

Chapter 3 shows how a datapath of quadruple precision floating-point divider implemented with the SRT radix-4 algorithm can be split into two parts in order to support both one quadruple precision and two parallel double precision division operations. The technique and modifications used to design the dual-mode quadruple precision divider was also applied to implement a dualmode double precision divider that supports both one double precision and two parallel single precision operations. Both the dualmode double and quadruple precision division units were tested for four different rounding modes specified by IEEE 754 Standard.

1.4 Dual-Mode Quadruple Precision SRT Radix-4 Square-Root Overview

The implementation of square-root is very similar to division. Most of the units are identical and a look-up table used for the division can also be used by the square-root unit. Having all these common points, building a dual-mode square-root unit, unlike the division unit, was not too much challenging.

Although floating-point square-root is not the most frequent operation among floating-point arithmetic operations, the speed of it is important for the overall speed of the FPU especially because it is the most time consuming floating-point operation along with division and is hard to pipeline [36]. Especially in DSP applications which use single precision floating-point arithmetic, speed of the square-root operation is essential as stated in [20, 21]

Chapter 4 shows how a datapath of quadruple precision floating-point square-root unit implemented with the SRT radix-4 algorithm can be split into two parts in order to support

one quadruple precision and two parallel double precision square-root operations. Both the dualmode double and quadruple precision square-root units were tested for four different rounding modes specified by IEEE 754 Standard.

1.5 Contribution

In this thesis, my contributions are as follows:

- SRT algorithm is examined in detail in order to use the smallest number of bits (6) for the residual (which enters in the PLA or lookup table). Found that using the smallest number of bits of the residual is not optimizing the whole design. Decided to use seven bits of the residual and three bits of the divisor which resulted in the smallest table (PLA).

- An OpenGL code is written to visualize the PD-Plot (table or PLA) in order to debug quotient digit selection (QDS) function for correctness.

- SRT algorithm requires a table look-up system (PLA) in order to calculate quotient bits. Several tables were created using espresso. The one having the best area to delay ratio is embedded in to the division and square-root unit designs.

- After grasping the algorithms for division and square-root, I coded a division and a square-root simulator in *C* to make future VHDL debugging easier.

- Quadruple precision division unit is modified and extended in order to design dual-mode quadruple precision division unit. The correctness of the quadruple precision division units are verified by the *C* simulator.

- Double precision division unit is modified and extended in order to design dual-mode double precision division unit.

- Quadruple precision square-root unit is extended to dual-mode quadruple precision square-root unit. The correctness of the dual-mode quadruple precision square-root units are tested by the *C* simulator.

- Double precision square-root unit is extended to design dual-mode double precision square-root unit.
- All designs are implemented in VHDL, simulated in ModelSim and synthesized in Leonardo to determine the clock cycle time and number of gates for each unit.

1.6 Outline

The outline for this thesis is as follows: Chapter 2 gives the required background for floating-point arithmetic, common division algorithms and table look-up logic. Chapter 3 presents the conventional SRT radix-4 double precision division first and then goes in detail to quadruple precision dual-mode design implementation. Furthermore, the area and delay estimates of the implementations are also compared in this chapter. Similarly, Chapter 4 presents the conventional SRT double precision radix-4 square-root unit first and then goes into the implementation details of dual-mode quadruple precision design. Area and delay estimates of the implementations are also stated and compared in this chapter. Chapter 5 gives our final conclusions. Appendix A and B includes simulation results of the *C* code and VHDL simulations of division and square-root units respectively.

Chapter 2

BACKGROUND

In this chapter, floating-point, rounding modes, SRT division and square-root algorithms are 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.

2.1 Floating-Point Numbers

Among several representations, floating-point is the most commonly utilized representation to approximate real numbers on computers. Floating-point representation basically represents real numbers in 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 1.17×10^{-38} to 3.40×10^{38} for single precision and 2.22×10^{-308} to 1.79×10^{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.

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 [29, 30, 31, 36].

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 bits represent the exponent bits. The remaining 23 bits are the fraction bits, in other words Mantissa.

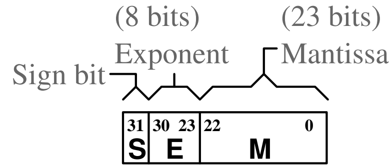

Figure 2.1: IEEE 754 Single Precision Floating-Point Number

The sign bit, which serves to identify whether the represented real number is 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) one, 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 M is nonzero, then V = NaN (Not a Number)
- If E = 255 and M is zero and S = 1, then V = $-\infty$
- If E = 255 and M is zero and S = 0, then V = ∞
- If $0 < E < 255$ then $V = (-1)^S \times 2^{E-127} \times (1.M)$, where; 1.M is intended to represent the binary number created by prefixing the fraction part with an implicit leading one (hidden one), and a binary point (.), as well the exponent bias being set to 127.
- If E = 0 and M is nonzero, $V = (-1)^S \times 2^{-126} \times (0.M)$, where these are defined as denormalized numbers.
- If E = 0 and M is zero and S = 1, then V = -0
- If E = 0 and M is zero and S = 0, then V = 0

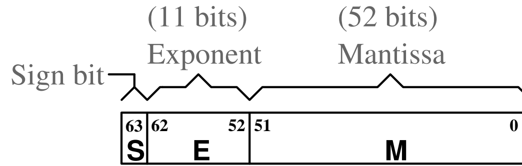

Figure 2.2: IEEE 754 Double Precision Floating-Point Number

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. Arithmetic 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 bits are the exponent bits. The remaining 52 bits are the fraction bits, in other words 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, 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 M is nonzero, then V = NaN (Not a Number)

- If E = 2047 and M is zero and S = 1, then V = $-\infty$
- If E = 2047 and M is zero and S = 0, then V = ∞
- If $0 < E < 2047$ then $V = (-1)^S \times 2^{E-1023} \times (1.M)$, where; 1.M 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 M is nonzero, $V = (-1)^S \times 2^{-1022} \times (0.M)$, where these are defined as denormalized numbers.
- If E = 0 and M is zero and S = 1, then V = -0
- If E = 0 and M 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 number. However, many scientific computing applications, such as computational geometry, computational physics and climate modeling need more precise arithmetical operations [11]. 128-bit quadruple precision arithmetic significantly improves the numerical stability of those scientific applications as stated in [5]. A quadruple precision number stated by the IEEE-754 standarts is shown in Figure 2.3. 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 arithmetic operations are at least two slower than double precision in hardware [14]. Much faster quadruple precision floating-point multiplier is presented in [27] 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 [28].

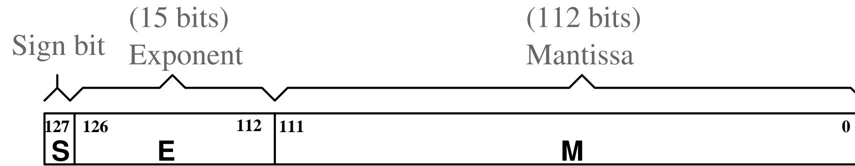

Figure 2.3: IEEE 754 Quadruple Precision Floating-Point Number

2.2 Rounding Errors in Floating-Point Arithmetic

It is impossible to represent some real numbers by IEEE-754 floating-point due to two reasons [17, 36]:

- 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 below example.

$$1.10011001100110011001100 \times 2^{-4} = 0.099999999403953552$$

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, then the error is stated as 0.1 *ulp*. Another measure of a rounding error is relative error. 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

Precision Type	Max. Rounding Error (0.5 *ulp*)
Single Precision	0.596046447754e-07
Double Precision	0.1110223024625156541e-15
Quadruple Precision	0.4814824860968089632639944 5e-34

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

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.

2.3 Rounding Modes in IEEE-754 Floting-Point Arithmetic

Four rounding modes are defined in IEEE-754 Floating-Point Arithmetic Standard [9, 32, 34, 36].

- 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 significant bit) 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 IEEE754 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, which 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].

2.4 Digit-Recurrence Division Algorithms

In digit-recurrence division methods, quotient is represented in a radix-r form and one digit of it is obtained per iteration. The radix determines the number of cycles necessary to complete the correct division. Redundant quotient-digit-set and residual format are usually favored in digit-recurrence algorithms since they have significant speed and area advantages. Digit-recurrence algorithms consist of n iterations of a recurrence, in which each iteration produces one digit of the quotient, the most-significant digit first [2]. The most popular digit-recurrence division algorithm implemented on modern processors is SRT algorithm [35, 37, 38, 41, 43, 45, 46, 49].

2.4.1 SRT Division

SRT division is named after Sweeney, Robertson[3] and Tocher[4], each of whom developed it independently at around the same time. Basically SRT division is very similar to the paper-pencil method for division as seen in Figure 2.4. It is composed of an iterative algorithm. The cycle time of each iteration affects the overall delay time of a unit. At the first iteration, the left most bit of the quotient and at the last iteration, the right most bit of the quotient is generated. If paper-pencil method was directly copied to the division algorithm, it would require for the quotient-bit to be multiplied with the divisor and then to be subtracted from the dividend at every iteration. Multiplying the divisor with a non-power of two quotient-bit could take long amount of time. Also another time consuming full-precision subtraction operation is required at every iteration if direct paper-pencil algorithm was implemented [36, 41].

The most advantageous property of SRT division is that, it can do these time consuming multiplication and subtraction operations independent of the precision of the operands. Four

Figure 2.4: Derivation of SRT Formula Using Paper-Pencil Division Method

properties that lead to this advantage are the iterative formula, redundant quotient-digit-set, redundant remainder format and on-the-fly conversion and rounding algorithm. The precision of the operands only affects the number of iterations of the algorithm, not the duration of each iteration.

Another difference of SRT division from the direct paper-pencil method is that it could be possible to have more than one valid quotient-bit in each iteration. With the help of this flexibility, the precision of the lookup-table (PLA) can be dropped from full-precision ($2^{52} \times 2^{52}$ for double-precision numbers) to 10 bits ($2^3 \times 2^7$). And this leads to a great benefit of area and delay. The derivation of the SRT algorithms formula is given below:

x - Dividend

d - Divisor

Q[j] - Quotient(at the jth iteration)

q_j - j^{th} quotient-bit

Rem[j] - Remainder(at the jth iteration)

n - Precision of the operands(also number of iterations)

r - Radix

The final result of a division must satisfy:

$$\frac{x}{d} - Q[n] = rem[n] < r^{-n}$$

So in every middle step, this inequality must hold:

$$\frac{x}{d} - Q[j] = rem[j] < r^{-j}$$

Multiplying both sides by d yields:

$$x - d \times Q[j] < d \times r^{-j}$$

Let's choose the reminder, $W[j]$ as:

$$W[j] = r^j \times (x - d \times Q[j])$$

According to the above statement, two important properties:

$$W[j] < d$$
$$W[0] = x$$

In order to have a recursion independent of x, $W[j+1]$ and $r \times W[j]$ are created.

$$\begin{array}{rr} & W[j+1] = r^{j+1} \times (x - d \times Q[j+1]) \\ - & r \times W[j] = r^{j+1} \times (x - d \times Q[j]) \\ \hline \end{array}$$

$$W[j+1] = r \times W[j] - d \times \underbrace{r^{j+1} \times (Q[j+1] - Q[j])}_{q_{j+1}}$$

The subtraction of those yields the main reccurence of SRT algorithm. The simplified last version is below:

$$W[j+1] = r \times W[j] - d \times q_{j+1} \tag{2.1}$$

The SRT radix-r division algorithm performs $\frac{n}{log_2(r)}$ division steps, each corresponding to one iteration of the final recurrence. Moreover, each iteration consists of four subcomputations as follows [2]:

1. One digit (could be several bits, to be exact: $log_2(r)$ bits) arithmetic left-shift of $W[j]$ to produce $r \times W[j]$.

2. Determination of the quotient digit q_{j+1} by the quotient-digit selection function.

3. Generation of the divisor multiple $d \times q_{j+1}$; and

4. Subtraction of $d \times q_{j+1}$ from $r \times W[j]$

Equation 2.1 is the main recursion of the SRT division. To summarize, r (which is a power of two) and $W[j]$ (previous remainder) are multiplied, which can be implemented by only a shift operation. Next, divisor will be multiplied by the quotient-digit selected from the table (q_{j+1}) and the result is subtracted from the $r \times W[j]$. If the quotient-digit-set is composed of numbers that are power of two, the multiplication of the divisor can be implemented with fast shift operations. Because the remainder is in redundant (Carry-Save) form, the subtraction can be managed in independent time of the precision of the operands. The magnitude of the final remainder, usually, is not very important. But the polarity of the remainder is important since it determines for the algorithm whether to run for an extra iteration or not. To get the polarity of the redundant final remainder, it is not needed to convert the number to its conventional form. There is a smart mechanism developed for finding the polarity of a redundant number without converting it to its conventional form. When the q_{j+1}s generated at every iteration are put next to each other, final quotient does not make a meaningful number. The reason is that q_{j+1}'s are in redundant form and need to be converted to conventional format. After q_{last} is found, it is a very time consuming operation to convert $Q[n]$ to its conventional form and round it. The reason for the time consuming conversion is that positive and negative components of $Q[n]$ needs to be separated and summed together. Where else, on-the-fly conversion and rounding algorithm basically does the conversion and rounding not after q_{last} is found, but instead it generates the converted $Q[n]$ during the division process continues and while q_{j+1}'s are being found. This way, by the time q_{last} is generated, converted form of $Q[n]$ is ready as well. This means a full-precision addition is being avoided at each iteration.

Radix-r, in a way, is simulating a base-r number using base-2 digits. There can be more than one valid combination for simulating a radix-r number. Because there is no uniqueness, these kind of numbers are called *Redundant*. A radix-r digit-set consists of symmetric positive and negative numbers. The negative number symbols are showed with a bar on the top. The greatest radix-r digit can be in between $a = r - 1$ and $a = \frac{r}{2}$, inclusive as shown in Table 2.2. For the radix-4 and a=2 case, two numbers having different digits

a	Quotient Digit Set	Property
2	$\{\bar{2}, \bar{1}, 0, 1, 2\}$	Minimally Redundant
3	$\{\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3\}$	Maximally Redundant

Table 2.2: Possible Radix-4 Quotient Digit Sets

can have the same value as shown below. If a was equal to 3, there would be many more distinct numbers having the same values than a to be equal to 2.

$$1\bar{1}\bar{2} = 1 \times 16 - 1 \times 4 - 2 \times 1 = 10$$
$$022 = 0 \times 16 + 2 \times 4 + 2 \times 1 = 10$$

This property of the redundant numbers leads to a remarkable benefit in SRT division. For generation of q_{j+1}, the number of bits to extract from $W[j]$ (remainder) drops from full precision to just some small number of left most bits. As the radix increases this small number gets bigger but never approaches the full precision. For the case of $r = 4$ and $a = 2$ only left most 7 bits of $W[j]$ and left most 3 bits of the divisor are enough. These two numbers are outcomes of quotient digit selection function derivations. At the end, these derivations can be visually shown in graphics called P-D Plots. First, a certain P-D plot is decided according to the *radix* and a choices, and then it is implemented in a PLA. So in the digital design, a small PLA is doing the Quotient-digit-selection functions job (table look-up service). As seen in Figure 2.5, different colored boxes correspond to different quotient-digits. The pair of neighboring diagonal lines around the color separation area defines the overlap regions. These overlap regions are directly the outcome of the use of redundant quotient-digit-set. The quotient-digit-selection can be made with any two values that are touching the overlap area. This flexibility leads the P-D plot to be quantized with bigger terms, hence letting it to reserve less space. For instance, the $y - axis$ of the P-D plot would have been divided in to 2^{52} if no overlap regions existed. Instead, the $y - axis$ is only divided in to $2^7 = 128$ pieces.

Using conventional formats, a full-precision subtraction operation takes long time. But if one of the operands of the subtraction is kept in redundant form (i.e. , carry-save format),

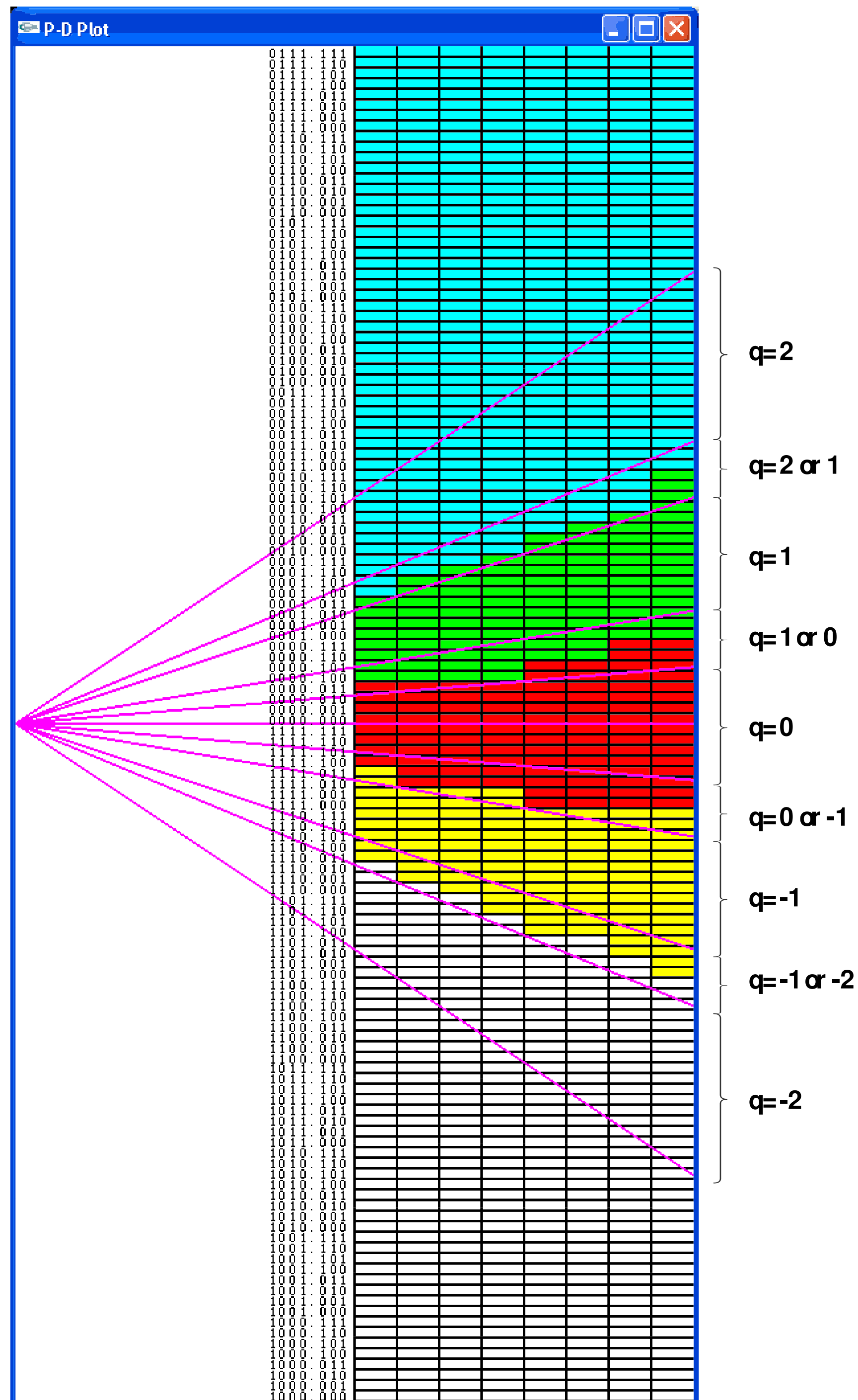

Figure 2.5: SRT radix-4 a=2 P-D Plot

then along with the other operand, all three can be summed with a 3-to-2 carry-save adder. Doing this saves us from a full-precision addition at every cycle, but brings an extra cost. The cost is doubling the hardware used for generating the subtraction operand which is decided to be in redundant form. The operand having less amount of hardware associated with it, will be the smarter choice for keeping in carry-save format. In SRT division, it is the $r \times W[j]$ since this component's generation only requires a full-precision multiplexor and a full-precision shifter. The two components of $r \times W[j]$: $r \times WC[j]$ and $r \times WS[j]$ along with $-d \times q_{j+1}$ will enter in to the 3-to-2 CSA to output $WC[j+1]$ and $WS[j+1]$. When $-d$ needs to be multiplied with a positive q_{j+1}, we need to take two's complement of the result. This requires for $1 - ulp(r^{-n})$ to be added to the inverted result which means a full-precision addition, but it can be avoided by a nice trick. $-d \times q_{j+1}$ will be entered in to the CSA along with $r \times WC[j]$ and $r \times WS[j]$ and one thing to note is the least significant bits of these two are always zero. The $1 - ulp$ which needs to be added to the inverted $d \times q_{j+1}$ can instead replace the *zero* at the least significant place of either one of the remainder's components. All these advantages make SRT division to be used in all our today's modern processors.

2.5 SRT Square-Root Algorithm

SRT square-root algorithm is composed of an iterative method. The cycle time of each iteration affects the overall delay time of the unit. At the first iteration, the left most bit of the result and at the last iteration, the right most bit of the result is generated [36, 46]. The formula derivation is as follows:

x - Number to be square-rooted

s - Square root of x

S[j] - Partial Result (at the jth iteration)

s_j - j^{th} Square root bit

$\epsilon[j]$ - Error (at the jth iteration)

n - Precision of the operands (also number of iterations)

r - Radix

$$s = \sqrt{x}, \qquad \frac{1}{4} \leq x < 1, \qquad \frac{1}{2} \leq s < 1$$

Final result $S[n]$ is composed of square root bits $(s_0 s_1 \cdots s_n)$. Mathematically this yields:

$$s = S[n] = \sum_{i=0}^{n} s_i \times r^{-i} \qquad s_0 = 1$$

The final result of a square root operation must satisfy;

$$|x^{\frac{1}{2}} - S[n]| = \epsilon[n] < r^{-n}$$

So in every middle step, this inequality must hold;

$$|x^{\frac{1}{2}} - S[j]| = \epsilon[j] < r^{-j}$$

Taking the absolute value out;

$$-r^{-j} < x^{\frac{1}{2}} - S[j] < r^{-j}$$

Leave x alone in the inequality;

$$S[j] - r^{-j} < x^{\frac{1}{2}} < r^{-j} + S[j]$$

Square all sides to get rid of the power of x;

$$r^{-2j} - 2 \times S[j] \times r^{-j} + S[j]^2 < x < r^{-2j} + 2 \times S[j] \times r^{-j} + S[j]^2$$

Take $S[j]^2$'s to the middle;

$$r^{-2j} - 2 \times S[j] \times r^{-j} < x - S[j]^2 < r^{-2j} + 2 \times S[j] \times r^{-j}$$

Pick partial residual $W[j]$ as follows:

$$W[j] = r^j \times (x - S[j]^2)$$

According to the above statement, an important property to keep in mind:

$$W[0] = x - S[0]^2$$
$$W[0] = x - 1$$

Plugging $W[j]$ back to the inequality:

$$r^{-2j} - 2 \times S[j] \times r^{-j} < \tfrac{W[j]}{r^j} < r^{-2j} + 2 \times S[j] \times r^{-j}$$

Multiply all sides with r^j to get a bound on the partial residual;

$$r^{-j} - 2 \times S[j] < W[j] < r^{-j} + 2 \times S[j]$$

In order to have a recursion independent of x, $W[j+1]$ and $r \times W[j]$ are created.

$$\begin{array}{rr} & W[j+1] = r^{j+1} \times (x - S[j+1]^2) \\ - & r \times W[j] = r^{j+1} \times (x - S[j]^2) \\ \hline \end{array}$$

$$W[j+1] = r \times W[j] + r^{j+1} \times (S[j]^2 - S[j+1]^2)$$

In the above format, the recursion formula would be in no use since square operation to be implemented is hard. Using difference of two squares, above simplifies to:

$$W[j+1] = r \times W[j] + r^{j+1} \times (S[j] - S[j+1]) \times (S[j] + S[j+1])$$

Remember also that the next partial result is just the current result concatenated with next square-root bit.

$$S[j+1] = S[j] + s_{j+1} \times r^{-(j+1)}$$

Using this equality, recursion formula simplifies in to a more implementable format:

$$W[j+1] = r \times W[j] + r^{j+1} \times \underbrace{(S[j] - S[j+1])}_{-r^{-(j+1)} \times s_{j+1}} \times \underbrace{(S[j] + S[j+1])}_{2 \times S[j] + s_{j+1} \times r^{-(j+1)}}$$

Multiplying the new substitutes;

$$W[j+1] = r \times W[j] \underbrace{-(2 \times S[j] \times s_{j+1} + s_{j+1}^2 \times r^{-(j+1)})}_{F[j]} \tag{2.2}$$

What we have called $F[j]$ is now in a suitable form which is ready to be implemented by just doing inversions and concatenations. The simplified last version of the main recursion of square-root algorithm is below. As seen, it is very similar to the recursion of the division. Implementations of these two operations are very similar as you will be seeing in next two chapters.

$$W[j+1] = r \times W[j] + F[j] \tag{2.3}$$

So, to summarize this final recursion formula step by step:

1. An arithmetic left-shift of $W[j]$ by $log_2(r)$ digits to produce $r \times W[j]$.

2. Determination of the result digit s_{j+1} using the square-root digit selection function $Select$.

3. Formation of the adder input by shifts, negations and concatenations

$$F = -(2 \times S[j] \times s_{j+1} + s_{j+1}^2 \times r^{-(j+1)})$$

4. Addition of $r \times W[j]$ with F to produce $W[j+1]$. As in division, to have a fast iteration step, a redundant adder (CSA) is used for this addition. At the end, it will be necessary to convert the signed-digit form to 2's complement form by means of an on-the-fly conversion as discussed for division.

Chapter 3

DUAL-MODE SRT RADIX-4 DIVISION UNIT

3.1 Conventional SRT Radix-4 Division Unit

As stated in the previous chapter, SRT radix-4 division algorithm uses the following recurrence for each iteration:

$$W_{j+1} = 4W_j - d \times q_{j+1} \tag{3.1}$$

where, W_j is the partial reminder, or residual, W_0 is the dividend, d is the divisor, and q_j is the jth quotient bits [36].

The number of iterations for n-bit division takes roughly $\frac{n}{log_2(4)}$ stages. The quotient bits are selected by a quotient-digit selection (QDS) function, q_{j+1} = QDS($4W_k[j]$, d_m) where k and m are the number of left most bits to be extracted from $W[j]$ and d respectively. If the final residual W_n is negative, an additional correction step is required. In such a case, the divisor is added to the remainder and r^{-n} (1-ulp) is subtracted from final quotient. When the final quotient is updated, it is converted to the conventional representation. As a last step, the result is rounded. If on-the-fly conversion and rounding is used, then there is no need to spend extra time for correction step, conversion and rounding.

3.1.1 Quotient Digit Set

Using radix-4 leaves us with two choices for the digit set:

Maximally redundant: q_j: { -3, -2, -1, 0, 1, 2, 3 }

Minimally redundant: q_j: { -2, -1, 0, 1, 2 }

Minimally redundant digit set means an easier divisor multiple generation and more complex QDS function whereas maximally redundant digit set brings an ease on the QDS and more complex divisor multiple generation. Minimally redundant digit set is used in this radix-4 design to avoid multiplying by integer 3. Multiplying d with q_{j+1} in minimally redundant digit set case, only requires some arithmetic shifts and negations [42].

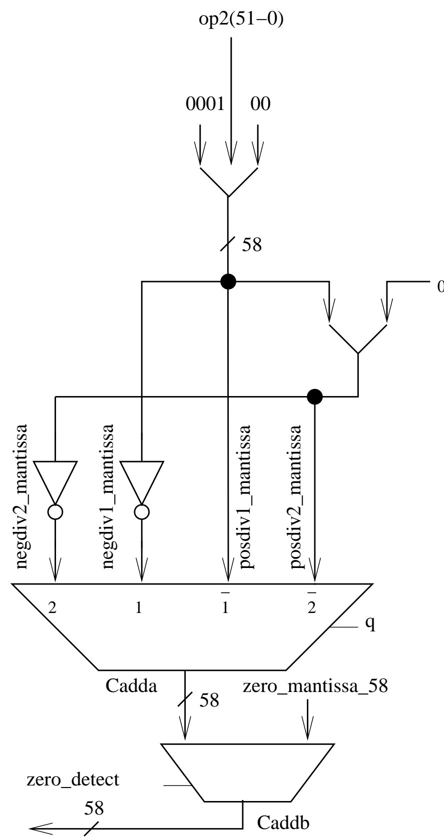

Figure 3.1: SRT Radix-4 Divisor Multiple Generation Unit

As seen in the Figure 3.1, the divisor's mantissa (posdiv1_mantissa) is multiplied by 2 (shifted left one bit) to get posdiv2_mantissa and then it is negated to get negdiv2_mantissa. In order to get negdiv1_mantissa, posdiv1_mantissa is just negated. For creating the negative of a two's complement number, addition by one is necessary along with the negation. Adding one to the negated number will take $O(n)$ time and a nice trick to avoid this will be shown in section 3.1.3.

3.1.2 Quotient Digit Selection Function

QDS function basically chooses the right quotient digit at every cycle. The inputs to this function are the residual and the divisor. Using a redundant digit set brings the advantage of feeding the QDS with fewer precision of $4W[j]$ and d. The overlap regions of the quotient digits in the P-D plot in Figure 2.5 allow this nice efficient structure to occur. The efficiency is so great that the precision of the residual to be used for QDS drops from 52 to 7 bits and the divisor from 52 to 3 bits. Also now knowing that only the 7 most significant bits of the residual are needed to be computed, it will be smart to use a carry-save adder for the addition of $4W[j] + d \times (-q_{j+1})$. Residual, by the use of the CSA, will be kept in twice the amount of the registers and are named $WC[j]$ and $WS[j]$, corresponding to carry and save components, respectively. The only full addition will be performed by a 7-bit carry-lookahead adder before result is fed into the QDS [44].

$$W[j] = WC[j] + WS[j]$$
$$WC[j+1] + WS[j+1] = 4WC[j] + 4WS[j] + d \times (-q_{j+1})$$

Keeping the residual in carry-save format will not make it possible to read in conventional form without a full-precision addition. Luckily the partial remainder nor the final remainder is needed to be in the conventional form for finalizing the division operation. Only the sign of the final remainder is needed along with whether it is zero or not. Final remainder, $WC[n]$ and $WS[n]$ will not be fed in to the recurrence cycle back again since the quotient-bits are all ready for conversion. However, if the sign of the final remainder $(WC[n] + WS[n])$ is negative, divisor needs to be added to make it positive. And also, final quotient needs to be decremented by $1 - ulp$. Adding the divisor to the negative remainder is not necessary since

having the true, positive remainder is useless in most cases. Decrementing the quotient is, of course, essential and very easy to do since $1 - ulp$ less quotient is ready in the OTF (On-the-Fly Conversion and Rounding) unit at any time. The real work is to understand if remainder is negative or not. In order to find that, a full-precision addition of the carry and save components of the final remainder is not necessary. A simple algorithm which requires much less hardware than a full-precision adder is shown in Figure 3.2. This implementation is for understanding whether $W[n]$ is positive or not. Final remainder's polarity will update the following values:

$$rem = \begin{cases} W[n] \times r^{-n} & if \quad W[n] \geq 0 \\ (W[n] + d) \times r^{-n} & if \quad W[n] < 0 \end{cases}$$

$$result = \begin{cases} Q[n] & if \quad W[n] \geq 0 \\ QM[n] = Q[n] - r^{-n} & if \quad W[n] < 0. \end{cases}$$

The table in Figure 3.3 shows the QDS function used for the conventional and dual-mode designs. The same QDS can be used for both double and quadruple precision divisions since both divisions use the same quotient digit set and in that case QDS is precision-independent.

3.1.3 Main Recurrence and Carry-Save Adder

Having the residual in redundant (carry-save) form saves great amount of time by avoiding a 52-bit full addition at every cycle of the division. CSA takes in $4WC[j]$, $4WS[j]$ and $d \times -q_{j+1}$ and outputs $WC[j+1]$ and $WS[j+1]$. The payback of this advantage is only doubling a multiplexer, arithmetic shifter (used to multiply $W[j]$ by 4). When the quotient bit is positive, the divisor multiple must be multiplied by a negative factor (-2 or -1). Previously shifting and negating has been explained but adding the necessary '1' for the correct complementing was left. The objective was to avoid a full 52-bit addition. Since two of the CSA's inputs are the partial residuals ($4WC[j]$, $4WS[j]$), it is obvious that the least significant two bits of these inputs are always zero. Placing '1' to the least significant bit of $4WC[j]$ when q_{j+1} is equal to 1 or 2 instead of adding '1' to the divisor multiple

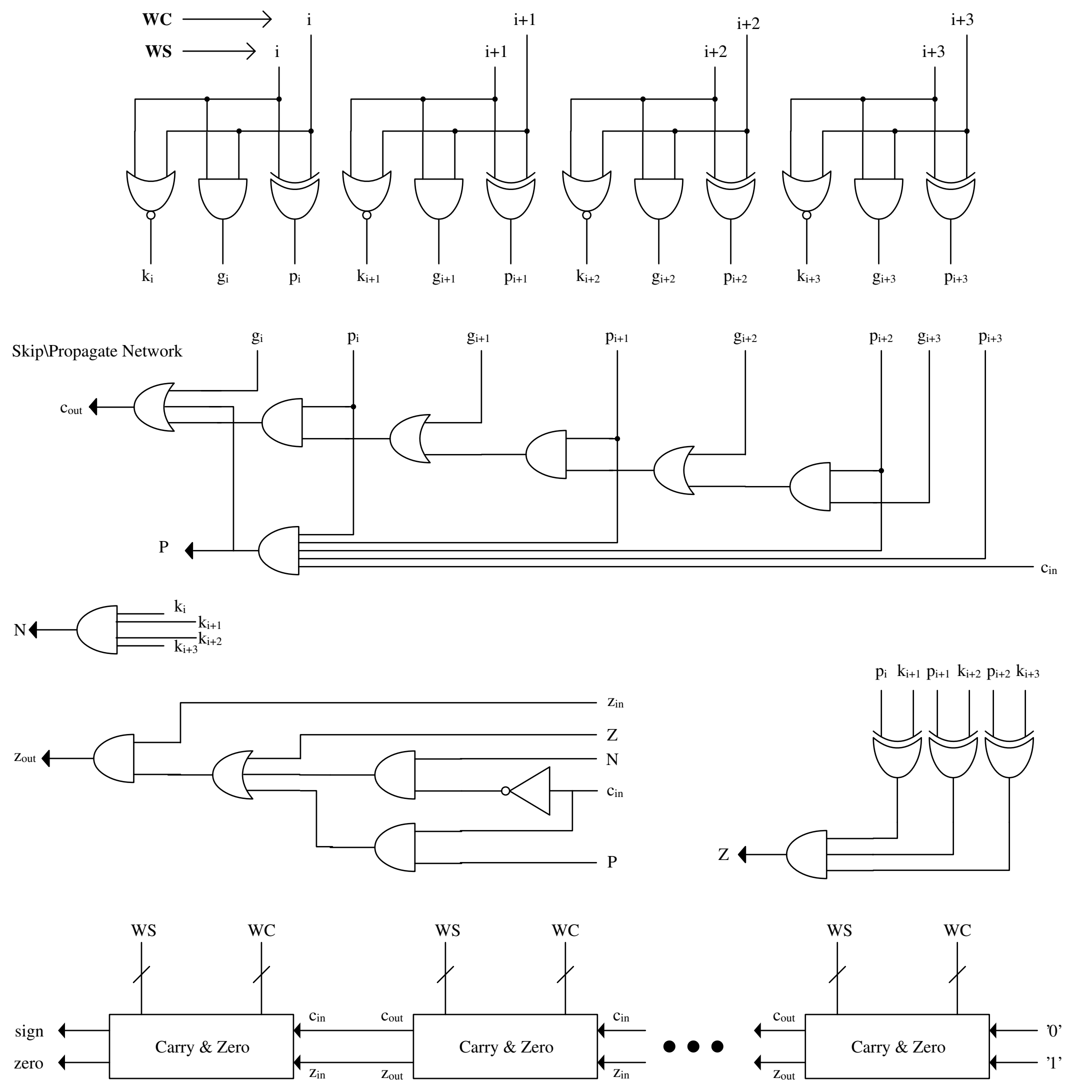

Figure 3.2: Sign and Zero Detection Network

divisor (53-51)	QTotal	q[k+1]
---	10-----	0001
---	01-----	1000
---	1-00---	0001
0--	0-1----	1000
---	0-11---	1000
00-	1-0----	0001
0-0	1-0----	0001
0--	1-0-0--	0001
-0-	1-0-0--	0001
0--	1-0--0-	0001
-0-	0-1-1--	1000
--0	0-1-1--	1000
--0	1-0-00-	0001
1--	1110---	0010
-1-	1110---	0010
---	11101--	0010
1--	0001---	0100
---	00010--	0100
-00	0-1--1-	1000
-0-	0-1--11	1000
1--	11011--	0010
--1	1110-1-	0010
0--	00001--	0100
000	0--11--	1000
00-	0--111-	1000
000	1--000-	0001
000	1--00-0	0001
00-	1--0000	0001
111	1101---	0010
11-	1101-1-	0010
0--	111100-	0010
111	0010---	0100
11-	00100--	0100
1--	001000-	0100
0-0	0--1111	1000
000	11110--	0010
-11	110111-	0010
1-1	00100-0	0100
011	000-1--	0100
-0-	000011-	0100
01-	000-10-	0100
0-1	000-10-	0100
01-	000-1-0	0100
0-1	111-001	0010
000	111-011	0010

Figure 3.3: PLA used for Quotient Digit Selection

makes no difference in terms of the output of the CSA. This concatenation is sufficient to save us from a full-precision addition at every cycle. In the cases of q_{j+1} being equal to 0, $\overline{1}$ or $\overline{2}$, there is no need for any concatenation since divisor multiple is already in correct form. The reason for that is, none of the three cases require two's complements, hence no addition with '1' is necessary.

Figure 3.4 shows the heart of the recurrence of the radix-4 SRT division unit. This logic is slightly modified when it is used for dual-mode division unit. At the beginning, MUX1 and MUX2 chooses either W[0] which is equal to the dividend or the partial remainder, W[j] ($j \neq 0$). Selection signals for these two multiplexors are same and they trigger according to the state counter, j, being equal to zero or not. Dividend is then passed through MUX1 (MUX2 lets a 58-bit number composed of all zeros through, making the $W[0] = WS[0] + WC[0] = dividend + zeros = dividend$ as it is supposed to be). In the next cycles of the algorithm, these multiplexer will let the partial residuals $WC[j]$ and $WS[j]$ pass through. Next, the residuals are multiplied by the radix (r=4). This is implemented by a two bit arithmetic left shifter. Meanwhile the divisor is multiplied by the quotient-bit received from the QDS function. Both the redundant residual and divisor multiple enters the CSA. Seven most significant bits of the residual's carry and save components ($SumN4$ and $CarryN4$) are added with the $7-bitCLA$. Then, along with the three most significant bits of the divisor's mantissa, output of the $7-bitCLA$ feeds the PLA (QDS function). As the cycles pass, the quotient is outputted from the PLA digit by digit (most significant digit first). For the conversion, these bits are fed to the on-the-fly conversion unit. The reason flip-flops (latches, also the black triangles in the figures) exist in this design is that the algorithm for division is iterative and the same hardware is used in a cycle several times before the result is generated. Without the flip-flops, outputs of the CSA ($SumN5$ and $CarryN5$) could not be inputs to $MUX1$ and $MUX2$. Doing so would cause the circuit to enter in an infinite loop.

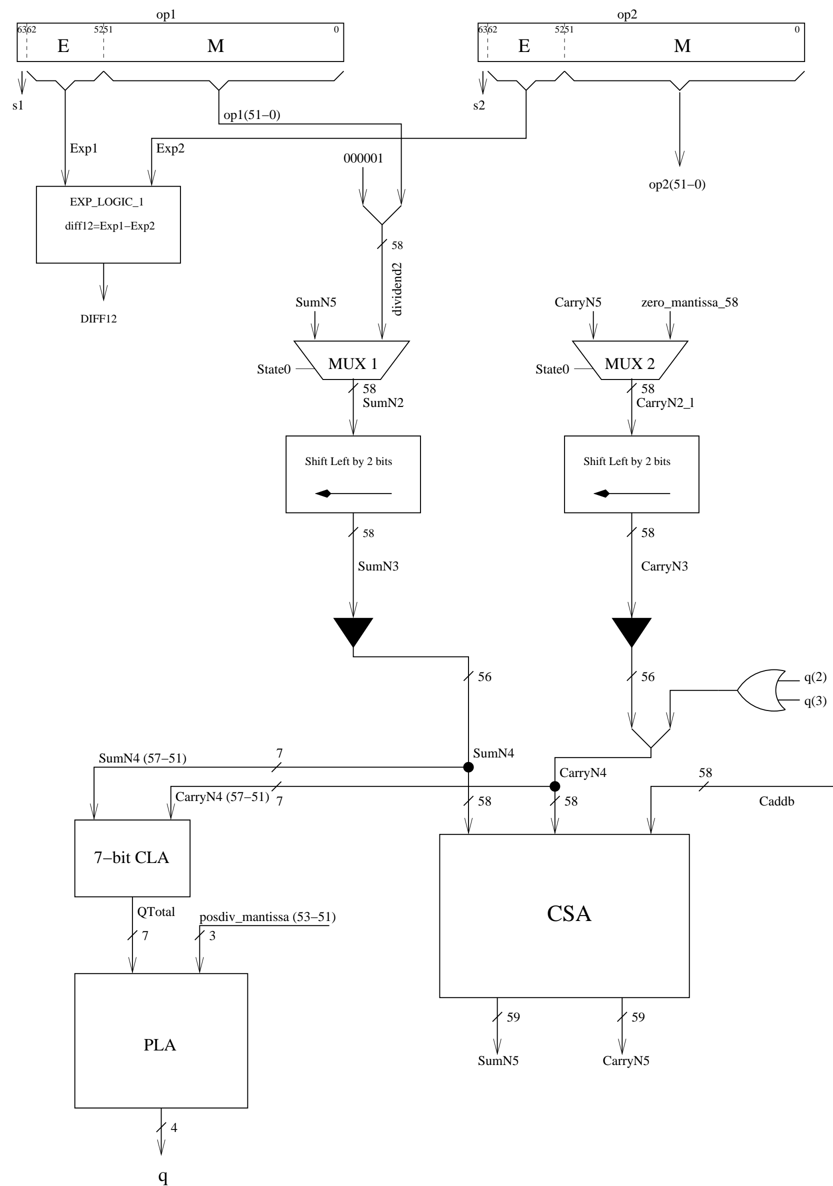

Figure 3.4: Main Recurrence Unit for SRT radix-4 Division

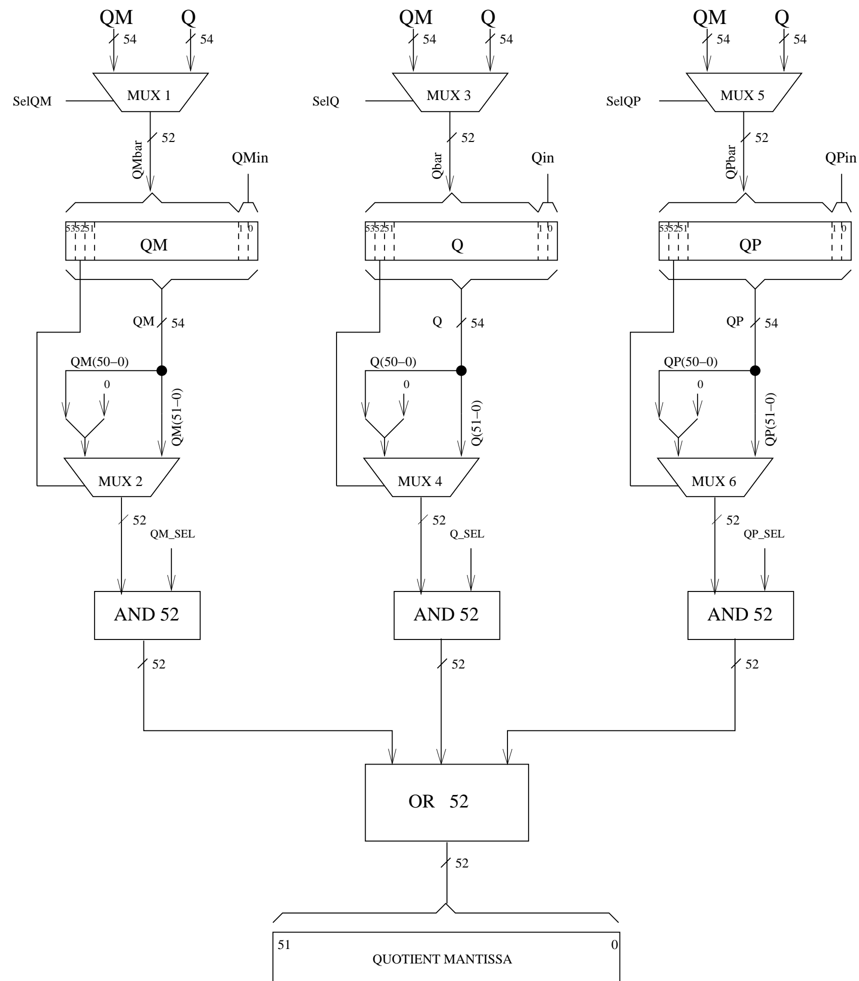

Figure 3.5: On-the-Fly Conversion and Rounding Unit

3.1.4 On-the-Fly Conversion and Rounding

What OTF conversion and rounding unit does is creating 2-bit signals, QMin, Qin and QPin from the quotient-digit selected and inserting those 2-bits in the trailing positions of QM, Q and QP registers accordingly. Heart of the on-the-fly conversion lies in these three registers QM, Q and QP which keeps three consecutive binary numbers, QP being the greatest and QM being the smallest one. At any time, QP is equal to $(Q + 1ulp)$ and $(QM + 2ulp)$. For updating, Q uses QM or itself depending on the quotient bit selected by the help of MUX3 in Figure 3.5. Same applies for the QM and QP registers. The multiplexors above the QM, Q and QP registers (MUX1, MUX3, MUX5) seen in Figure 3.5 are for this purpose. QP register keeps a number one *ulp* bigger than Q register for rounding. Since every redundant quotient bit produced generates two bits of the conventionally represented quotient, these multiplexors output only the 52 least significant bits. The details of the logic behind the updating rules can be found in [18]. The effect of this multiplexer-register loop is somewhat like the number in the register shifting to left by two bits at every cycle until filling the whole register with meaningful data. At the end of the last cycle, some bitwise operations are done to decide if Q, QM or QP is the final quotient. Nonetheless the mantissa needs to be normalized. If the 52nd bit of the register is a 1, the remaining 52 (51 downto 0) bits are the normalized result. If not, the result is 51 least significant bits of the register and a least significant bit is computed according to the 52nd bit, the conversion and rounding unit also outputs an one bit exponent_add_one signal which must be added to the exponent. Dual-mode on-the-fly conversion is a little bit more complex to implement considering no hardware is wasted in none of the modes and extra delay brought is only from a couple of multiplexors.

$$QP[k+1] = \begin{cases} Q[k], (q_{k+1} + 1) & if \quad q_{k+1} \geq -1 \\ QM[k], ((r+1) - |q_{k+1}|) & if \quad q_{k+1} < -1 \end{cases}$$

$$Q[k+1] = \begin{cases} Q[k], q_{k+1} & if \quad q_{k+1} \geq 0 \\ QM[k], (r - |q_{k+1}|) & if \quad q_{k+1} < 0 \end{cases}$$

$$QM[k+1] = \begin{cases} Q[k], (q_{k+1} - 1) & if \quad q_{k+1} > 0 \\ QM[k], ((r-1) - |q_{k+1}|) & if \quad q_{k+1} \leq 0 \end{cases}$$

q_{k+1}	QM[k+1]	Q[k+1]	QP[k+1]
2	Q[k], 01	Q[k], 10	Q[k], 11
1	Q[k], 00	Q[k], 01	Q[k], 10
0	QM[k], 11	Q[k], 00	Q[k], 01
-1	QM[k], 10	QM[k], 11	Q[k], 00
-2	QM[k], 01	QM[k], 10	QM[k], 11

Figure 3.6: On-the-Fly Conversion's Update Scenarios

These are the updating rules for the three consecutive versions of the quotient. $Q[k]$ is the exact converted version of the quotient-bits. $QM[k]$ and $QP[k]$ are 1-ulp less and more than $Q[k]$, respectively. Final result can be any of these three according to the rounding mode and final remainder. When a new quotient bit arrives, $Q[k+1]$ is created by concatenating either q_{k+1} at the end of $Q[k]$ or $r-|q_{k+1}|$ at the end of $QM[k]$. If the selected quotient-digit is negative, $QM[k]$ is being used. $QM[k+1]$ and $QP[k+1]$'s updating rules are in a similar manner. All the scenarios are stated in Figure 3.6. A *comma* (,) means concatenation in OTF related figures.

In Figure 3.7 a radix-4 on-the-fly conversion example is demonstrated. This example goes for 10 cycles, creating at the end a converted (to conventional format) 20-bit binary number. The states of all three registers can be seen at every cycle. Also a paper-pencil method to check the correctness of the OTF algorithm is shown. Implementing the paper-pencil method would require a costly full-precision (20-bit in this example) addition.

3.2 Dual-Mode SRT Radix-4 Quadruple Precision Division Unit

There are several design choices for the SRT division. Each of these brings an advantage on one part of the design while being costly on the other parts [16, 17]. Choice of the radix,

k	*q-bit*	**QM** (QMin)	**Q** (Qin)	**QP** (QPin)
0	***2***	01	10	11
1	***1***	1000	1001	1010
2	***-1***	100010	100011	100100
3	***-2***	10001001	10001010	10001011
4	***0***	1000100111	1000101000	1000101001
5	***0***	100010011111	100010100000	100010100001
6	***2***	10001010000001	10001010000010	10001010000011
7	***2***	1000101000001001	1000101000001010	1000101000001011
8	***-2***	100010100000100101	100010100000100110	100010100000100111
9	***2***	10001010000010011001	10001010000010011010	10001010000010011011

crosscheck:

radix-4 quotient: $21\bar{1}\bar{2}0022\bar{2}2$

positive parts: 2100002202 ⟶ 10010000000010100010

negative parts: 0012000020 ⟶ − 00000110000000001000

converted result same as Q[9]: 10001010000010011010

Figure 3.7: Step by step On-the-Fly Conversion Example

quotient digit set, residual format, and lots of other factors determine the overall speed and gate count of a division unit. In this work, the design choices are as follows: radix is four, quotient digit set is $\{2,1,0,\bar{1},\bar{2}\}$, residual is kept in carry-save format, and on-the-fly conversion and rounding is used [36].

Dual-mode quadruple precision divider supports both one quadruple precision and two parallel double precision division operations. The block diagram of the unit is shown in Figure 3.8. Basically, it consists of two conventional double precision division units, but the datapath of divider on the right is extended with additional two bits in order to support 118-bit datapath for quadruple precision. There are also additional multiplexors to ensure that both datapaths work together to support quadruple precision division. For example, Mux9 and Mux10 are used between the shifters and Mux12 is used between the CSAs.

When the divider is used for quadruple precision division, the *quad* signal is set and it is assumed that register pairs op1_left - op1_right and op2_left - op2_right hold input operands. A quadruple precision number consists of a 1-bit sign, a 15-bit biased exponent, and a 112-bit mantissa [7]. Quadruple division takes fifty nine cycles. When this unit performs two double precision division in parallel, it is assumed that the first pair of input operands is hold in registers op1_left and op2_left, and the second pair of input operands is hold in registers op1_right and op2_right. A double precision number consists of a 1-bit sign, an 11-bit biased exponent, and a 52-bit mantissa [19]. Two parallel double precision division takes twenty nine cycles. A finite state machine (FSM) is designed to accommodate for dual-mode cycle counting. Since the design is fully sequential, same hardware can be used for different precisions only with different number of stages required.

The partial remainder in carry-save format, the output of CSA, goes to the 7-bit CLA in every iteration. The PLA takes as inputs the output of CLA and the divisor in order to predict the next quotient bits. CLA_2 and PLA_2 are not used when the operation mode is quadruple.

3.2.1 Main Recurrence

CSA_Left and CSA_Right in Figure 3.8 are used as a whole 118-bit CSA in the quadruple mode. A 1-bit multiplexor (Mux12) is used to get the possible carry-out from the CSA_Right

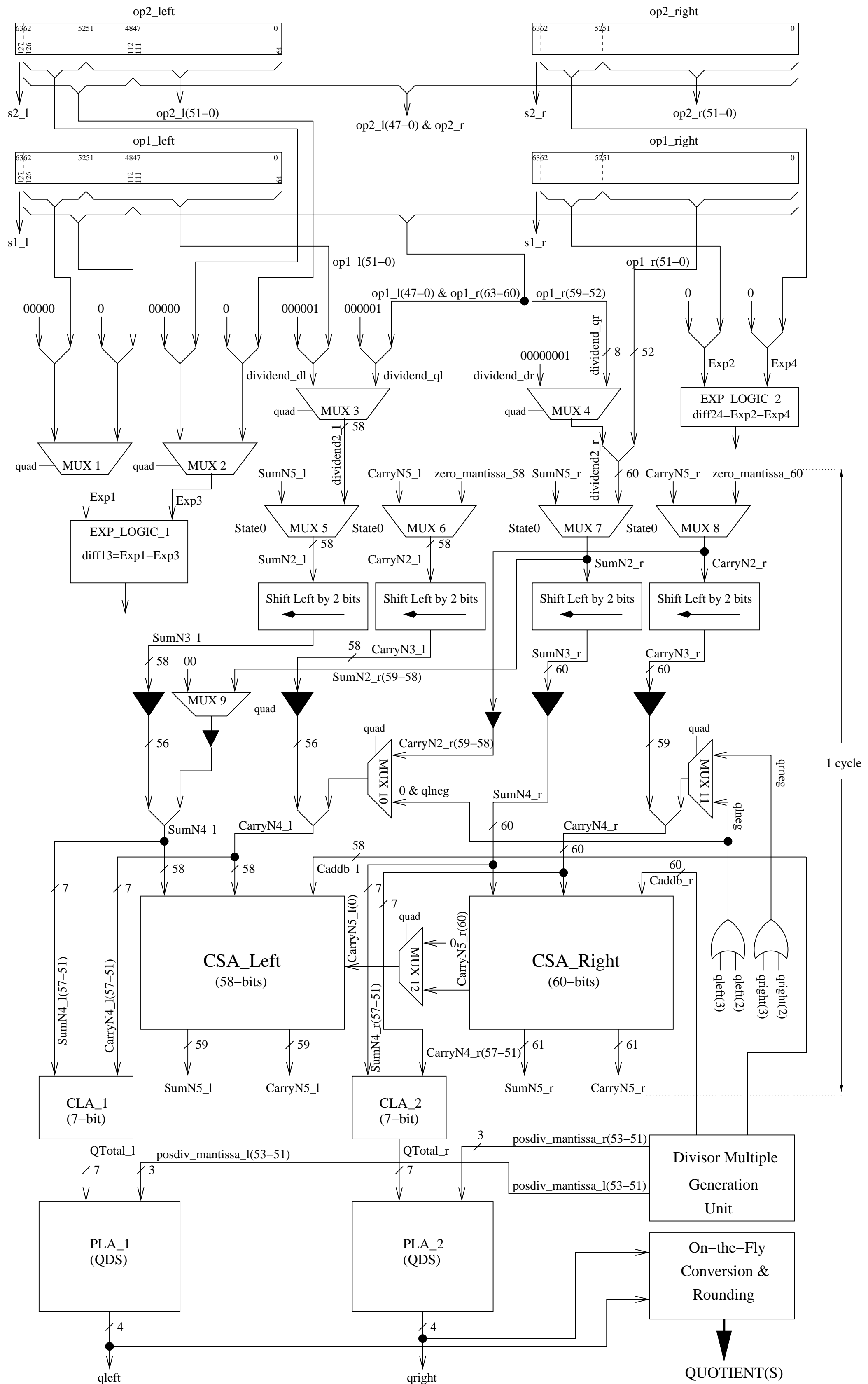

Figure 3.8: Dual-Mode Quadruple Precision Division Unit

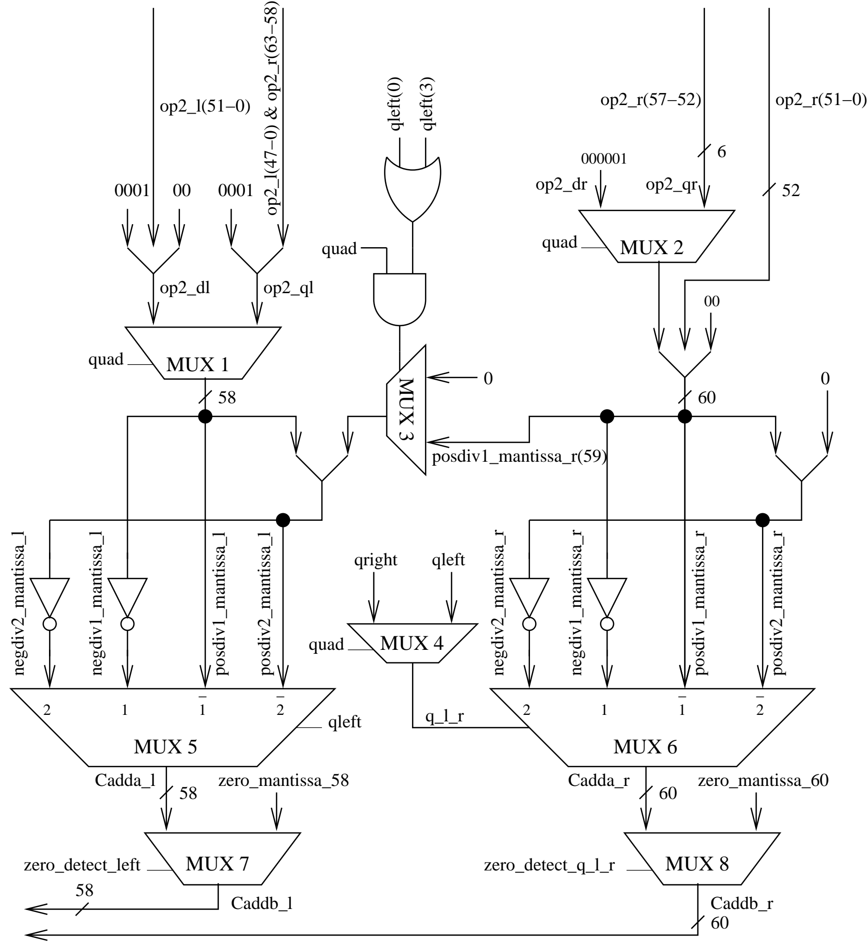

Figure 3.9: Divisor Multiple Generation Unit for Dual-Mode Quadruple Precision Division

and replaces it with the least significant bit of the carry-out of the CSA_Left. Similar method is used for multiplying residual W_j by four (shift left by 2 bits). Before multiplication (2-bit left shift), two most significant bits of the sum and carry of W_j *right* ($SumN2_r$ and $CarryN2_r$)are fed into Mux9 and Mux10. In quadruple precision mode, output of these multiplexors will replace the two least significant bits of the sum and carry of W_j *left*. Radix-4 SRT division requires seven bits of the residual and three bits of the divisor for the QDS. Therefore, only one QDS function is enough and the PLA_1 and CLA_1 pair are used for quadruple mode.

3.2.2 Generating Divisor Multiples

The unit shown in Figure 3.9 basically generates the divisor multiplies which will be added/subtracted from the previous residual to create the next residuals. Divisor multiple generation is not in the critical path. The divisor multiples (2d, d, -d, and -2d) can be formed by shifting. In order to generate -d and -2d, one's complement is obtained first. Since the least significant bit of the $4W_j$ is always zero, $1 - ulp$ is just concatenated to the least significant bit of the $4W_j$. This eliminates the carry propagation to obtain two's complement. Similar to the procedures in *Residual Creation*, additional multiplexors, Mux1 through Mux4, are used to generate divisor multiples for a quadruple and two double numbers.

3.2.3 On-the-Fly Rounding and Conversion

Basic idea of the on-the-fly conversion and rounding is to generate the conventional representation of the quotient from the radix-4 representation. In the dual-mode design, Q, QM, and QP registers are all divided into left and right blocks as shown in Figure 3.10. For instance, Q register is divided into 54-bit Qr (right) and 60-bit Ql (left) registers. Furthermore, additional multiplexors, Mux11, Mux14 and Mux23, are used to support both quadruple and two parallel double precision operations. Mux17 and Mux20 are used to normalize the right and the left halves of the result, respectively. Normalization is necessary when the magnitude of dividend's mantissa is smaller than divisor's mantissa. In such a case, resulting quotient's mantissa will be smaller than *one* and it needs to be shifted to the left by 1-bit. Mux20's selection depends on *quad* and the most significant bit of Qr and Ql registers. Mux20 is doing the exact same thing with Mux17, but in the quadruple precision mode, Mux20 needs to take selection bit of Mux17 into consideration. For instance, if 59th bit of *Ql* is selected through Mux14 and Mux17, Mux20 should select the bits 58 through 0 to keep the whole 112-bit quadruple quotient in rigid form. Finally, according to the selection logic, one of three register pairs which keeps the rounded quotient is fed into the final result register.

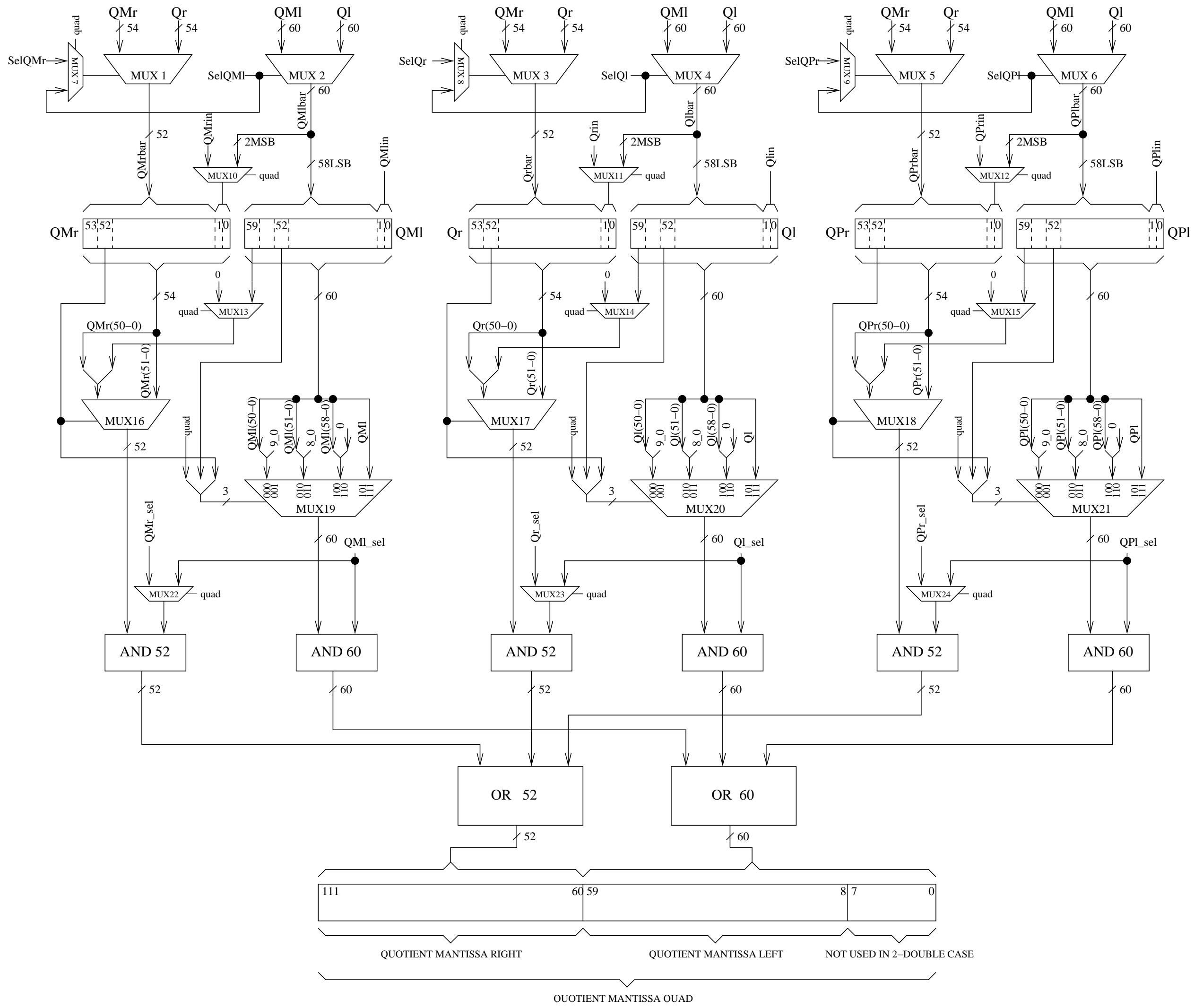

Figure 3.10: OTF Conversion and Rounding Unit for Dual-Mode Quadruple Precision Division

Design	**Number of Gates**	**Delay (ns)**
Conventional Double	178,257	3.61
Dual-Mode Double	212,854	3.92
Conventional Quadruple	428,783	4.21
Dual-Mode Quadruple	525,897	4.25

Table 3.1: Area and Delay Estimates.

3.3 Comparison of Synthesis Results

For comparison purposes, a conventional double precision, a conventional quadruple precision, and a dual-mode double precision division units are also implemented in VHDL and synthesized. The dual-mode double precision divider is similar to the dual-mode quadruple precision divider except that it supports one double precision and two parallel single precision division operations. The area and worst case delay estimates for all four dividers are shown in Table 3.1. The synthesis results are obtained with Mentor Graphics' LeonardoSpectrum synthesis tool and the TSMC 0.25 micron CMOS standard cell library. The TSMC 0.25 micron library has five metal layers and one polysilicon layer. The area is given in terms of equivalent gates and results are normalized, such that an equivalent gate corresponds to the area of a single minimum-size inverter. The dual-mode quadruple precision division unit requires 22% more area and only 1% more delay than the conventional quadruple division unit. Compared to the conventional double precision divider, the dual-mode double precision divider requires roughly 20% more gates and the worst case delay increases by 6%.

Chapter 4

DUAL-MODE SRT RADIX-4 SQUARE-ROOT UNIT

As in division, different specific versions of the SRT algorithm are possible, depending on the radix, the redundancy factor, the type of representation of the residual and the result-digit selection function. Moreover, the implementation can be sequential, combinational, or a combination of both. Even pipelining can be used if preferred. For simplicity, square-root implementation uses the exact same design choices with the division shown in this thesis. The PLA (containing the result-digit selection function) used by the two operations are same as well. However, as will be seen in next section, additional logic gates are necessary for square root algorithm to share the same PLA used for division. Since square-root operation involves just one operand, the equivalent unit to *Divisor Multiple Generation* does not exist. Instead, *F Generation Unit* exists and it is the third input to the CSA in the main unit. OTF (On-the-Fly Conversion and Rounding) algorithm is embedded into the F generation mechanism since they have too much in common. Other than these natural differences of these two operations, the implementations are very similar. The similarities of the block diagrams (Figure 3.8 and Figure 3.4) proves this visually [36].

4.1 Conventional SRT Radix-4 Square Root Unit

4.1.1 Main Recurrence and Carry-Save Adder

$$W[j+1] = r \times W[j] + F[j] \qquad \longrightarrow \text{Square-Root}$$
$$W[j+1] = r \times W[j] - d \times q_{j+1} \qquad \longrightarrow \text{Division}$$

Main recurrence of the square-root algorithm is very similar to the division as seen in the above formulas. The only difference is the F component being used instead of the divisor multiple component. First, the operand's mantissa is put next to four '1's. The reason for this is that simple but first remember the formulas driven in Section 2.5:

$$W[0] = x - 1$$

and,

$$W[j] = WC[j] + WS[j]$$

therefore,

$$WS[0] = x - 1$$
$$WC[0] = 0$$

Subtracting '1' from a floating-point number which is smaller than one leaves the fractional part same but negates the part before the fractional point. This is the reason why four leading '1's are prefixed to $op1(51-0)$ in Figure 4.1. At the next stage, $SumN2$ and $CarryN2$ are multiplied by 4 (radix) which is a left shift by 2-bit to create $SumN3$ and $CarryN3$. These two signals pass through 56-bit flip-flops (latches). These flip-flops hold an important job. Because both the division and square-root algorithms are iterative, all the unit blocks inside the implementation will have their updated outputs as their inputs many times. If not passed through a latch, an infinite loop might occur and timing difficulties arise. When such a mistake is done and tried to be simulated, an error occurs. The outputs of these flip-flops $SumN4$ and $CarryN4$ are basically the previous cycles' $SumN3$ and $CarryN3$. So latches bring an extra cycle to the implementation and the FSM is designed to accommodate this extra delay. Most significant 7-bit of $SumN4$ and $CarryN4$ are added with CLA and the result is fed into result-digit selection PLA. Full precision $SumN4$ and $CarryN4$ are input of the CSA. PLA's result s_{j+1} is fed into F generation unit and F is fed into the CSA as well. At this stage, the new residuals are found and these steps are carried out 29 times for calculating square-root of double precision operand. At every cycle, one digit of the result is being computed. F generation unit takes care of converting these redundant radix-4 result-digits to conventional form. Each radix-4 result-digit is converted to two binary-bits. Details of how F is generated is shown in F Generation Unit's section.

4.1.2 Result-Digit Selection Function

The same PLA shown in Figure 3.3 and Figure 2.5 is used for square-root too. Notice however that in division, PLA's 3-bit input was the most significant 3-bit of the divisor

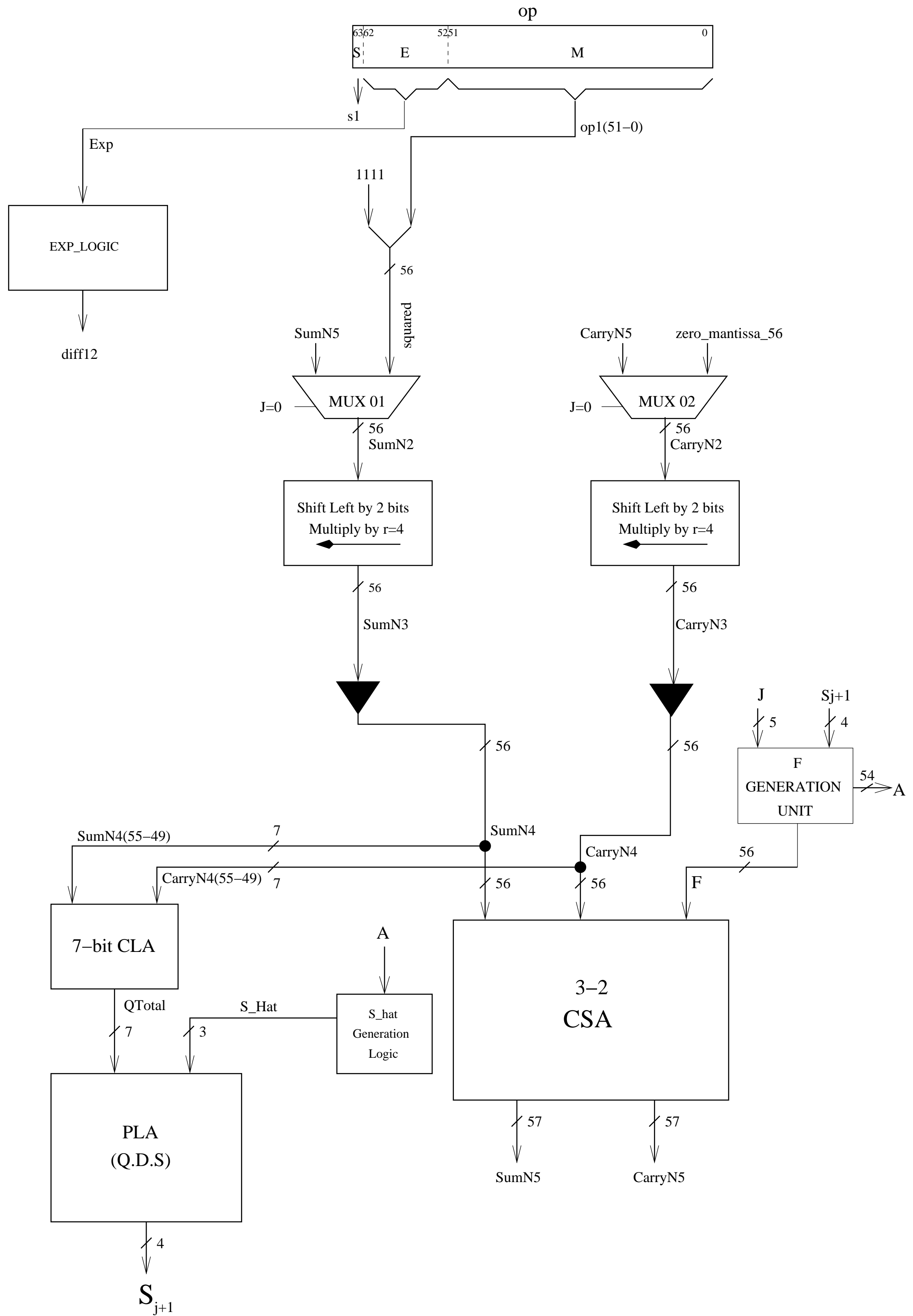

Figure 4.1: Square-Root Main Unit

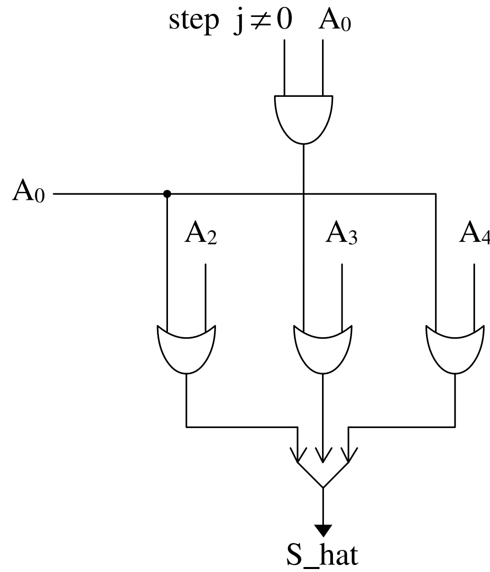

Figure 4.2: Square-Root S_hat Generation Logic

which is a constant value throughout the algorithm. In square-root implementation, there is a new block called S_hat generation logic. The input to the result-digit selection function is a three-bit value which is determined according to S[j] and j. Figure 4.2 shows inside of the S_hat generation logic. Signals A_0 through A_4 are the most significant bits of the current result. These signals are necessary for the division's PLA to work perfectly for square-root as well. S_hat signal is generated using these signals and then fed into the PLA. The details on deriving this logic is explained in [2]. Basically the only difference in result-digit selection logic which effects the use of the same PLA is at the first three iterations of the square-root algorithm. Using S_hat logic makes it possible to use the same PLA used for division in order to get correct values from the PLA in the first three cycles of the square-root algorithm.

4.1.3 F-Generation Unit

$$A[j] = S[j]$$

$$B[j] = S[j] - r^{-j}$$

$$F = -(2 \times S[j] \times s_{j+1} + s_{j+1}^2 \times r^{-(j+1)})$$

For $s_{j+1} > 0$:

$$F = -(2 \times A[j] + s_{j+1} \times r^{-(j+1)}) \times s_{j+1}$$

For $s_{j+1} < 0$:

$$F = (2B[j] + (2 \times r - |s_{j+1}|) \times r^{-(j+1)}) \times |s_{j+1}|$$

Note that these last two expressions are obtained by concatenation and multiplication by a radix-r digit. Since minimally-redundant radix-4 result-digit set is being used, the multiplications are just arithmetic shifts. As seen in Table 4.1, the values of F[j] are extracted from these two equalities. Plugging, r equals four and different result-digits to the formulas above will yield the values stated in the table. Bit-String column in Table 4.1 is the implementation view of F[j]. What the F generation unit in Figure 4.3 does is updating A[j] and B[j] at every cycle accordingly using $MUXXX01$ through $MUXXX03$ and $MUX01$ through $MUX03$. If result-digit is '1' negating A[j] and concatenating '111' at the end. If result-digit is '2', shifting negated A[j] left once and then concatenating '1100'. If result digit is negative, for instance '-1', B[j] only needs to be concatenated by '111' without being negated. Finally, if the result-digit selected is '-2', B[j] needs to be shifted left by 1-bit and then concatenated by '1100'. Concatenations are done with the help of $AND54$, $XOR54$, $REG58$ and $REG59$ units. The F Generation unit in Figure 4.3 holds about twice the area of on-the-fly conversion and rounding unit of division because in square-root, F generation unit is responsible of both generating F and also doing on-the-fly conversion.

4.1.4 On-the-Fly Conversion and Rounding

As stated before, on-the-fly conversion and rounding is embedded into the F generation unit. For generating F, the values of S[j] which is same as A[j] is necessary and it is a good idea to combine these two units into one. B[j], A[j] and C[j] are three consecutive numbers going from smallest to biggest. Figure 4.3 shows the details of on-the-fly (OTF) conversion and F-generation. As an example, A[j]'s creation can be explained. According to the polarity of s_{j+1}, $MUX01$ lets either A[j] or B[j] to the output signal to be named $Abar$.

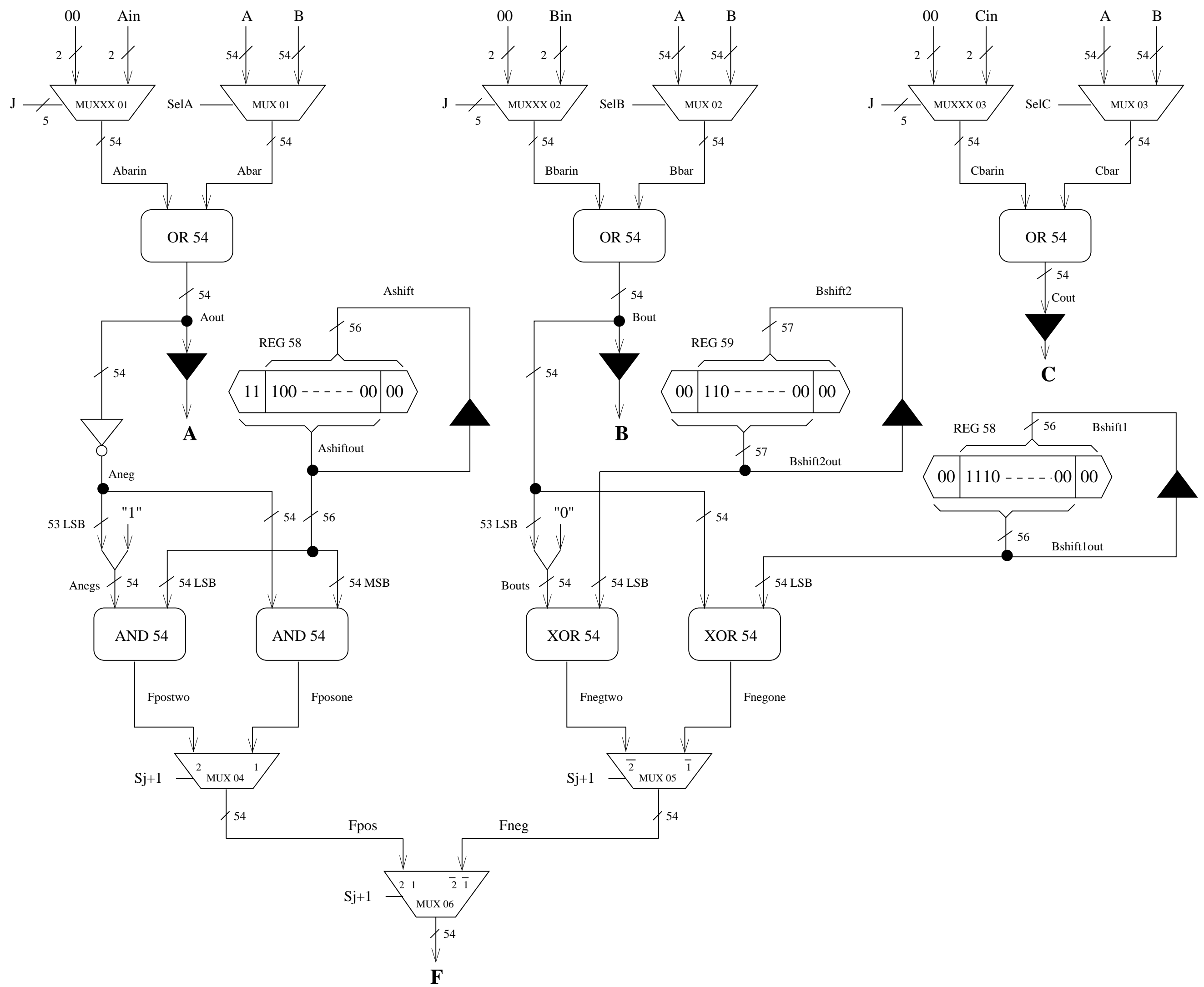

Figure 4.3: A Combined F generation and OTF Unit for Square-Root

	F[j]		
s_{j+1}	Value (in terms of S[j])	Value (in terms of A[j] and B[j])	Bit-String
-2	$4S[j] - 4 \times 4^{-(j+1)}$	$4B[j] + 12 \times 4^{-(j+1)}$	$b \cdots\cdots b1100$
-1	$2S[j] - 4^{-(j+1)}$	$2B[j] + 7 \times 4^{-(j+1)}$	$b \cdots\cdots bb111$
0	0	0	$0 \cdots\cdots 00000$
1	$-2S[j] - 4^{-(j+1)}$	$-2A[j] - 4^{-(j+1)}$	$\overline{a} \cdots\cdots \overline{aa}111$
2	$-4S[j] - 4 \times 4^{-(j+1)}$	$-4A[j] - 4 \times 4^{-(j+1)}$	$\overline{a} \cdots\cdots \overline{a}1100$

Table 4.1: F Generation Scenarios

$MUXXX01$ is a special multiplexor outputting a fifty-four bit number and inputting two bit numbers. The selection signal in this case is five bits. Basically this multiplexor which is different than the conventional ones puts Ain in the jth position of the output, $Abarin$. The remaining bits are all '00's which was the first input of $MUXXX01$. $Abarin$ and $Abar$ are ORed so that Ain is concatenated at the end of A[j] or B[j] in order to output A[j+1]. The reason why there is a flip-flop is because A[j+1] is fed into $MUX01$ back again and a "zero delay oscillation" is being avoided by using the system clock. The updating rules defined below are implemented with these multiplexors, OR gates and flip-flops. A[j] and B[j] are used for generating F[j] as explained in Section 4.1.3.

$$C[k+1] = \begin{cases} A[k], (s_{k+1} + 1) & if \quad s_{k+1} \geq -1 \\ B[k], ((r+1) - |s_{k+1}|) & otherwise \end{cases}$$

$$A[k+1] = \begin{cases} A[k], s_{k+1} & if \quad s_{k+1} \geq 0 \\ B[k], (r - |s_{k+1}|) & otherwise \end{cases}$$

$$B[k+1] = \begin{cases} A[k], (s_{k+1} - 1) & if \quad s_{k+1} \geq 1 \\ B[k], ((r-1) - |s_{k+1}|) & otherwise \end{cases}$$

Conversion in square-root is same with division, but the implementation is different in square-root because A[j] and B[j] are further used to generate another value, F[j]. Rounding is done by just looking at the last result-digit, final remainder and deciding which one of

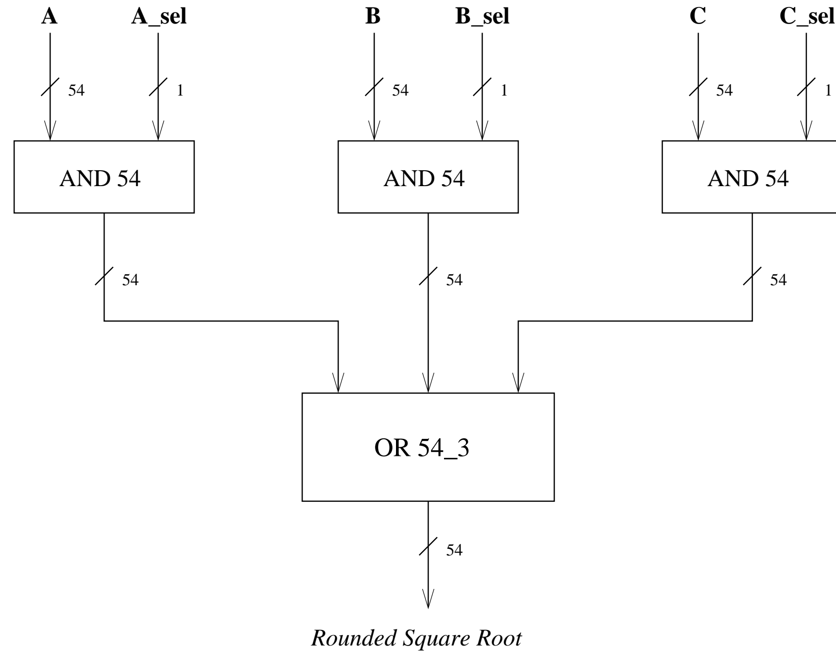

Figure 4.4: Square-Root Result from the OTF

the three registers, A[j], B[j] and C[j] is the final square-root result. Implementation of this is shown in Figure 4.4

4.2 Dual-Mode Quadruple Precision SRT Radix-4 Square-Root Unit

Dual-mode quadruple precision square-root unit supports both one quadruple precision and two parallel double precision square-root operations. The block diagram of the unit is shown in Figure 4.5. Basically, it consists of two conventional double precision square-root units as shown in Figure 3.8, but the datapath of the unit on the right is extended with additional four bits and the unit on the left with additional two bits in order to support 118-bit datapath for quadruple precision. There are also additional multiplexors to ensure that both the left and right datapaths work together as a whole unit to support quadruple precision division. For example, Mux9 and Mux10 are used between the shifters and Mux12 is used between the CSAs.

When the square-root unit is used for quadruple precision operation, the *quad* signal is

set and it is assumed that register pair op_left - op_right hold input operands. A quadruple precision number consists of a 1-bit sign, a 15-bit biased exponent, and a 112-bit mantissa [7]. Quadruple square-root takes fifty nine cycles. When this unit performs two double precision square-root operations in parallel, it is assumed that the first input operand is hold in registers op_left, and the second input operand is hold in registers op_right. A double precision number consists of a 1-bit sign, an 11-bit biased exponent, and a 52-bit mantissa [19]. Two parallel double precision square-root operations takes twenty nine cycles. A finite state machine (FSM) is designed to accommodate for dual-mode cycle counting. Since the design is fully sequential, number of states does not affect the hardware used per cycle.

The partial remainder in carry-save format (the output of CSA), goes to the 7-bit CLA in every iteration. The PLA takes as inputs the output of CLA and the S_hat in order to predict the next quotient bits. CLA_2 and PLA_2 are not used when the operation mode is quadruple.

4.2.1 Main Recurrence

Workings of both the quadruple and two parallel double-precision square-root modes will be explained in this section. Block diagram of the main recurrence unit which does the result-digit selection and partial remainder creation is shown in Figure 4.5. Finite-State-Machine is designed to handle both the modes according to the quad signal. If quad signal is set (which means square-root operation is done in quadruple precision mode), the algorithm will run for fifty-nine cycles. Otherwise, it will run for twenty-nine cycles to compute square-roots of two double precision numbers. How these numbers are found are as follows. For the double precision mode, the mantissa is 52-bit length and the *squared_dl* signal takes six leading ones as prefix in Figure 4.5. This makes $\frac{58}{lg_2(4)} = 29$ cycles as total. Right half is two-bits wider, 60-bits, and it does not make a difference in the number of cycles it has to run to get the correct result. The reason is that, both the CLAs inputs are SumN4_l(55-49), CarryN4_l(55-49) and SumN4_r(55-49), CarryN4_r(55-49). The bit positions of these pairs are the same and the most significant two-bit on the datapath of right side are not taken into account in the double precision mode. MUX09 and MUX10 are just used to pass '00's through because in double precision mode, after a left-shift of two-bit, the right-most two-

bit must be zero. These multiplexors are used to support quadruple-precision operation. The square-root result for the operand in *op_left* is computed using CLA left and PLA1. The right square-root operation's result bits are calculated with the right CLA and PLA2. The double precision mode can be seen as two double square-root datapaths put next to each other. The real challenge comes in when converting this datapath to work for also quadruple precision square-root operation as well.

Thinking of the two parallel datapaths as one rigid datapath, shift-left-by-2 units will damage the rigidness. Shift-left-by-2 units will pad two zeros at the least-significant-bit places of SumN3_l and CarryN3_l. However, in the quadruple mode, at that bit position there has to be the most-significant two-bit of the SumN3_l and CarryN3_l. MUX09 and MUX10 do this job in the quadruple precision mode. As explained before, the triangular shaped, black painted units are flip-flops and they play an important role on the functionality of the algorithm. Since the SRT square-root algorithm is iterative, the whole design is in a big loop. The output of CSA Left, SumN5_l is also the input of MUX05. The output of MUX05 goes to several gates, at the end, arriving to the CSA Left. If the latches were not in the way, it would not be possible to arrange timing on the circuit. At every cycle, PLA1 generates s_{j+1} for the quadruple precision square-root operation. Second CLA on the right and PLA2 are not used in this mode. In order to use the CSA Left and CSA Right as a rigid CSA, the carry-out of the right CSA needs to be fed into the left CSA as a carry-in. MUX12 does this job for the quadruple precision mode. F-Generation unit supplies both the CSA's with appropriate F's. The details of this is in the next section.

4.2.2 F-Generation and On-the-Fly Conversion and Rounding

Generating the number F[j] is very easy once A[j] and B[j] are ready. Creation of A[j] and B[j] are done by the on-the-fly conversion unit. How this unit converts to dual-mode was shown in the previous chapter in Figure 3.10. Generating F[j] using these two values requires negation, shift and finally concatenation. The same F-generation principles in Table 4.1 applies for the dual-mode design as well. Implementation of these operations requires three registers which are updating themselves at every iteration. In dual-mode, these units get squared and a total of nine registers are required. How the original design is

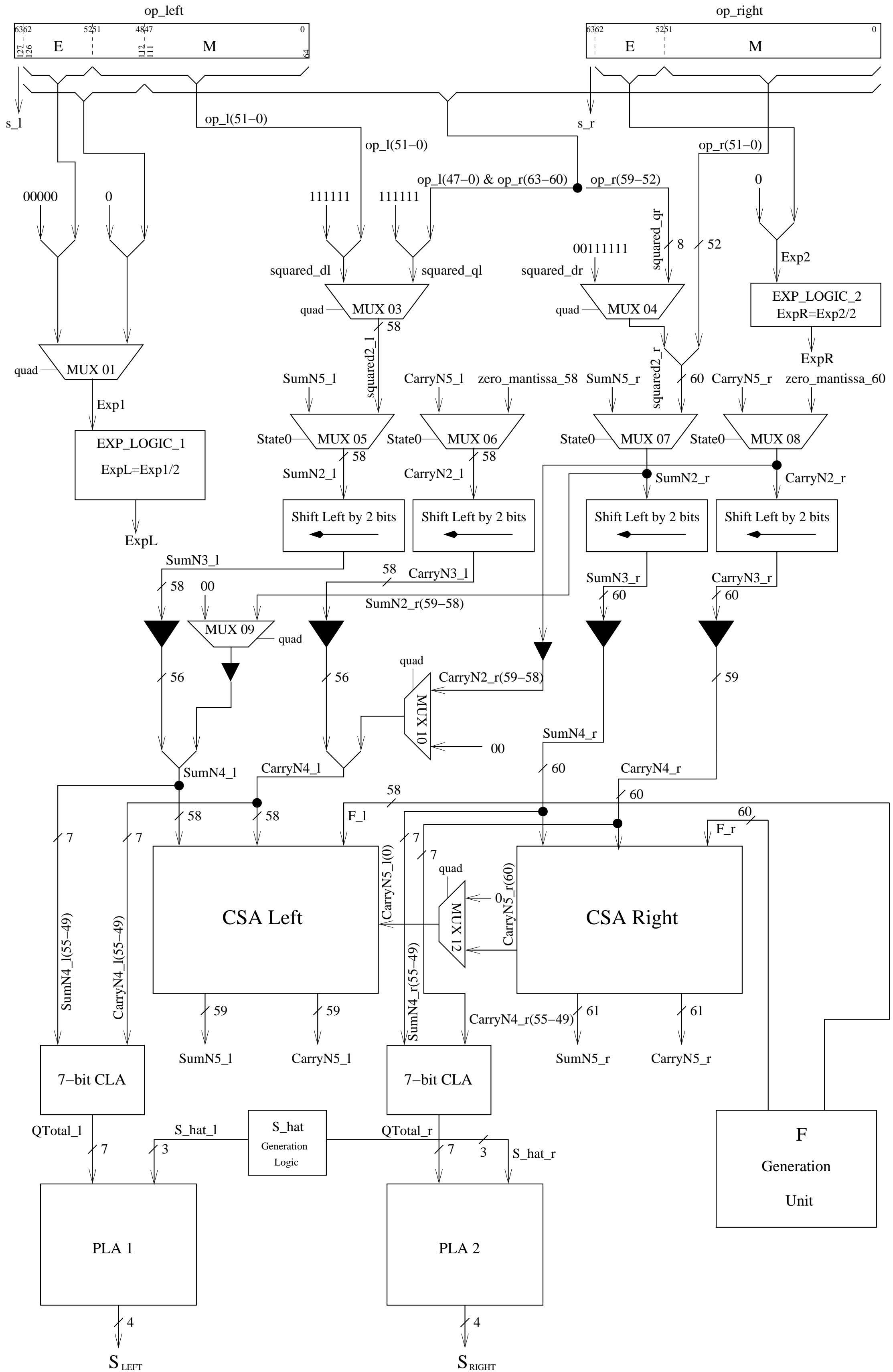

Figure 4.5: Dual-Mode Square-Root Residual and Result-Digit Generation Unit

Design	**Number of Gates**	**Delay (ns)**
Conventional Quadruple	497,625	4.61
Dual-Mode Quadruple	603,293	4.92

Table 4.2: Area and Delay Estimates.

turned into dual-mode for these registers are shown in Figure 4.6. Basically, if the mode is *quad*, The left and the right registers work as a whole, And when the mode is not *quad*, left and the middle registers go to the left and right double square-root operations individually. Since F-Generation module is more complex than the divisor multiple generation unit in division implementation, the area and delay results are a little bit worse than divisions. But the ratios of dual-mode to conventional are very similar to the ratios obtained for division. Since square-root is a more rare operation than division, this small drop in performance should not be a big problem.

4.3 Comparison of Synthesis Results

For comparison purposes, a conventional quadruple precision and a dual-mode quadruple precision square-root units are implemented in VHDL and synthesized. The area and worst case delay estimates for two square-root unit implementations are shown in Table 4.2. The synthesis results are obtained with Mentor Graphics' LeonardoSpectrum synthesis tool and the TSMC 0.25 micron CMOS standard cell library. The TSMC 0.25 micron library has five metal layers and one polysilicon layer. The area is given in terms of equivalent gates and results are normalized, such that an equivalent gate corresponds to the area of a single minimum-size inverter. Compared to the conventional quadruple square root unit, the dual-mode quadruple precision square root unit requires 22% more area and and have only 2% more delay.

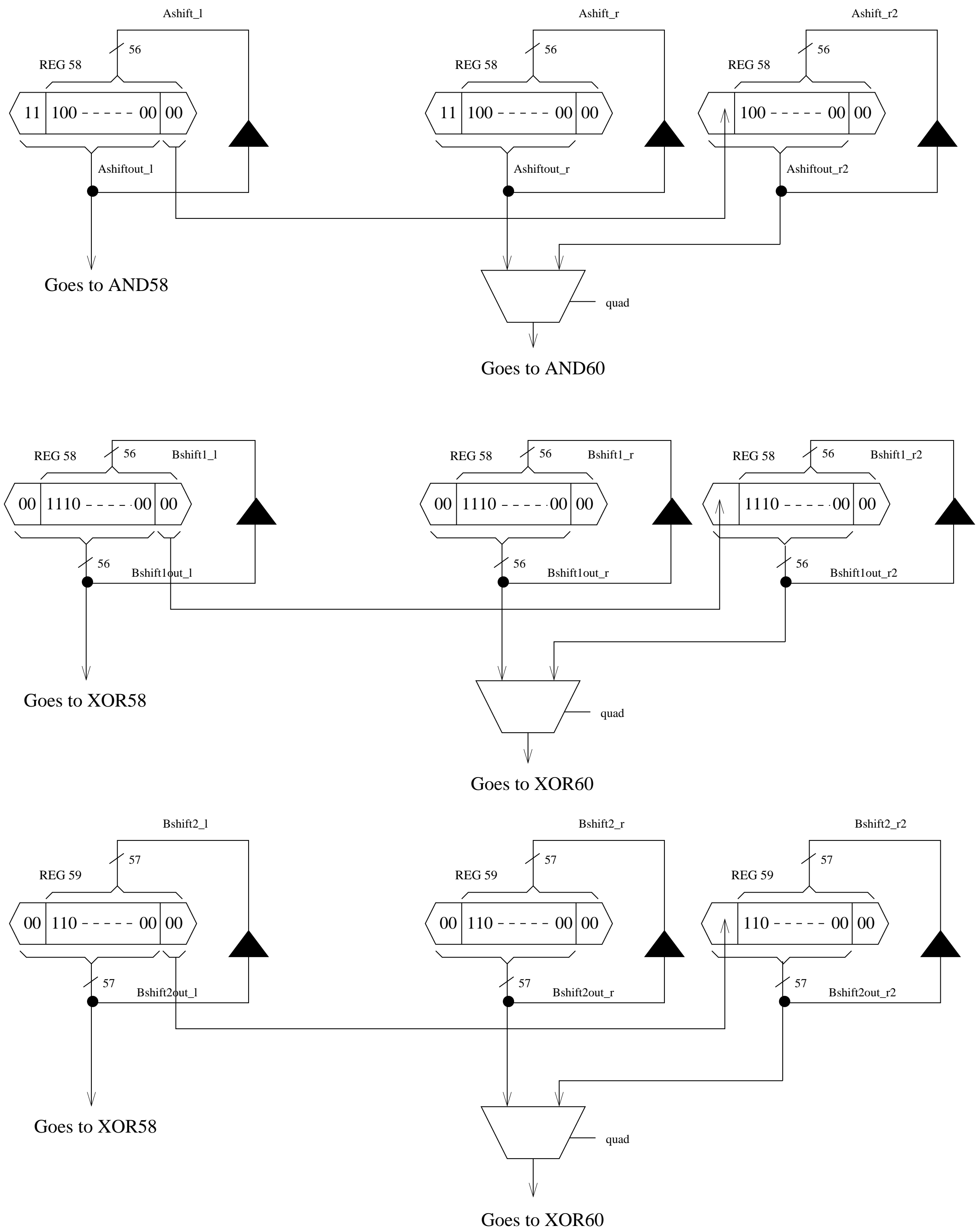

Figure 4.6: Dual-Mode Square-Root F-Generation Unit Register Extension Principle

Chapter 5

CONCLUSION

In this thesis, it is shown how a conventional double precision radix-4 SRT divider can be modified to design a dual-mode quadruple precision floating-point division unit. Also how a conventional double precision radix-4 SRT square-root unit can be modified to design a dual-mode quadruple precision floating-point square-root unit is explained i great detail in this thesis. These units support both one quadruple and two parallel double precision operations. The dual-mode quadruple precision floating-point division unit can perform either one quadruple precision division in fifty nine cycles or two parallel double precision division in twenty nine cycles. Same exact number of cycles apply to the square-root unit as well. The dual-mode quadruple precision division unit requires 22% more area and only 1% more delay than the conventional quadruple division unit. The same technique used to design the dual-mode quadruple precision divider is also applied to design a dual-mode double precision divider that supports both one double precision and two parallel single precision operations. Two single precision division operations take fifteen cycles and one double precision division take twenty nine cycles. The dual-mode quadruple precision square-root unit requires 22% more area and 2% more delay than the conventional quadruple square-root unit. The correctness of all designs are tested for four different rounding modes specified by IEEE standard for floating-point arithmetic. The greatest advantage of dual-mode units is to increase ILP for applications that requires lower precision operations.

Chapter 6

APPENDIX A

With the C division simulator coded, when the two numbers 0.456700 and 0.234500 are divided to each other, we get this step by step SRT radix-4 division algorithm's results. At the end of the simulation, computed result is compared with the CPU's own result. The name of the signals are exactly same as used in Figure 3.4.

DIVIDEND = 011101001110101001001010100011000001010101001100100110011 = 0.456700

DIVISOR = 001111000000010000011000100100110111010010111100011010 = 0.234500

X = 00000101110100111010100100101010001100000101010100110010011

D = 000100111100000001000001100010010011011101001011110001101000

r0 00000101110100111010100100101010001100000101010100110010011

Qhat0 0001011 and Shat 001 set q1 = 1

A1 101

B1 100

caddb0 11101100001111110111110011101101100100010110100001110010111

==

1

SumN3 00010111010011101010010010101000110000010101010011001001100

CarryN3 000000000000000000000000000000000000000000000000000000001

caddb 11101100001111110111110011101101100100010110100001110010111

+

SumN5 111110110111000111011000010001010101000000011110010111011010

CarryN5 00001000000011100010010010101000100000001010000000010000001010

r1 00000011100011100010000110010110010100101011110100111100100

Qhat1 0000110 and Shat 001 set q2 = 1

A2 10101

B2 10100

caddb1 1110110000111111011111001110110110010001011010000111001011 1

==

2

SumN3 1110110111000111011000010001010101000000111100101110110 1000

CarryN3 0010000000111000100100101010001000000010100000000100000010 1001

caddb 1110110000111111011111001110110110010001011010000111001011 1

+

SumN5 0010000110001001001110001011110011011011100110001001101011 0

CarryN5 1101100011101110110010101000101000000000011000100110010100 10

r2 1111101001111000000000011010001101101110001011101011001010 00

Qhat2 1110100 and Shat 001 set q3 = -1

A3 1010011

B3 1010010

caddb2 0001001111000000010000011000100100110111010010111100011010 00

==

3

SumN3 1000011000100100111000101111001101101110011000100110101100 0

CarryN3 0110001110111011001010100010100000000001100010011001010010 00

caddb 0001001111000000010000011000100100110111010010111100011010 00

+

SumN5 1111011001011111010010111100100100000011111001101100111100 0

CarryN5 0000011101000001010001000110010011011100001001100101001000 0

r3 1111110110100000100100000010110111100000000011010010000100 0

Qhat3 1111010 and Shat 001 set q4 = 0

A4 101001100

B4 101001011

caddb3 000000000

===

4

SumN3 110110010111110100101111001001000000111110011011001111100000

CarryN3 000111010000010100010001100100110111000010011001010010000000

caddb 000000000

+

SumN5 110001000111100000111110101101110111111100000010011101000000

CarryN5 001100100000101000000001000000000000000010011001000010000000

r4 111101101000001001000000101101111000000000110100100001000000

Qhat4 1101100 and Shat 001 set q5 = -2

A5 10100101110

B5 10100101101

caddb4 001001111000000100000110001001001101110100101111000110100000

===

5

SumN3 000100011110000011111010110111011111110000001001110100000000

CarryN3 110010000010100000000100000000000000001001100100001000000000

caddb 001001111000000100000110001001001101110100101111000110100000

+

SumN5 111111100100100111110100111110010010010111101110100010100000

CarryN5 000000110100000000010100000001001101110000000100101010000000

r5 000000011000101000001001000000101101111000000000100101010000

Qhat5 0000010 and Shat 001 set q6 = 0

A6 1010010111000

B6 1010010110111

caddb5 0000000000000

===

6

SumN3 1111100100100111110100111110010010010111101110100010100000

CarryN3 000011010000000000101000000100110111000000100101010000000000

caddb 0000000000000

+

SumN5 1111010000100111100000111100001001110111111100001010100000

CarryN5 000100100000000001010000001001001000000000001010000000000000

r6 00000110001010000010010000001011011110000000010010101000000

Qhat6 0001100 and Shat 001 set q7 = 1

A7 101001011100001

B7 101001011100000

caddb6 11101100001111110111110011101101100100010110100001110010111

==

7

SumN3 1101000010011110000011110000100111011111110000101010000000000

CarryN3 010010000000000101000000100100100000000000101000000000000001

caddb 11101100001111110111110011101101100100010110100001110010111

+

SumN5 011101001010001111110010110000000100111011111010110100101 10

CarryN5 100100000001111000000110100101101100100010100000000010000000010

r7 00000100111000000000011010001101101110001011110110001001 1000

Qhat7 0001001 and Shat 001 set q8 = 1

A8 10100101110000101

B8 10100101110000100

caddb7 11101100001111110111110011101101100100010110100001110010111

==

8

SumN3 110100101000111111001011000000010011101111101011010010 11000

CarryN3 010000000111100000011010010110110010001010000000001000000001001

caddb 11101100001111110111110011101101100100010110100001110010111

+

SumN5 011111100100000011011110100000000010000010000010001110001100

CarryN5 10000001011111101101001011011011001101101101001010000110010

r8 11111111101111111011000101011011010101110101010010111111000

Qhat8 1111110 and Shat 001 set q9 = 0

A9 1010010111000010100

B9 1010010111000010011

caddb8 000000000000000000

==

9

SumN3 11111001000000110111101000000000100000100000100011100011000

CarryN3 00000101111110110100101101101100110110110100101000011001000

caddb 000000000000000000

+

SumN5 11111100111110000011000101101100010110010100001011111010000

CarryN5 000000100000011010010100000000001000001000001000000000010000

r9 11111110111111101100010101101101010111010101001011111100000

Qhat9 1111101 and Shat 001 set q10 = 0

A10 101001011100001010000

B10 101001011100001001111

caddb9 0000000000000000000000

==

10

SumN3 11110011111000001100010110110001011001010000101111101000000

CarryN3 000010000001101001010000000000100000100000100000000001000000

caddb 0000000000000000000000

+

SumN5 11111011111110101001010110110101011101010100101111100000000

CarryN5 00000000000000001000000000000000000000000000000000010000000

r10 1111101111111011000101011011010101110101010010111111000000

Qhat10 1110111 and Shat 001 set q11 = -1

A11 1010010111000010011111

B11 1010010111000010011110

caddb10 00010011110000000100000110001001001101110100101111000110100O

==

11

SumN3 1110111111101010010101101101010111010101001011111000000000

CarryN3 000000000000001000000000000000000000000000000001000000000

caddb 0001001111000000010000011000100100110111010010111100011010000

+

SumN5 1111110000101000110101011100011110111011101110000100110100 0

CarryN5 0000011110000100000000100001000001000100000000111100000000000

r11 00000011101011001101100111101000010000111100011101001101000

Qhat11 0000111 and Shat 001 set q12 = 1

A12 101001011100001001111110 1

B12 1010010111000010011111100

caddb11 111011000011111101111100111011011001000101101000011100101 11

==

12

SumN3 11110000101000110101011100011110111011101110000100110100000

CarryN3 0001111000010000000010000100000100010000000011110000000000001

caddb 1110110000111111011111001110110110010001011010000111001011 1

+

SumN5 000000101000110000111011011100010101111110110101010001101 10

CarryN5 1111100001100110101010010001110101000000110100000110000001 0

r12 111110101111001011100100100011101010000001000010110100111000

Qhat12 1110101 and Shat 001 set q13 = -1

A13 10100101110000100111111 0011

B13 10100101110000100111111 0010

caddb12 00010011110000000100000110001001001101110100101111 0001101000

==

13

SumN3 00001010001100001110110111000101011111101101010100011011000

CarryN3 11100001100110101010010001110101000000110100000110000001000

caddb 00010011110000000100000110001001001101110100101111 0001101000

+

SumN5 11111000011010101100101010100010000100110000000110001011 1000

CarryN5 00000111001000010100101010101010110111011010101100010010000

r13 111111111000110000010101010011001111000010101110001010 01000

Qhat13 1111110 and Shat 001 set q14 = 0

A14 10100101110000100111111001100

B14 10100101110000100111111001011

caddb13 00000000000000000000000000000

==

14

SumN3 11100001101010110010101010001000010011000000011000101110 0000

CarryN3 00011100100001010010101010101011011101101010110001001000000

caddb 00000000000000000000000000000

+

SumN5 11111101001011100000000000010001100111010101000000000101 00000

CarryN5 00000001000000100101010100010000100010000000110001001 0000000

r14 11111110001100000101010100110011110000101011100010100100000

Qhat14 1111100 and Shat 001 set q15 = 0

A15 1010010111000010011111100110000

B15 1010010111000010011111100101111

caddb14 0000000000000000000000000000000

==

15

SumN3 1111010010111000000000000100011001110101010000000001010000000

CarryN3 0000010000001001010101000100001000100000011000100100000000

caddb 000000000000000000000000000000

+

SumN5 1111000010110001010101001100111011001010111000100001000000

CarryN5 0000100000001000000000000000000000100000000000001000000000

r15 11111000110000010101010011001111000010101110001010010000000

Qhat15 1110001 and Shat 001 set q16 = -1

A16 10100101110000100111111001011111

B16 10100101110000100111111001011110

caddb15 00010011110000000100000110001001001101110100101111000110100 0

==

16

SumN3 11000010110001010101001100111011001010111000100001000000000

CarryN3 0010000001000000000000000000000001000000000000001000000000000

caddb 00010011110000000100000110001001001101110100101111000110100 0

+

SumN5 11110001010001011101000000101000010001010001110111001101000

CarryN5 000001011000000000000011000100110010101010000010000000000000

r16 1111011011000101110101100100111010011010001000011100110100 0

Qhat16 1101101 and Shat 001 set q17 = -2

A17 1010010111000010011111100101111101 0

B17 1010010111000010011111100101111100 1

caddb16 00100111100000001000001100010010011011101001011110001101000 0

==

17

SumN3 11000101000101110100000001010000100010100011101110011010000 0

CarryN3 000101100000000000001100010011001010101000001000000000000000

caddb 00100111100000001000001100010010011011101001011110001101000

+

SumN5 1111010010010110010111100001110010011101010010000010111000

CarryN5 000011100000000100000000101000010101010000110111000100000000

r17 0000001010011000010111110101111101000101101101100100111000

Qhat17 0000101 and Shat 001 set q18 = 1

A18 10100101110000100111111001011111101001

B18 10100101110000100111111001011111101000

caddb17 11101100001111110111110011101101100100010110100001110010111

===

18

SumN3 1101001001011001011110000111001001110101001000001011100000

CarryN3 001110000000100000000101000010101010000110111000100000000 1

caddb 11101100001111110111110011101101100100010110100001110010111

+

SumN5 0000011001101110000000011001010101000101111110000010010101 10

CarryN5 111100000001100101111100011010101011000100101000101100000010

r18 1111011010100000111110100110101010101000010000011010101100 0

Qhat18 1101100 and Shat 001 set q19 = -2

A19 101001011100001001111110010111110100010

B19 101001011100001001111110010111110100001

caddb18 00100111100000001000001100010010011011101001011110001101000 0

===

19

SumN3 000110011011100000000110010101010001011111100000100101011000

CarryN3 110000000110010111110001101010101100010010100010110000001000

caddb 00100111100000001000001100010010011011101001011110001101000

+

SumN5 1111111011110010111000110010010001000011101010111011000000

CarryN5 000000110001001000001100101010110011101010001010000101100000

r19 000000100000010011101111110011110111111000110101110001100000

Qhat19 0000011 and Shat 001 set q20 = 0

A20 10100101110000100111111001011111010001000

B20 10100101110000100111111001011111010000111

caddb19 0000000000000000000000000000000000000000

==

20

SumN3 1111101111001011100011001001000100001110101011101100000000

CarryN3 000011000100100000110010101011001110101000101000010110000000

caddb 0000000000000000000000000000000000000000

+

SumN5 1111011110000011101111100011110111100100100001101001100000

CarryN5 000100001001000000000000100000000000101000101000010000000000

r20 000010000001001110111111001111011111100011010111000110000000

Qhat20 0010000 and Shat 001 set q21 = 2

A21 1010010111000010011111100101111101000100010

B21 1010010111000010011111100101111101000100001

caddb20 110110000111111011111001110110110010001011010000111001011111

==

21

SumN3 1101111000001110111110001111011110010010000110100110000000

CarryN3 010000100100000000000010000000000010100010100001000000000001

caddb 110110000111111011111001110110110010001011010000111001011111

+

SumN5 0100010000110000000000101001011001110000110001000100001011 10

CarryN5 101101001001110111110001101001100010010010100100110000000010

r21 11111000110011011111011011010011000001100010110101000110000

Qhat21 1110001 and Shat 001 set q22 = -1

A22 101001011100001001111110010111110100010000111

B22 101001011100001001111110010111110100010000110

caddb21 000100111100000001000001100010010011011101001011110001101000

==

22

SumN3 00010000110000000001010010110011100001100010001000010111000

CarryN3 11010010011101111100011010011000100100101001001100000001000

caddb 000100111100000001000001100010010011011101001011110001101000

+

SumN5 11010001011101110101000100111001011110100010011010011011000

CarryN5 001001011000000010000110100100101000011010010011000001010000

r22 11110110111110000101111001011110100001110100110010100101000

Qhat22 1101101 and Shat 001 set q23 = -2

A23 10100101110000100111111001011111010001000011010

B23 10100101110000100111111001011111010001000011001

caddb22 001001111000000010000011000100100110111010010111100011010000

==

23

SumN3 01000101110111010100010011100101111010001001101001101100000

CarryN3 1001011000000010000110100100101000011010010011000000101000000

caddb 001001111000000010000011000100100110111010010111100011010000

+

SumN5 11110100010110000111011001010101000000010010110101011110000

CarryN5 00001111000010100000100101001001111110010011010001010000000

r23 00000011011000100111111110011110111110100110000110101110000

Qhat23 0000110 and Shat 001 set q24 = 1

A24 1010010111000010011111100101111101000100001101001

B24 10100101110000100111111001011111010001000011010000

caddb23 111011000011111101111100111011011001000101101000011100101111

====================================

24

SumN3 1101000101100001110110010101010000000100101101010111100000 0

CarryN3 00111100001010000010010100100111111001001101000101000000001

caddb 111011000011111101111100111011011001000101101000011100101111

+

SumN5 0000000101110110100000001001111001110001000011000100101011 0

CarryN5 11111000010100101111101011001011000010011110001011100000010

r24 111110011100100101111011011010010111101011101111001010110 00

Qhat24 1110010 and Shat 001 set q25 = -1

A25 10100101110000100111111001011111010001000011010001 1

B25 10100101110000100111111001011111010001000011010001 0

caddb24 000100111100000001000001100010010011011101001011110001101000

====================================

25

SumN3 00000101110110100000001001111001110001000011000100101011000

CarryN3 11100001010010111110101100101100001001111000101110000001000

caddb 000100111100000001000001100010010011011101001011110001101000

+

SumN5 11110111010100010110101001000111100011010010110100100111000

CarryN5 00000011100101010000001100111000011001101001001110001001000 0

r25 11111010111001100111000010111000010110100101010000111001000

Qhat25 1110101 and Shat 001 set q26 = -1

A26 101001011100001001111110010111110100010000110100010 11

B26 101001011100001001111110010111110100010000110100010 10

caddb25 000100111100000001000001100010010011011101001011110001101000

====================================

26

SumN3 110111101010001011010100100011110001101001011010010011100000

CarryN3 000011100101010000011001110000110011010010011100010010000000

caddb 000100111100000001000001100010010011011101001011110001101000

+

SumN5 110000001101000100110011110011110110111010111111101011001000

CarryN5 001111101000100100010010001001000110100100101001000110000000

r26 111111110101101001000101111100111101011111101000011100010000

Qhat26 1111110 and Shat 001 set q27 = 0

A27 10100101110000100111111001011111010001000001101000101100

B27 10100101110000100111111001011111010001000001101000101011

caddb26 00000000000000000000000000000000000000000000000000000000

==

27

SumN3 000000110100010011001111001111011011101011111110101100100000

CarryN3 111110100010010001001000100100011010010010100100011000000000

caddb 00000000000000000000000000000000000000000000000000000000

+

SumN5 111110010110000010000111101011000001111001011001000001000000

CarryN5 000001000000100010010000000100011010000010100100011000000000

r27 111111010110100100010111110011110101111110100001110001000000

Qhat27 1111010 and Shat 001 set q28 = 0

A28 1010010111000010011111100101111101000100000110100010110000

B28 1010010111000010011111100101111101000100000110100010101111

caddb27 0000000000000000000000000000000000000000000000000000000000

==

28

SumN3 111001011000001000011110101100000111100101100100000100000000

CarryN3 000100000001000100100000001000110100000101001000110000000000

caddb 00000000000000000000000000000000000000000000000000000

+

SumN5 11110101101000000101111000111101011111000100011100010000000

CarryN5 0000000000000100000000001000000000000000100100000000000000000

r28 111101011010010001011111001111010111111010000111000100000000

Qhat28 1101011 and Shat 001 set q29 = -2

A29 10100101110000100111111001011111010001000011010001010111110

B29 10100101110000100111111001011111010001000011010001010111101

caddb28 00100111100000010000011000100100110111010010111100011010000

==

29

SumN3 11010110100000010111100011110101111100010001110001000000000

CarryN3 0000000000010000000000100000000000000010010000000000000000000

caddb 00100111100000010000011000100100110111010010111100011010000

+

SumN5 11110001000100000111101011010001001001010011001101011010000

CarryN5 000011010000000100000100001001001101100100001100000000000000

NUMBER 1 IS .01110100111010100100101010001100000101010100110010011 = 0.456700

NUMBER 2 IS .00111100000001000001100010010011011101001011110001101000 = 0.234500

RESULT IS .001011100001001111110010111110100010000110100010101111100 = 0.179992

REAL RESULT IS .00101110000100111111001011111010001000011010001010110000000 =

1.179992

ERROR IS 0.000000

Chapter 7

APPENDIX B

With the C square-root simulator coded, when the number 0.65 is square-rooted, we get this step by step SRT radix-4 square-root algorithm's results. At the end of the simulation, computed result is compared with the CPU's own result. The name of the signals are exactly same as used in Figure 4.1.

y = 1010011001100110011001100110011001100110011001101 = 0.65

X = 1111101001100110011001100110011001100110011001100110011001101

r0 1111101001100110011001100110011001100110011001100110011001101

Qhat0 1110100 and Shat 100 set q1 = -1

A1 011

B1 010

F0 000111

==

1

SumN3 111010011001100110011001100110011001100110011001100110100

CarryN3 0000000000000000000000000000000000000000000000000000000

F 000111

+

SumN5 111101011001100110011001100110011001100110011001100110100

CarryN5 0001000000000000000000000000000000000000000000000000000

r1 000001011001100110011001100110011001100110011001100110100

Qhat1 0001011 and Shat 100 set q2 = 1

A2 01101

B2 01100

F1 11100111

==

2

SumN3 1101011001100110011001100110011001100110011001100110011010000

CarryN3 01000000000000000000000000000000000000000000000000000000000000

F 11100111

+

SumN5 0111000101100110011001100110011001100110011001100110011010000

CarryN5 10001100000000000000000000000000000000000000000000000000000000

r2 1111110101100110011001100110011001100110011001100110011010000

Qhat2 1111010 and Shat 101 set q3 = 0

A3 0110100

B3 0110011

F2 0000000

==

3

SumN3 1100010110011001100110011001100110011001100110011001101000000

CarryN3 00110000000000000000000000000000000000000000000000000000000000

F 0000000

+

SumN5 1111010110011001100110011001100110011001100110011001101000000

CarryN5 00000000000000000000000000000000000000000000000000000000000000

r3 1111010110011001100110011001100110011001100110011001101000000

Qhat3 1101011 and Shat 101 set q4 = -2

A4 011001110

B4 011001101

F3 001100111100

==

4

SumN3 110101100110011001100110011001100110011001100110100000000

CarryN3 0000000000000000000000000000000000000000000000000000000000

F 001100111100

+

SumN5 111001011010011001100110011001100110011001100110100000000

CarryN5 0010010010000000000000000000000000000000000000000000000000

r4 00001010001001100110011001100110011001100110011010000000

Qhat4 0010100 and Shat 100 set q5 = 2

A5 01100111010

B5 01100111001

F4 11001100011100

==

5

SumN3 100101101001100110011001100110011001100110011010000000000

CarryN3 1001001000000000000000000000000000000000000000000000000000

F 11001100011100

+

SumN5 110010001110100110011001100110011001100110011010000000000

CarryN5 0010110000100000000000000000000000000000000000000000000000

r5 111101010000100110011001100110011001100110011010000000000

Qhat5 1101001 and Shat 100 set q6 = -2

A6 0110011100110

B6 0110011100101

F5 0011001110011100

==

6

SumN3 001000111010011001100110011001100110011001101000000000000

CarryN3 1011000010000000000000000000000000000000000000000000000000

F 0011001110011100

+

SumN5 1010000010111010011001100110011001100110011010000000000000

CarryN5 0110011100001000000000000000000000000000000000000000000000

r6 00000111110000100110011001100110011001100110100000000000000

Qhat6 0001111 and Shat 100 set q7 = 1

A7 011001110011001

B7 011001110011000

F6 111001100011001111

==

7

SumN3 1000001011101001100110011001100110011001101000000000000000

CarryN3 1001110000100000000000000000000000000000000000000000000000

F 111001100011001111

+

SumN5 1111100011111010010110011001100110011001101000000000000000

CarryN5 0000110001000011000000000000000000000000000000000000000000

r7 000001010011110101011001100110011001100110100000000000000000

Qhat7 0001010 and Shat 100 set q8 = 1

A8 01100111001100101

B8 01100111001100100

F7 11100110001100110111

==

8

SumN3 11100011111010010110011001100110011001101000000000000000000

CarryN3 0011000100001100000000000000000000000000000000000000000000

F 11100110001100110111

+

SumN5 0011010011010110000101100110011001100110100000000000000000

CarryN5 11000110010100101100000000000000000000000000000000000000

r8 111110110010100011010110011001100110011010000000000000000

Qhat8 1110110 and Shat 100 set q9 = -1

A9 0110011100110010011

B9 0110011100110010010

F8 00011001110011001001111

==

9

SumN3 1101001101011000010110011001100110011010000000000000000000

CarryN3 00011001010010110000000000000000000000000000000000000000

F 00011001110011001001111

+

SumN5 1101001111011111110001011001100110011010000000000000000000

CarryN5 00110010100100000011000000000000000000000000000000000000

r9 0000011001101111111101011001100110011010000000000000000000

Qhat9 0001100 and Shat 100 set q10 = 1

A10 01100111001100100 1101

B10 0110011100110010011 00

F9 111001100011001101100111

==

10

SumN3 01001111011111110001011001100110011010000000000000000000000

CarryN3 1100101001000000110000000000000000000000000000000000000000

F 111001100011001101100111

+

SumN5 0110001100001100101100010110011001101000000000000000000000

CarryN5 1001110011100110100011000000000000000000000000000000000000

r10 11111111111100110011110101100110011010000000000000000000000

Qhat10 1111111 and Shat 100 set q11 = 0

A11 01100111001100100110100

B11 01100111001100100110011

F10 0000000000000000000000

==

11

SumN3 1000110000110010110001011001100110100000000000000000000000

CarryN3 0111001110011010001100000000000000000000000000000000000000

F 000000000000000000000

+

SumN5 1111111110101000111101011001100110100000000000000000000000

CarryN5 0000000000100100000000000000000000000000000000000000000000

r11 11111111110011001111010110011001101000000000000000000000

Qhat11 1111111 and Shat 100 set q12 = 0

A12 0110011100110010011010000

B12 0110011100110010011001111

F11 000000000000000000000000

==

12

SumN3 1111111010100011110101100110011010000000000000000000000000

CarryN3 0000000010010000000000000000000000000000000000000000000000

F 0000000000000000000000000

+

SumN5 1111111000110011110101100110011010000000000000000000000000

CarryN5 0000000100000000000000000000000000000000000000000000000000

r12 11111111001100111101011001100110100000000000000000000000000

Qhat12 1111110 and Shat 100 set q13 = 0

A13 011001110011001001101000000

B13 011001110011001001100111111

F12 000000000000000000000000000

==

13

SumN3 1111100011001111010110011001101000000000000000000000000000

CarryN3 0000010000000000000000000000000000000000000000000000000000

F 000000000000000000000000000

+

SumN5 1111110011001111010110011001101000000000000000000000000000

CarryN5 0000000000000000000000000000000000000000000000000000000000

r13 1111110011001111010110011001101000000000000000000000000000

Qhat13 1111001 and Shat 100 set q14 = 0

A14 01100111001100100110100000000

B14 01100111001100100110011111111

F13 00000000000000000000000000000

==

14

SumN3 1111001100111101011001100110100000000000000000000000000000

CarryN3 0000000000000000000000000000000000000000000000000000000000

F 00000000000000000000000000000

+

SumN5 1111001100111101011001100110100000000000000000000000000000

CarryN5 0000000000000000000000000000000000000000000000000000000000

r14 1111001100111101011001100110100000000000000000000000000000

Qhat14 1100110 and Shat 100 set q15 = -2

A15 0110011100110010011001111111110

B15 0110011100110010011001111111101

F14 0011001110011001001100111111111100

==

15

SumN3 110011001111010110011001101000000000000000000000000000000

CarryN3 0000000000000000000000000000000000000000000000000000000000

F 0011001110011001001100111111111100

\+

SumN5 111111110110110010101010010111110000000000000000000000000

CarryN5 0000000100100010001000110100000000000000000000000000000000

r15 00000000100011101100110110011111000000000000000000000000

Qhat15 0000000 and Shat 100 set q16 = 0

A16 0110011100110010011001111111111000

B16 0110011100110010011001111111110111

F15 00000000000000000000000000000000

==

16

SumN3 1111110110110010101010010111110000000000000000000000000000

CarryN3 0000010010001000100011010000000000000000000000000000000000

F 00000000000000000000000000000000

\+

SumN5 1111100100111010001001000111110000000000000000000000000000

CarryN5 0000100100000000100010010000000000000000000000000000000000

r16 0000001000111011001101100111110000000000000000000000000000

Qhat16 0000100 and Shat 100 set q17 = 0

A17 011001110011001001100111111111100000

B17 011001110011001001100111111111011111

F16 000000000000000000000000000000000000

==

17

SumN3 1110010011101000100100011111000000000000000000000000000000

CarryN3 0010010000000010001001000000000000000000000000000000000000

F 000000000000000000000000000000000000

+

SumN5 1100000011101100110110011111000000000000000000000000000000

CarryN5 0100100000000000000000000000000000000000000000000000000000

r17 0000100011101100110110011111000000000000000000000000000000

Qhat17 0010001 and Shat 100 set q18 = 1

A18 0110011100110010011001111111110000001

B18 0110011100110010011001111111110000000

F17 1110011000110011011001100000000011111111

==

18

SumN3 0000001110110011011001111100000000000000000000000000000000

CarryN3 0010000000000000000000000000000000000000000000000000000000

F 1110011000110011011001100000000011111111

+

SumN5 1100010110000000000000001110000001111111000000000000000000

CarryN5 0100010001100110110011000000000000000000000000000000000000

r18 0000100111100110110011011100000011111110000000000000000000

Qhat18 0010011 and Shat 100 set q19 = 2

A19 0110011100110010011001111111111000000110

B19 0110011100110010011001111111111000000101

F18 11001100011001101100110000000000111111101100

==

19

SumN3 0001011000000000000000011100000011111110000000000000000000

CarryN3 0001000110011011001100000000000000000000000000000000000000

F 11001100011001101100110000000000111111101100

+

SumN5 1100101111111101111110110000001000000111000000000000000000

CarryN5 0010100000000100000010000000001111110000000000000000000000

r19 1111010000000010000000110000010111110111000000000000000000

Qhat19 1100111 and Shat 100 set q20 = -2

A20 0110011100110010011001111111111000000010110

B20 0110011100110010011001111111111000000010101

F19 00110011100110010011001111111111000000001011100

==

20

SumN3 0010111111110111111011000000100000011100000000000000000000

CarryN3 1010000000001000000100000000011111100000000000000000000000

F 00110011100110010011001111111111000000001011100

+

SumN5 1011110001111110111111111111100111011001110000000000000000

CarryN5 0100011100100010010000000001110000001000000000000000000000

r20 0000001110100001010000000000101011110000111000000000000000

Qhat20 0000110 and Shat 100 set q21 = 1

A21 01100111001100100110011111111110000001011001

B21 01100111001100100110011111111110000001011000

F20 111001100011001101100110000000000111111101001111

==

21

SumN3 1111000111111011111111111110011101100111000000000000000000

CarryN3 0001110010001001000000000111000000100000000000000000000000

F 111001100011001101100110000000000111111101001111

+

SumN5 0000101101000001100110011001011110111010001111000000000000

CarryN5 1110100101110110110011001100000011001010000000000000000000

r21 1111010010111000011001100101100010000100001111000000000000

Qhat21 1101001 and Shat 100 set q22 = -2

A22 01100111001100100110011111111100000010110001 0

B22 011001110011001001100111111111000000101100001

F21 0011001110011001001100111111111000000101100011 00

==

22

SumN3 0010110100000110011001100101111011101000111100000000000000

CarryN3 10100101110110110011001100000011001010000000000000000000000

F 0011001110011001001100111111111000000101100011 00

\+

SumN5 1011101101000100011001101010001111000101011111000000000000

CarryN5 01001011001101100110011010111100010100010000000000000000000

r22 00000110011110101100110101100000000101100111110000000000 0

Qhat22 0001100 and Shat 100 set q23 = 1

A23 0110011100110010011001111111110000001011000100 1

B23 0110011100110010011001111111110000001011000100 0

F22 111001100011001101100110000000000111111010011101111

==

23

SumN3 11101101000100011001101010001111000101011111000000000000000

CarryN3 0010110011011001100110101111000101000100000000000000000000

F 111001100011001101100110000000000111111010011101111

\+

SumN5 00100111111110110110011001111110101011001100101111 0000000

CarryN5 1101100000100011001101010000001010101010011000000000000000

r23 0000000000011110100110111000000101010111001010111100000000

Qhat23 1111111 and Shat 100 set q24 = 0

A24 0110011100110010011001111111110000001011000100100

B24 0110011100110010011001111111110000001011000100011

F23 0000000000000000000000000000000000000000000000000

==

24

SumN3 1001111111101101100110011111010101100110010111100000000

CarryN3 0110000010001100110101000000101010101001100000000000000000

F 000000000000000000000000000000000000000000000000000

+

SumN5 1111111101100001010011011111000000011010101011110000000000

CarryN5 0000000100011001001000000000101010100001000000000000000000

r24 000000000111101001101110000001010101110010101111000000000

Qhat24 0000000 and Shat 100 set q25 = 0

A25 0110011100110010011001111111110000001011000100100000

B25 0110011100110010011001111111110000001011000100011111

F24 000000000000000000000000000000000000000000000000000

==

25

SumN3 1111110110000101001101111110000000110101010111100000000000

CarryN3 000001000110010010000000010101010000100000000000000000000

F 0000000000000000000000000000000000000000000000000000

+

SumN5 1111100111100001101101111001010101100010101111000000000000

CarryN5 0000100000000100000000000010000000000100000000000000000000000

r25 0000000111101001101110000001010101110010101111000000000000

Qhat25 0000011 and Shat 100 set q26 = 0

A26 011001110011001001100111111111000000101100010010000000

B26 011001110011001001100111111111000000101100010001111111

F25 00000000000000000000000000000000000000000000000000000

==

26

SumN3 11100111100001101101111001010101100010101111000000000000

CarryN3 001000000010000000000001000000000010000000000000000000000

F 00000000000000000000000000000000000000000000000000

+

SumN5 11000111101001101101110001010101110010101111000000000000

CarryN5 010000000000000000001000000000000000000000000000000000000

r26 000001111010011011100000010101011100101011110000000000000

Qhat26 0001111 and Shat 100 set q27 = 1

A27 01100111001100100110011111111100000010110001001000000001

B27 01100111001100100110011111111100000010110001001000000000

F26 11100110001100110110011000000000011111101001110110111111111

===

27

SumN3 000111101001101101110001010101110010101111000000000000000

CarryN3 000000000000000000010000000000000000000000000000000000000

F 11100110001100110110011000000000011111101001110110111111111

+

SumN5 11111000101010000000011101010111110101101111101101111111

CarryN5 000011000010011011100000000000000010100100000000000000000

r27 00000100110011101110011101011000001010001111101101111111

Qhat27 0001001 and Shat 100 set q28 = 1

A28 011001110011001001100111111111000000101100010010000000101

B28 011001110011001001100111111111000000101100010010000000100

F27 111001100011001101100110000000000111111010011101101111110111

===

28

SumN3 111000101010000000011101010111110101101111101101111111100

CarryN3 001100001001101110000000000000000101001000000000000000000

F 111001100011001101100110000000000111111010011101101111110111

+

SumN5 0011010000001000111110110101111011101110110101101000000010

CarryN5 110001010110011000001000000000101011001001010010111111000

NUMBER IS .1010011001100110011001100110011001100110011001101 = 0.650000

RESULT IS .11001110011001001100111111111000000101100010010000000101 = 0.806226

REAL RESULT IS .11001110011001001100111111111000000101100010010000000000 = 0.806226

ERROR IS 0.000000

BIBLIOGRAPHY

[1] A. Moshovos and G. Sohi. Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling. Proceedings of the IEEE, vol. 89, no. 11, Noveber 2001, pp. 1560-1575.

[2] M. Ercegovac and T. Lang. Division and Square-Root : Digit-Recurrence Algorithms and Implementations.

[3] J. E. Robertson. A New Class of Digital Division Methods, IRE Trans. Electronic Computers, vol. EC-7, pp. 218222, Sept. 1958.

[4] K. D. Tocher. Techniques of Multiplication and Division for Automatic Binary Computers, Quart. J. Mech. Appl. Math., vol. 11, pt. 3, pp. 364384, 1958.

[5] Y. He and C. Ding. Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications. *Journal of Supercomputing*, 18:259–277, 2001.

[6] Y. Hida, X.S. Li, and D.H. Bailey. Algorithms for Quad-Double Precision Floating Point Arithmetic. In *Proceedings of 15th IEEE Symposium on Computer Arithmetic*, pages 155–162, 2001.

[7] *DRAFT IEEE Standard for Floating-Point Arithmetic*, 2005. Available from: http://754r.ucbtest.org/.

[8] M. Suzuoki et al. A Microprocessor with a 128-bit CPU, Ten Floating-Point MAC's, Four Floating-Point Dividers, and an MPEG-2 Decoder. *IEEE Journal of Solid-State Circuits*, 34(11):1608–1618, Nov. 1999.

[9] D.C. Bossen, J.M. Tendler, and K. Reick. POWER4 System Design for High Reliability. *IEEE Micro*, 22:16–24, 2002.

[10] A. Naini, A. Dhablania, W. James, and D.D. Sarma. 1 GHz HAL SPARC64 Dual Floating Point Unit with RAS Features. In *Proceedings of 15th Symposium on Computer Arithmetic*, pages 173–183, 2001.

[11] J. Tyler, J. Lent, A. Mather, and H. Nguyen. AltiVec: Bringing Vector Technology to the PowerPC Processor Family. In *IEEE International Performance, Computing and Communications Conference*, pages 437–444, November 1999.

[12] A. Akkas and M.J. Schulte. Dual-Mode Floating-Point Multiplier Architectures with Parallel Operations. *Journal of Systems Architecture*, 52:549–562, 2006.

[13] G. Even, S.M. Mueller, and P.-M. Seidel. A Dual Precision IEEE Floating-Point Multiplier. *Integration, the VLSI journal*, 29:167–180, 2000.

[14] A. Akkas. Dual-Mode Quadruple Precision Floating-Point Adder. In *9th Euromicro Conference on Digital System Design*, pages 211–220, 2006.

[15] A. Akkas. Dual-Mode Floating-Point Adder Architectures. submitted to Journal of Systems Architecture.

[16] S.F. Oberman and M.J. Flynn. Design Issues in Division and Other Floating-Point Operations. *IEEE Transaction on Computers*, 46:154–161, 1997.

[17] M.D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[18] M.D. Ercegovac and T. Lang. On-The-Fly Rounding. *IEEE Transaction on Computers*, 1992.

[19] *ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*, 1985.

[20] R.W. Steward, R. Chapman, and T. Durrani. The Square-Root in Signal Processing. Proceedings of Real Time Signal Processing XII, 1989 pp. 89-100.

[21] V. K. Jain, G. E. Perez, and J. M. Wills. Novel Reciprocal and Square-Root VLSI Cell: Architecture and Application to Signal Processing. International Conference on Acoustics, Speech and Signal Processing, vol.2, 1991, pp. 1201-1204.

[22] S. Hollasch. IEEE Standart 754 Floating-Point Numbers. [Online] Available: http://stevehollasch.com/cgindex/coding/ieeefloat.html.

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

[24] A. Akkas. Instruction Set Enhancements for Reliable Computations. Ph.D. disertation, Graduate Department of Computer Science, Lehigh University, August 2001.

[25] 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.

[26] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. Computing Surveys, March 1991.

[27] A. Akkas and M.J. Schulte. A Quadruple Precision and Dual Double Precision Floating-Point Multiplier. Euromicro Symposium on Digital System Design, 2003, pp. 76-81.

[28] 754R Working Group Some Proposals for Revising ANSI/IEEE Std 754-1985, IEEE. [ONLINE]. Available: http://754r.ucbtest.org.

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

[30] C. Severance An Interview with the Old Man of Floating-Point. February 1998. [ONLINE]. Available: http://cch.loria.fr/documentation/IEEE754/wkahan/754story.html.

[31] E. M. Schwarz and C. A. Krygowski The S/390 G5 Floating-Point Unit. [ONLINE]. Available: http://www.research.ibm.com/journal/rd/435/schwarz.pdf.

[32] 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.

[33] G. Even and W. J. Paul On the Design of IEEE Compliant Floating-Point Units. IEEE Transactions on Computers, vol. 49, no. 5, pp. 398-413, May 2000.

[34] 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.

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

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

[37] 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..

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

[39] 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.

[40] 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.

[41] H. R. Srinivas and K. K. Parhi A fast radix 4 division algorithm..

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

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

[44] 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. 16061611, December 1992.

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

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

[47] M. J. Schulte and E. E. Swartzlander, Jr A Hardware Design and Arithmetic Algorithms for a Variable-Precision, Interval Arithmetic Coprocessor. Proceedings of the 12th Symposium on Computer Arithmetic, July 1995, pp. 163-171.

[48] J. E. Stine Design Issues for Accurate and Reliable Arithmetic. Ph.D. dissertation, Graduate Department of Computer Science, Lehigh University, 2001.

[49] J. E. Stine and M. J. Schulte A Combined Interval and Floating-Point Divider. Thisrty-Second Asilomar Conf. on Signals, Systems and Computers, November 1998.

[50] 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.

VITA

Aytunç İşseven, the son of Tuncay and Ayşegül İşseven, was born in İstanbul, Türkiye on January 2nd, 1984. At the age of 5, his father send him to elementary school close to their home just to give him an idea. He liked the school and his teacher found him successful so he stayed. His mother wanted him to study at KOC high school so he attended to transfer exams and passed the exam. He was in the IB (International Baccalaureate) Math-Science class. This also meant the beginning of an undergraduate study in Ny, USA at Rensselaer Polytechnic Institute as a Computer and Systems Engineer. During his undergraduate studies, he completed his summer interns in 2002, at Tikle, İstanbul. After graduating on May 2004 from RPI, he applied to KOC University, İstanbul, Turkey, where he is currently pursuing his Masters of Science degree in Electrical and Computer Engineering. During his studies, he worked on the implementation and design of dual-mode division and square-root units in which two parallel double-precision operations or one quadruple-precision operations can be done.

PUBLICATION

- Aytunç İşseven and Ahmet Akkaş, "A Dual-Mode Quadruple Precision Floating-Point Divider" Asilomar Conference on Signals, Systems and Computers ($ASIL'06$), Pacific Grove, CA, USA 29 Oct - 1 Nov, 2006, pages 1697-1701 1424407850/06/$20.00

CONTACT INFORMATION

- E-mail address : aytunch@gmail.com