Algorithms and Architectures for Decimal Transcendental Function Computation

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
University of Saskatchewan
Saskatoon, Saskatchewan, Canada

By
Dongdong Chen

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Electrical and Computer Engineering
57 Campus Drive
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5A9

Abstract

Nowadays, there are many commercial demands for decimal floating-point (DFP) arithmetic operations such as financial analysis, tax calculation, currency conversion, Internet based applications, and e-commerce. This trend gives rise to further development on DFP arithmetic units which can perform accurate computations with exact decimal operands. Due to the significance of DFP arithmetic, the IEEE 754-2008 standard for floating-point arithmetic includes it in its specifications. The basic decimal arithmetic unit, such as decimal adder, subtracter, multiplier, divider or square-root unit, as a main part of a decimal microprocessor, is attracting more and more researchers' attentions. Recently, the decimal-encoded formats and DFP arithmetic units have been implemented in IBM's system z900, POWER6, and z10 microprocessors.

Increasing chip densities and transistor count provide more room for designers to add more essential functions on application domains into upcoming microprocessors. Decimal transcendental functions, such as DFP logarithm, antilogarithm, exponential, reciprocal and trigonometric, etc, as useful arithmetic operations in many areas of science and engineering, has been specified as the recommended arithmetic in the IEEE 754-2008 standard. Thus, virtually all the computing systems that are compliant with the IEEE 754-2008 standard could include a DFP mathematical library providing transcendental function computation. Based on the development of basic decimal arithmetic units, more complex DFP transcendental arithmetic will be the next building blocks in microprocessors.

In this dissertation, we researched and developed several new decimal algorithms and architectures for the DFP transcendental function computation. These designs are composed of several different methods: 1) the decimal transcendental function computation based on the table-based first-order polynomial approximation method; 2) DFP logarithmic and antilogarithmic converters based on the decimal digit-recurrence algorithm with selection by rounding; 3) a decimal reciprocal unit using the efficient table look-up based on Newton-Raphson iterations; and 4) a first radix-100 division unit based on the non-restoring algorithm with pre-scaling method. Most decimal algorithms and architectures for the DFP transcendental function computation developed in this dissertation have been the first at-

tempt to analyze and implement the DFP transcendental arithmetic in order to achieve faithful results of DFP operands, specified in IEEE 754-2008.

To help researchers evaluate the hardware performance of DFP transcendental arithmetic units, the proposed architectures based on the different methods are modeled, verified and synthesized using FPGAs or with CMOS standard cells libraries in ASIC. Some of implementation results are compared with those of the binary radix-16 logarithmic and exponential converters; recent developed high performance decimal CORDIC based architecture; and Intel's DFP transcendental function computation software library. The comparison results show that the proposed architectures have significant speed-up in contrast to the above designs in terms of the latency. The algorithms and architectures developed in this dissertation provide a useful starting point for future hardware-oriented DFP transcendental function computation researches.

Acknowledgements

It is a pleasure to express my sincere appreciation to the people who have assisted me throughout the years of research that have led to this Ph.D dissertation at University of Saskatchewan.

First and foremost, I sincerely thank my supervisor, Dr. Seok-Bum Ko, who has supported me throughout this dissertation with his expertise, understanding, and patience whilst allowing me the room to work in my own way. Without his constant guidance, advice and encouragement, this dissertation would not have been possible.

Second, I would like to thank the members of my dissertation committee, Dr. Daniel Teng, Dr. Khan Wahid and Dr. Derek Eager, for the invaluable suggestions for my Ph.D dissertation. Also, I would like to thank Dr. Tor Aamodt from the University of British Columbia for taking time out from his busy schedule to serve as my external examiner.

Third, I thank all people in the Department of Electrical and Computer Engineering for providing such a great academic environment, in which I can carry out this dissertation. In particular, I would like to thank the faculties and staff of VLSI research group; and thank all workmates for collaborations on several research projects.

Fourth, I would appreciate the anonymous reviewers from *IEEE Symposium on Computer Arithmetic* and *IEEE Transactions on Computer* for their invaluable comments. I am very grateful to the Dr. Ivan Godard for his insightful advice and brilliant idea for several future research projects. I would like to mention Dr. Liang-Kai Wang, Dr. Mark A. Erle and Dr. Álvaro Vázquez for their impressive Ph.D works in the area of the DFP computer arithmetic, which continually inspire me on doing my Ph.D research.

Fifth, I would to deliver my thanks to my friends in 2C60 for their friendship and help; special thanks to my beloved family for their love and support. They always encouraged me and asked me to be patience and work harder, and that is the thing I really kept in mind.

Finally, I am grateful to the College of Graduate Studies and the Department of Electrical and Computer Engineering for providing financial assistance through scholarships that were invaluable to me.

Dedicate to my beloved family

Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application Specific Circuit
BCD	Binary Coded Decimal
BFP	Binary Floating-Point
BID	Binary Integer Decimal
BSPS	Binary Search Partition Scheme
BXP	Binary Fixed-Point
CS	Carry-Save Number System
DCC	Decimal Carry Counter
DCFA	Decimal Carry-free Adder
DCLA	Decimal Carry-look-ahead Adder
DCSA	Decimal Carry-save Adder
DFP	Decimal Floating-Point
DLAU	Decimal Linear Approximation Unit
DLAU	Decimal Logarithmic Arithmetic Unit
DMUL	Decimal Multiply Logic
DPD	Densely Packed Decimal
DPPG	Decimal Partial Product Generation
DSP	Digital Signal Processing
DXP	Decimal Fixed-Point
FLP	Floating-Point Number System
FPGA	Field-Programmable Gate Array
IP	Intellectual Property
LNS	Logarithmic Number System
LOD	Leading One Detector
LSB	Least Significand Bit
LSD	Least Significand Digit
LUTs	Look-up Tables
LZD	Leading Zero Detector
MSB	Most Significand Bit
MSD	Most Significand Digit
MTBDD	Multi-terminal Binary Decision Diagram
MiniBit	Static Bit-width Optimization Approach
NaN	Not-a-Number
ROM	Read Only Memory
SD	Signed-Digit Number System
SIE	Segment Index Encoder
SIMD	Single Instruction Multiple Data
UFB	Uniform Fractional Bit-width
ulp	Unit in the Last Place

Part I

Preface

Chapter 1

Introduction

This chapter discusses the importance of Decimal Floating-Point (DFP) arithmetic and the necessity that existing computer processors support DFP arithmetic. Based on the existing basic decimal arithmetic units, more complex DFP transcendental arithmetic could be built in computer processors to meet the strict requirements on computational speed and accuracy for tomorrow. This motivates the need to create DFP transcendental arithmetic in hardware, which is the focus of this dissertation. Section 1.1 presents an overview of the necessity of decimal arithmetic. Section 1.2 discusses the motivation of this research work. Section 1.3 describes the road-map of this research dissertation. Section 1.4 summarizes research contributions.

1.1 Why Decimal Arithmetic

We will start this chapter with a story shown in a design article in EETimes [1]:

> "If you ask engineers how numbers are represented, stored, and manipulated in computers and calculators, most will reply: "as signed or unsigned binary integers or as binary fixed-point (BXP) or floating-point (BFP) values." (...) And even if one should happen to enquire about Binary Coded Decimal (BCD) representations, the response is almost invariably: "Oh, that went out of style 25 to 30 years ago; no one uses it now." Is that true?"

Decimal number is a whole lot simpler for people to understand and use than binary number, because it is basically based upon the decimal arithmetic that we have learned in school. In the early computers, the decimal arithmetic units were implemented in microprocessors, however, they have not been popular due to its lower computation speed and

larger hardware cost compared with binary arithmetic. The reasons are evident, first, binary arithmetic is more suitable for scientific computations than decimal, due to its mathematical properties and performance advantage; second, the substrate of digital systems is based on two-state transistors so that binary number can be stored more efficiently and processed faster than decimal number in hardware. We realize the advantages of the binary computation, however, we intend to show in this section that the decimal computation still has its significance on many financial, commercial and Internet business applications, such as banking, accounting, tax calculation, currency conversion, insurance, marketing, retail sales, e-commerce and e-banking [2]. The following is more comprehensive reasons for the need of decimal arithmetic.

First of all, numbers in commercial databases are primarily decimal. Dr. M. F. Cowlishaw in his paper [2] cites a survey of commercial databases reported by Tsang [3]. The databases in this survey cover a wide range of applications, including airline systems, banking, financial analysis, insurance, inventory control, management reporting, marketing services, order entry and processing, pharmaceutical, and retail sales. In those databases, over 456,420 columns contained numeric data and of which 55% are decimal, and the further 43.7% are integer types which could have been stored as decimal numbers. Dr. M. F. Cowlishaw concludes that the extensive use of decimal numbers in these commercial databases suggests that it is worthwhile to study how the decimal data are used and how decimal arithmetic should be defined.

Second, most decimal fractional numbers can not be exactly represented or exactly rounded by BFP numbers. In [4], Dr. M. F. Cowlishaw presents several examples to show computation problems. The first example shows that the decimal number 0.1 requires an infinitely recurring binary number (0.00011001...), which can only be approximated to a decimal number (0.09765...) instead of the exact value of 0.1. It is evident that a conversion error between the decimal and binary format can not be avoided, so using BFP to compute decimal is not possible to guarantee the same results as those from decimal arithmetic. Another potential problem may be produced later on when rounding the BFP result after the decimal computation by binary. For example, consider a calculation, 0.70×1.05, using the most widely used double-precision BFP format, the exact result is little less than

0.73499999999999999. This result, rounded to a decimal number with two decimal fraction digits, is 0.73, which is 0.01 less than manual computation result (0.735), rounded to the same fraction digits, 0.74. Although the error in a single operation shown in above examples is very small, these tiny errors may accumulate and lead to a large error after several operations. Dr. M. F. Cowlishaw presents a study [4] which shows that a large telephone billing system can accumulate errors of up to 5 million dollars per year, if using BFP arithmetic (the telco benchmark [5]).

Third, decimal arithmetic could be a significant part of commercial workloads with the increasing use of DFP format. In [2], Dr. M. F. Cowlishaw presents a case study about a new benchmark, designed to model an extreme case such as a telephone company's daily billing application. This case study indicates that the decimal processing overhead in the benchmark could reach over 90% and very time consuming. Therefore, he predicts that this kind of applications would clearly benefit from the hardware component of DFP arithmetic units built in microprocessors. Such a hardware could be two to three orders of magnitude faster than software.

1.2 Motivation

In today's world, few microprocessors have instructions or dedicated hardware components for DFP arithmetic. In a microprocessor without DFP support, decimal numbers are usually processed with two methods. One method is to convert the decimal numbers to binary numbers before computation in binary hardware arithmetic units, and then the binary format results are converted back to decimal format. The other method is to store the data in decimal format and process data using decimal software arithmetic library, such as the Java *BigDecimal* class [6] and the C/C++ *decNumber* library [7]. The first method is error-prone because BFP format can not exactly represent decimal fractions, while the software based method is typically 100 to 1000 times slower than binary arithmetic implemented in hardware [2]. These problems give rise to the need to develop hardware DFP arithmetic components in microprocessors for accurate and fast calculation fully in decimal. Due to the significance of DFP arithmetic, it is included in the specifications of the IEEE 754-2008

standard [8], which defines three decimal data formats that can be used for decimal integer, decimal fixed-point (DXP), and DFP computer arithmetic. For the same reason, the DFP arithmetic units have been implemented in IBM's microprocessors POWER6 [9], system z9 [10] and z10 [11] microprocessors.

Most microprocessors in computer systems usually include the basic arithmetic units, such as adder, subtracter, multiplier, divider and square-root unit ($+$, $-$, $\times$, $\div$ and $\sqrt{}$). These basic decimal arithmetic units, as the main components of a decimal microprocessor, have been designed and implemented to be compliant with the DFP arithmetics defined in IEEE 754-2008 standard by recent works. Examples include the decimal adders in [12, 13, 14, 15, 16, 17, 18, 19], the decimal multipliers [20, 21, 22, 23, 24], the decimal dividers in [25, 26, 27, 28, 29] and the decimal square-root unit in [30]. A complete survey of hardware designs for the basic decimal arithmetic is summarized in [31].

The transcendental functions, such as logarithm, antilogarithm, exponential, and trigonometric, defined in the Dr. J-M Muller's book [32], are useful arithmetic concepts in many areas of science and engineering. Some applications, such as computer 3D graphics, scientific computing, artificial neural networks, logarithmic number system (LNS), digital signal processing (DSP) [33, 34, 35, 36, 37] are implemented in hardware by using binary transcendental arithmetics to replace the basic computer arithmetic units. For instance, the multiplication and division can be simplified to the level of addition and subtraction by using logarithmic units [33]. The decimal transcendental function computation is also very useful for some specific applications, such as some computations used in financial applications in banks [38] (eg. the compound interest computation), the scientistic decimal calculator [39], and some pocket computers [40]. Furthermore, with the continuous reduction of the size of the transistor and the scale of the integration, the decimal transcendental arithmetic units are more likely to be cheap enough to be implemented in microprocessors, and its performance could be close to the performance of the binary units. And then, maybe the decimal transcendental arithmetic units will finally replace or co-exist with binary units in all applications because of the human preference for the decimal representation.

The decimal transcendental functions, as recommended decimal arithmetic operations, have been specified in the IEEE 754-2008 standard [8]. Therefore, virtually all the computing

systems that are compliant with the IEEE 754-2008 standard should provide a software or hardware solution for the decimal transcendental function computation. Recently, *Intel Cooperation* provides the first software solution to compute DFP and DXP transcendental functions using an existing and well-established BFP transcendental function mathematical library [41]. However, with the strict requirement on computational speed and accuracy in the future, the hardware components may be included in the high-end microprocessor to support the decimal transcendental computation.

The previous hardware implementations of the decimal transcendental function computation are reported in several patents [42, 43, 44, 45, 46] in which the decimal transcendental function computation is based on a binary arithmetic, rather than decimal. Therefore, they are not compliant with the DFP formats specified in the IEEE 754-2008 standard. In [40], a radix-10 BKM algorithm is presented for an efficient computation of DXP exponential and logarithm results. Unfortunately, this radix-10 BKM algorithm is not built according to the IEEE 754-2008 DFP standard. Recently, based on the recent development of basic decimal arithmetic units, more complex decimal transcendental arithmetic would be the next useful hardware components built for the future microprocessors. Therefore, the decimal transcendental arithmetic, as one of the hottest topic in decimal computer arithmetic, is attracting more and more researchers' attentions.

Some new algorithms and architectures, that are based on the IEEE 754-2008 standard, are developed for the computation of the DXP or DFP transcendental functions. In [47, 48, 49, 50, 51], the CORDIC-based architectures for a high performance decimal computation are described. The CORDIC-based methods have the potential to be modified for the computation of more transcendental functions in further development. However, since the CORDIC-based method needs to apply decimal multiplication and division operations, it has a large latency and hardware complexity. All of these considerations motivate us to design and implement the more efficient architectures for DFP or DXP units to satisfy the increasing potential demands of high-performance decimal transcendental computations.

1.3 Research Overview

A set of new algorithms and architectures for the DFP and DXP transcendental function computation based on different approaches are investigated in this dissertation. We mainly focus on the following DFP or DXP transcendental operations in this dissertation: the DFP and DXP based-10 logarithms (*log10*) and the antilogarithms (*exp10*), where v presents the DFP or DXP operands. The proposed algorithms and architectures of the based-10 logarithms and antilogarithms can be easily modified to compute the DXP natural logarithms (*log*) and the exponential (*exp*) operations. Moreover, the DXP reciprocal (*rootb*(v,-1)) and division (*div*(v_1, v_2)), which are considered as the most complex basic decimal arithmetic operation, are also investigated in this dissertation. This dissertation is structured in six parts with nine chapters shown as follows:

-Part I: **Preface** includes:

Chapter 1: *Introduction* presents the background, motivation, overview and contributions of this dissertation.

-Part II: **Research Background** includes:

Chapter 2: *DFP Transcendental Arithmetic* gives the overview about the DFP standard specified in IEEE 754-2008 and the common issues of DFP transcendental arithmetic design.

From this point, this dissertation is an original research, starting with:

-Part III: **Table-based First-Order Polynomial Approximation** includes:

Chapter 3: *A Novel Dynamic Non-Uniform Segmentation* presents a novel dynamic non-uniform segmentation method for the first-order polynomial elementary function approximation.

Chapter 4: *Efficient Decimal Logarithmic and Antilogarithmic Converters* present the new algorithms and architectures of DXP logarithmic and antilogarithmic converters based on the decimal first-order approximation polynomial method.

In Chapter 3, a novel dynamic non-uniform segmentation method is presented in detail, which can approximate the transcendental function by an optimized linear approximation with few non-uniform segments. Compared with previous non-uniform static method, the proposed method can significantly reduce the memory size occupied in hardware. Moreover, the proposed dynamic method can approximate the function to satisfy accuracy by the linear approximation in which the input, coefficients, and intermediate values are rounded to least bit-width, and this can not be achieved by the previous static non-uniform segmentation method.

In Chapter 4, we analyze the tradeoff of the hardware performance, and scale up the proposed architecture to achieve a higher precision of accuracy (obtain faithful results up to the precision of 10^{-7}). Moreover, a binary-based decimal linear approximation algorithm and its architecture are simulated as a benchmark to evaluate the performance of the proposed design. As far as we know, this work is the first attempt to study the decimal logarithmic and antilogarithmic converters based on the decimal piecewise first-order polynomial approximation method.

-Part IV: **Digit-Recurrence with Selection by Rounding** includes:

Chapter 5: *Improved Design of Decimal Floating-Point Logarithmic Converter* presents the algorithm and architecture of the DFP logarithmic converter, based on digit-recurrence algorithm with selection by rounding.

Chapter 6: *Improved Design of Decimal Floating-Point Antilogarithmic Converter* presents the algorithm and architecture of DFP antilogarithmic converter, based on digit-recurrence algorithm with selection by rounding.

In Chapter 5 and Chapter 6, the proposed algorithms and architectures can compute faithful DFP logarithm and antilogarithm results for any one of the three DFP formats, which are specified in the IEEE 754-2008 standard. These two designs make the first attempt to comprehensively analyze and implement the DFP logarithmic and antilogarithmic converters based on digit-recurrence algorithm with selection by rounding. The rough delay estimation results of the proposed architectures indicate that the latencies are close to or shorter than that of the binary radix-16 logarithmic and exponential converters, and that they have a

significant decrease in terms of the latency in contrast with our original published designs, the recent decimal CORDIC design, and the software implementation.
-Part V: **Function Iteration Method** includes:

Chapter 7: *A Decimal Reciprocal Unit Using Efficient Table Look-up* presents the efficient design and implementation of a 16-digit DXP decimal reciprocal unit based on Newton-Raphson iteration method.
Chapter 8: *Design and Implementation of A Radix-100 Decimal Division* presents a new algorithm and architecture of the 16-digit DXP radix-100 decimal divider based on the decimal non-restoring algorithm with pre-scaling method.

In Chapter 7, we analyze the computation error for the look-up tables with different sizes, in order to find the smallest size of look-up table for the efficient hardware implementation. The proposed design can utilize a half size of the look-up table as that used in the previous design to compute a faithful reciprocal result.

In Chapter 8, we design and implement a new radix-100 divider, which can can reduce the clock cycle to 1/4 of the previous radix-10 design, while maintaining accuracy of the result. As far as we know, this is the first work to research the algorithm and architecture of the radix-100 division. In addition, we would appreciate Mr. Yu Zhang's help for this work.
-Part VI: **Conclusion** includes:

Chapter 9: *Summary and Future Research* includes a summary of research, and thoughts on related future research.

1.4 Research Contributions

In this dissertation, we have researched and developed several new decimal algorithms and architectures for the DFP transcendental function computation. Some of them are the first published designs which can achieve faithful logarithm or antilogarithm results of DFP or DXP operands, specified in the IEEE 754-2008 standard. The algorithms and hardware designs presented in this dissertation provide a basis for the future hardware-oriented DFP or DXP transcendental function computation research. To help researchers evaluate the

performance of the proposed DFP transcendental arithmetic, we analyze the hardware implementation results of these arithmetic units, and compare the hardware performances of the proposed architectures with the binary based architectures, the recent decimal CORDIC designs, and the Intel's DFP software library. Our research on DFP transcendental arithmetic serves as a useful starting point for researchers who are interested in this area. In the future, as the performance gap between BFP and DFP arithmetic becomes smaller, DFP arithmetic units may replace or co-exist with BFP arithmetic units in the microprocessor. This dissertation may also be useful in guiding the design of future hardware and instruction set extensions for the computation of the decimal transcendental function in the microprocessor.

List of Publications:

We show next a list of the publications, arranged according to their correspondence to the different chapters of this dissertation.

Chapter 4: *Efficient Decimal Logarithmic and Antilogarithmic Converters*

- **D. Chen** and S. Ko, et al "A novel decimal-to-decimal logarithmic converter," *Proc. IEEE Symp. on Circuit and System (ISCAS'08)*, Seattle, Washington, pp. 688-691, May 2008.
- **D. Chen** and S. Ko, et al "A decimal-to-decimal antilogarithmic converter", *Proc. IEEE Canadian Electrical and Computer Engineering (CCECE'08)*, Niagara Falls, Ontario, Canada, pp. 1223-1226, May 2008.

Chapter 5: *Improved Design of Decimal Floating-Point Logarithmic Converter*

- **D. Chen**, Y. Zhang and S. Ko, et al "A 32-bit decimal floating-point logarithmic converter," *Proc. 19th IEEE Symp. on Computer Arithmetic (ARITH'19)*, Portland, Oregon, pp. 195-203, June 2009.
- **D. Chen** and S. Ko, et al "Improved decimal floating-point logarithmic converter based on selection by rounding", accepted for publication, *IEEE Trans. on Computers*, to appear, 2011.

Chapter 6: *Improved Design of Decimal Floating-Point Antilogarithmic Converter*

- **D. Chen** and S. Ko, et al "A new decimal antilogarithmic converter," *Proc. IEEE Symp. on Circuit and System (ISCAS'09)*, Taipei, Taiwan, pp. 688-691, May 2009.

Chapter 7: *A Decimal Reciprocal Unit Using Efficient Table Look-up*

- **D. Chen** and S. Ko, "Design and implementation of decimal reciprocal unit", *Proc. IEEE Canadian Electrical and Computer Engineering (CCECE'07)*, Vancouver, British Columbia, Canada, pp. 1094-1097, Apr. 2007.

Chapter 8: *Design and Implementation of A Radix-100 Decimal Division*

- Y. Zhang, **D. Chen** and S. Ko, et al "Design and implementation of a radix-100 decimal division", *International Workshop on Multimedia Signal Processing and Transmission*, Seoul, Korea, pp. 123-126, Sep. 2009.

The following papers are also published, but not considered to be the part of this dissertation.

- **D. Chen**, B. Zhou, Z. Guo and P. Nilsson, "Design and implementation of reciprocal unit", *48th Midwest Symp. on Circuits and Systems (MWSCAS'05)*, Cincinnati, Ohio, pp. 1318-1321, Aug. 2005.
- A. Malik, **D. Chen** and S. Ko, et al "A study on the design trade-off analysis of floating-point adders in FPGAs," *Canadian J. Electrical and Computer Engineering*, Vol. 33, No. 3/4, pp. 169-175, Summer/Fall 2008.
- K. Muma, **D. Chen** and S. Ko, et al "Combining ESOP minimization with BDD-based decomposition for improved FPGA synthesis," *Canadian J. Electrical and Computer Engineering*, Vol. 33, No. 3/4, pp 177-182, Summer/Fall 2008.
- Y. Zhang, **D. Chen** and S. Ko, et al "A high performance pseudo-multi-core ECC processor over GF(2^{163})", *Proc. IEEE Symp. on Circuit and System (ISCAS'10)*, pp 701-704, May 2010.
- Y. Zhang, **D. Chen** and S. Ko, et al "A high performance ECC hardware implementation with instruction-level parallelism over GF(2^{163})", *Elsevier J. Microprocessors and Microsystems*, vol. 34, pp. 228-236, Apr. 2010.

Part II

Research Background

Chapter 2

Decimal Transcendental Arithmetic

This chapter provides background information and discusses related fundamental concepts about the DFP and DXP transcendental function computation described in this dissertation. Section 2.1 gives an overview of DFP standard specified in IEEE 754-2008, which includes DFP formats and encoding, DFP arithmetic operations, DFP rounding modes, DFP special values and exception handling. Section 2.2 presents the common issues of the decimal transcendental arithmetic design, which include 1) the computation of DFP transcendental operations; 2) different types of hardware methods and their main features; and 3) the considerations of hardware implementation of DFP transcendental arithmetic.

2.1 DFP Formats in IEEE 754-2008 Standard

The IEEE 754 standard for floating-point arithmetic is the most widely-used standard for floating-point computation, and is implemented for many microprocessor designs and software mathematical libraries. The current version, IEEE 754-2008 [8], is the revision to the original IEEE 754-1985 [52] standard for BFP arithmetic and IEEE 854-1987 [53] for radix-independent floating-point arithmetic. One of the most important revisions to IEEE 754-1985 is the introduction of DFP formats and operations, which includes:

- **DFP formats**: which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special not-a-number (NaN).
- **DFP arithmetic operations**: which include basic DFP arithmetic operations, two decimal-specific operations, different types of conversions and some recommended DFP operations (transcendental operation).

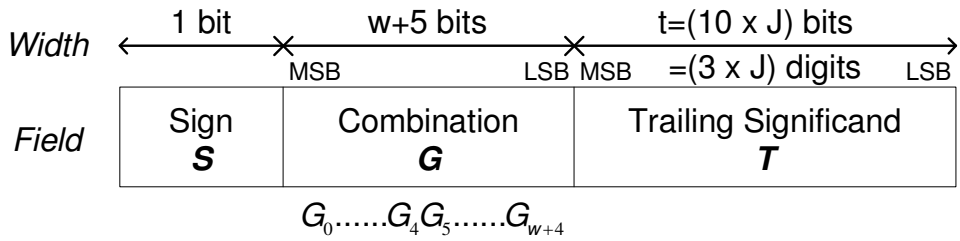

Figure 2.1: DFP interchange format with DPD encoding.

- **DFP rounding modes**: which include five rounding modes (roundTiesToEven (RNE), roundTiesToAway (RNA), roundTiesToPositive (RPI), roundTiesToNegative (RMI) and roundTowardZero (RNZ)) to guarantee the exactly or faithful rounded results for the inexact computation and conversion.
- **DFP exceptions handling**: which include the invalid operation, division by zero, overflow, underflow and inexact.

2.1.1 DFP Formats and Encodings

The encodings for DFP operands allow for a range of positive and negative values together with values of ± 0, $\pm\infty$, and not-a-number (NaN). Three interchange DFP formats are specified in the IEEE 754-2008 standard, which includes a storage format (Decimal32) and two basic computational formats (Decimal64 and Decimal128) as follows:

- **Storage format** is an interchange format not required by DFP arithmetic. The standard defines one decimal storage floating-point format encoded in 32-bit (Decimal32).
- **Basic format** is an interchange format available for DFP arithmetic. The standard defines two basic decimal computational floating-point format encoded in 64-bit (Decimal64) and 128-bit (Decimal128) respectively.

Figure 2.1 shows the basic DFP interchange format specified in the IEEE 754-2008 standard, which includes:

- **1-bit Sign Field:** 1-bit field S, which indicates the sign of the number in the same way as BFP numbers.
- w+5**-bit Combination Field:** w+5-bit field G, and can be considered as two subfields: first, the five most significant bits (MSBs) of the *Combination Field*, $G_0...G_4$, is defined

Table 2.1: Decoding of the combination field.

$G_0G_1G_2G_3G_4$	**Type**	**Exponent MSBs (2-bit)**	**Coefficient MSD (4-bit)**
a b c d e	$d_0 < 8$	a b	0 c d e
1 1 c d e	$d_0 > 7$	c d	1 0 0 e
1 1 1 1 1	Infinity	- -	- - - -
1 1 1 1 0	NaN	- -	- - - -

as one subfield, which 1) encodes two MSBs of the nonnegative biased exponent and the most significant digit (MSD) of the decimal significand, d_0, 2) indicates the Not-a-Number (NaN) and infinite number ($\pm\infty$); second, the remaining w-bit of *Combination Field*, $G_5...G_{w+4}$, is defined as a suffix to the two MSBs derived from $G_0...G_4$, which consists of $w+2$-bit nonnegative biased exponent. The whole encoded exponent is an unsigned binary integer with the largest unsigned value. The value of the exponent is calculated by subtracting an *Exponent Bias* from the value of the encoded exponent, in order to represent both negative and positive exponents. Table 2.1 shows the detail of the encoding for the 5-bit MSBs of the *Combination Field*, $G_0...G_4$, where a, b, c, d, e are used to represent the value (0 or 1) of $G_0...G_4$.

- $J\times10$**-bit Trailing Significand Field:** $J\times10$-bit field T can be specified by Densely Packed Decimal (DPD) encoding [54] or Binary Integer Decimal (BID) encoding [55] in IEEE 754-2008. In this dissertation, the DPD encoding is chosen to represent *Trailing Significand Field*, as a suffix to the MSD derived from *Combination Field* to construct q-digit decimal significand, ($q = 3 \times J + 1$). *Trailing Significand Field* ($J\times10$-bit) is a multiple of 10-bit, and the most significant group is on the left. Every 10-bit group represents three decimal digits, using DPD encoding and can be decoded to a 12-bit binary-coded decimal (BCD) representation. The DPD encoding has the advantage of straightforward decimal rounding and shifting, so it is usually used in the hardware implementations of decimal arithmetic units. The BID encoding has the advantage of using the current high-speed binary integer arithmetic logic unit (ALU) in the microprocessor, so it is usually used in the software implementations [55] for decimal computations. For more fundamental concepts of the BID encoding, refer

Table 2.2: Parameters in DFP interchange formats.

Format Name	**Decimal32**	**Decimal64**	**Decimal128**
Storage Width (bit)	32	64	128
Trailing significand field ($J\times$10-bit)	20	50	110
Combination Field (w+5-bit)	11	13	17
Decimal Significand (q-digit)	7	16	34
Exponent bias	101	398	6176
emax	+96	+384	+6144
emin	−95	−383	−6143

to [8]. Table 2.2 provides important parameters used in the standard for different DFP interchange formats. In this dissertation, the DFP format in the DPD encoding is selected for all research works so that the decimal significand of a DFP operand can be decoded to binary-coded decimal (BCD) representation in hardware.

In the IEEE 754-2008 standard, the value of the decimal significand is a non-normalized unsigned decimal fraction in the form of $d_0.d_1d_2...d_{q-1}$, $0 \leq d_i < 10$. In decimal computer arithmetic, the decimal significand is usually represented as an integer. The value of a DFP number is represented as follows:

$$v = (-1)^S \times 10^e \times \mathit{significand} \tag{2.1}$$

In (2.1), S is the sign of the DFP number; the value of the real exponent e is in the range of $(\mathit{emin}-q+1) \leq e \leq (\mathit{emax}-q+1)$; and the decimal significand, *significand*, is represented as a non-normalized decimal integer. Since the decimal significand is non-normalized, DFP number may have multiple representations, which is called DFP number's **cohort**. For example [8], if *significand* is a multiple of 100 and e is less than its maximum allowed value, then (S, e, *significand*) and (S, $e+2$, *significand*/100) are two representations for the same DFP number which are members of the same **cohort**.

Based on an example shown in [56], we illustrate how the decimal number $v = -8.35 = -835\times10^{-2}$ is encoded in the Decimal64 format using the DPD encoding.

1. The decimal significand can be represented using BCD encoding, $significand$=00000000 $00000835_{radix\text{-}10}$, except the most significand digit (MSD), the remaining fifteen decimal digits are represented using the 50 bits DPD encoding (T=000...001000111101).
2. The $w+5$-bit *combination field* G = 0100010001100 where the two MSBs and eight least significand bits (LSBs) of G are from the biased exponent e, and the middle three bits of G are from the MSD of the significand.
3. The sign bit $S=1$ represents the negative DPF number, thus, the representation of DFP number $v=-835\times10^{-2}$ in the DPD format is: $10100010001100000...001000111101_{radix-2}$.

The interpretation of the combination of the various fields in a DFP **special value** is as follows:

- **Not-a-Number (NaN):** If G_0 through G_4 are 11111 (refer to Table 2.1), then v is NaN regardless of S. Furthermore, if G_5 is 1, then v is a signaling NaN ($sNaN$); otherwise, v is a quiet NaN ($qNaN$). The remaining bits of G are ignored, and T constitutes the NaN's payload, which can be used to distinguish various NaNs.
- **Infinite number:** If G_0 through G_4 are 11110, then v represents $+\infty$ or $-\infty$, according to the sign bit. The values of the remaining bits in G, and T, are ignored.
- **Overflow:** If DFP numbers with absolute values are larger than the largest DFP number ($|v_{\max}|=(10^q-1)\times10^{emax-q+1}$) then overflow occurs.
- **Underflow and Subnormal:** If DFP number is less than the smallest DFP number ($|v_{\min}|=10^{emin-q+1}$) then underflow occurs. If the absolute value of DFP number is less than 10^{emin} and larger than $10^{emin-q+1}$, it produces subnormal.
- **Normal number:** The remaining exponent values and significands represent normal numbers.

2.1.2 DFP Arithmetic Operations

The hardware implementations of DFP arithmetic units, which are compliant with the IEEE 754-2008 standard, must provide support for at least one basic computational format (Decimal64 and/or Decimal128). For each of specified DFP operation, first, DFP operands with the external format (DPD or BID) are converted to the internal format (BCD or binary),

accompanied with sign bit, unsigned binary exponent and exception flag; second, the DFP operation is computed to produce an intermediate result correct to infinite precision without rounding; third, the intermediate result is rounded to a target finite digit-width of decimal significand; fourth, if necessary, the rounded results are packaged back to the destination's DFP format. The decimal arithmetic operations specified in IEEE 754-2008 are mainly classified as follows:

- **Basic DFP arithmetic operations:** include DFP addition, subtraction, multiplication, division, square-root and fused multiply addition, which are usually implemented in most today's microprocessors. The standard recommends to provide exactly rounded DFP computation results for these basic DFP arithmetic operations.
- **Two decimal-specific DFP operations:** SameQuantum(v_1,v_2) compares the exponents of v_1 and v_2 and the output true if they are the same and false if they are different. Quantize(v_1,v_2) generates a DFP number that has the same value as v_1 and the same exponent as v_2, unless rounding or an exception occurs.
- **DFP comparison operations:** Comparison(v_1,v_2) compares the numerical values of v_1 and v_2. The comparison operations do not differentiate the redundant representations of the same number.
- **DFP conversion operations:** Conversion(v) supports the conversions among the decimal integer, BFP and DFP formats. The conversions between DFP and BFP must be exactly rounded.
- **Recommended DFP operations:** decimal transcendental arithmetic operations are defined as recommended operations, such as logarithms, exponentials, trigonometric functions as so on. The standard recommends to provide exactly rounded DFP computation results for these transcendental functions.

Since a DFP number might have multiple representations (DFP number's cohort), the value of a DFP result is not only determined by the operation and the DFP operands' values, but also depends on the selection of the proper representation of DFP numbers. Because of this characteristic, the standard defines the preferred representation exponent, which refers to a required exponent (quantum). The selection of a particular representation for a DFP

result is dependent on whether the representation of operation result is exact or inexact:

- **Exact DFP operation result:** If the DFP arithmetic operation result is exact, the **cohort** member is selected based on the preferred exponent (quantum) for a DFP result of that operation. The preferred exponents have been specified in the standard for different operations. For DFP addition, if the result is exact, the preferred exponent is the minimum quantum of the operands for a DFP addition, that is $e_R = \min(e_{v_1}, e_{v_2})$.
- **Inexact DFP operation result:** If the DFP arithmetic operation (except for quantize operation) result is inexact, the cohort member of the least possible exponent is used to get the maximum number of significant digits. For DFP addition, if the result is inexact, the preferred exponent may be decreased to keep the MSD of decimal significand not zero.

2.1.3 DFP Rounding Modes

Rounding is often done on purpose to obtain a finite number that can exactly represent the closest value to the exact infinite result. In general, since the result of floating-point operation is not a finite number, the inexact result must be converted to a close representable floating-point number in a given finite precision format, which is referred as rounding. The IEEE 754-2008 standard specifies five types of *active rounding modes* for the DFP arithmetic:

- **roundTiesToEven:** rounds the result to the nearest representable DFP number. The number with an even least significant digit (LSD) should be selected, if a tie occurs.
- **roundTiesToAway:** rounds the result to the nearest representable DFP number. The number with a larger magnitude should be selected, if a tie occurs.
- **roundTiesToPositive:** rounds the result toward positive infinity. The closest DFP number, which is greater than the exact result, should be selected.
- **roundTiesToNegative:** rounds the result toward negative infinity. The closest DFP number, which is lower than the exact result, should be selected.
- **roundTowardZero:** truncates the result. The closest DFP number, which is lower in magnitude than the exact result, should be selected.

2.1.4 Exception Handling

The exceptions happen when the result of an operation is not the expected floating-point number. In this case, the default non-stop exception handling delivers a default result, and raises the corresponding status flag for exceptions. The IEEE 754-2008 standard specifics five kinds of exceptions, shown as follows:

- **Invalid operation:** is signaled when there is no usefully definable DFP result. The invalid operation usually happens if the operand is invalid, such as the NaN or infinite operand, or if the operation is invalid, such as quare-root of negative operands. In this case, the default result is a *qNaN*.
- **Division by zero:** is signaled only if an operation with finite operands produces an exact infinite result. For example, the dividend is a finite non-zero operand and the divisor is zero. The default result may be plus or minus infinity.
- **Overflow:** is signaled if the magnitude of a result exceeds the largest finite representable number in the format of the operation. The default result may be plus or minus infinity, or plus or minus the largest representable number in the format, depending on the rounding mode.
- **Underflow:** is signaled if the magnitude of a result exceeds the smallest finite representable number in the format of the operation. This is detected before rounding examining the precision digits and the exponent range. The default result may be zero, a subnormal number or a smallest finite number representable in the format.
- **Inexact:** is signaled if the rounded result of an operation differs from the infinite precision result. The default result is the rounded or the overflowed result.

2.2 Decimal Transcendental Unit Design

In this section, some considerations about the design and implementation of DFP and DXP transcendental arithmetic units are presented. First, the description of DFP transcendental operations covered in this dissertation is described in terms of the exception handling, the range reduction, and the rounding. Second, different types of BFP hardware-oriented

Table 2.3: DFP arithmetic functions included in this dissertation.

Operations	Domain	Exceptions
log10	$[0, +\infty]$	invalid operation, divideByZero, inexact
exp10	$[-\infty, +\infty]$	invalid operation, overflow, underflow, inexact
rootn(v,-1)	$[-\infty, +\infty]$	invalid operation, divideByZero
div(v_1, v_2)		overflow, underflow, inexact

methods and their main features are presented based on the summary in [32, 57].

2.2.1 Some Details of DFP Transcendental Operations

In this dissertation, the decimal arithmetic functions include the DFP and DXP based-10 logarithm (*log10*) and the antilogarithm (*exp10*), the DXP reciprocal (*rootb*(v,-1)) and the DXP division (*div*(v_1, v_2)) as shown in Table 2.3, where v presents the DFP or DXP operands. More decimal transcendental functions (not covered in this dissertation) are specified in the IEEE 754-2008 standard as the recommended DFP operations [8].

Exception Handling

All the exceptions for the each specific transcendental function are shown in Table 2.3. Details about the exception handling of the based-10 logarithm and antilogarithm operations, can be referred to Section 5.2.1 and Section 6.2.1. However, since the DFP computation for reciprocal and division operations is not covered in this dissertation, more details about the exception handling of these two operations can be referred to the description in [8, 28].

Range Reduction

To compute a DFP transcendental function, $f(v)$, over a range $v \in [a, b]$ with a given target precision requirement, the range reduction leads to compute the transcendental function in a smaller interval, $[a', b']$, which can simplify the function approximation, reduce the size of tables, and achieve the faster speed for the hardware implementation [32]. Table 2.4 presents the reduced intervals we will use throughout this dissertation for approximating

Table 2.4: Reduced intervals for considered transcendental functions.

Functions	Domain	Reduced Interval
$\log_{10}(v)$	$[0, +\infty]$	$[0.1, 1)$
10^v	$[-\infty, +\infty]$	$(-1, 1)$
v^{-1} and v_1/v_2	$[-\infty, +\infty]$	$[0.1, 1)$

considered DFP transcendental functions. For the DFP based-10 logarithm operation, the non-normalized decimal significand of DFP operand, which is in the range of $[0, +\infty]$, should be adjusted into the range of $[0.1, 1)$ before it is computed. For the DFP antilogarithm operation, the non-uniform decimal significand of DFP operand, which is in the range of $[-\infty, +\infty]$, should be adjusted into the range of $(-1, 1)$ before it is computed. For DFP reciprocal and division operations, the non-uniform decimal significand of DFP operand, which is in the range of $[-\infty, +\infty]$, should be adjusted into the range of $[0.1, 1)$ before it is computed [28]. More details about the range reduction of DFP based-10 logarithm and antilogarithm operations are presented in Section 5.2.2 and Section 6.2.2.

Rounding

The arithmetic operation in systems must behave as if the results are first computed exactly with infinite precision, and then rounded. In terms of *exact rounding*, the operation result provided by an algorithm computing a certain operation is always the floating-point number which is closest to the exact result. Another frequently used notion is *faithful rounding*, which happens when the intermediate result is in the interval between the two closest floating-point numbers surround the exact result. The unit in the last place *ulp*, denotes the absolute value of the difference between the two numbers in a given finite numerical representation which are closest to a given number [58]. It is used as a measurement of precision in numeric calculations. If the floating-point arithmetic computation error from the exact result is less than 0.5 *ulp*, it means that the exactly rounded result in the round-to-nearest rounding mode is always provided. On the other hand, the faithful rounding means that the computation error from the exact result is less than one unit in the last place (1 *ulp*) [57].

For DFP basic arithmetic operations, such as addition, multiplication, division and square

root, defined in the IEEE 754-2008 standard must provide exactly rounded results. Although the DFP recommended (transcendental) operations specified in the standard are required to provide exactly rounded results, it has been believed for many years that the exact rounding of transcendental functions would be too much expensive in terms of either speed or hardware requirements [59]. To achieve exactly rounded results by any one of rounding modes, it is needed to determine whether the value of the exact result (infinite precision) is less or higher than the midpoint between the two nearest DFP numbers. However, when the exact result is so close to the midpoint, exact rounding is difficult to perform, except that we can determine the maximum length of chain of nine or zero after rounding digit for every possible DFP results (Table Maker's Dilemma [60]). A study of worst cases for exact rounding of the exponential function in the IEEE 754-2008 Decimal64 format is presented in [61], in which all the bad cases whose distance from a breakpoint are computed. In this dissertation, we concentrate on the delay optimization of the proposed DFP transcendental arithmetic, so that we design the algorithms and architecture to guarantee faithfully rounded results (within 1 *ulp* of precision) using the *roundTiesToEven* mode.

2.2.2 Classification of Hardware Approaches in BFP

Both software and hardware-oriented algorithms to compute transcendental functions, and issues related to accurate binary floating-point (BFP) implementations of these functions are presented in the Dr. J-M Muller's book [32]. Most BFP transcendental functions are firstly considered and implemented by the software-oriented methods [62, 63, 64], due to their advantage of using large look-up tables and of providing more accurate results. However, some applications which require high-speed solutions for the computation of these transcendental operations, such as computer 3D graphics, scientific computing, artificial neural networks, logarithmic number system (LNS), digital signal processing (DSP) [33, 34, 35, 36, 65, 37], have led to the development of the new dedicated hardware to be implemented. According to the summarization in the Dr. A. Pińeiro's Ph.D dissertation [57], the main types of approaches used for the transcendental function computation in hardware can be separated into two groups: non-iterative and iterative methods.

- **Non-iterative Method:** mainly includes direct table-lookup, polynomial and rational approximations [66, 67, 68, 69, 70, 71, 72], and table-based methods [73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87].
- **Iterative Method:** mainly includes digit-recurrence and on-line algorithms [88, 89, 90, 91, 92, 93, 94, 95], and functional iteration methods, such as Newton-Raphson and Goldschmidt algorithms [96, 97, 65, 98, 99, 100, 101].

The non-iterative methods are usually adopted to compute low-precision, up to single-precision BFP operations or 32-bit BXP computations. In particular, if the objective is to implement transcendental arithmetic units for high-speed and low-power applications, the units have some tolerance for computational errors. On the other hand, iterative methods can be applied to both low-precision and high-precision transcendental function computations, including double-precision or double-extended precision BFP operations, and 64-bit or even more bit-width BXP computations. Thus, the transcendental arithmetic units implemented based on iterative methods are mainly for high-precision scientific computations which are impracticable for non-iterative methods.

Non-iterative method

-Direct table-lookup method needs to store all the computation results in the memory for the corresponding input operands. Thus, the huge memory requirements of such technique make this method impracticable for the computation with the large precision of accuracy. For example, for a single-precision BFP transcendental computation, the required table-lookup size is $2^{24} \times 24$-bits, which is too large to be implemented in hardware [57].

-Polynomial and rational approximation method [66, 67, 68, 69, 70, 71, 72] mainly uses a finite number of additions and multiplications, which are available in the microprocessors, to approximate transcendental functions by combinations of these operations. If a fast enough division is also available in the microprocessor, rational approximations can also be applied for the computation to other transcendental functions, such as exponentials, logarithms, trigonometric functions, etc. [57]. To achieve a high precision of the accuracy for the computation, the degree of the polynomial is usually selected with a large number, which leads to a very complex and time-consuming hardware implementation.

-Table-based method includes the piecewise linear and quadratic approximations algorithm [82, 85, 86, 87] and the bipartite tables method [73, 74, 75, 76, 77, 78, 79, 80, 81, 83, 84]. The piecewise linear and quadratic approximations algorithm splits the interval where the function is to be linearly or quadratically approximated into several smaller subinterval so that it is sufficient to store the coefficients of a low-degree approximation for each subinterval in a table. While the bipartite table method approximates the transcendental function based on two parallel tables, in that case, two approximations are usually provided in a carry-save format, and then these two approximations are added in a carry propagate adder to produce the final approximation of the transcendental function [74]. Table-based method is a very popular method for the transcendental function computation with a low accuracy (currently up to 24-bit), since this kind of method allows the hardware implementation with significantly lower hardware cost than that of a straightforward table implementation, and at the same time keeping a faster speed than digit-recurrence algorithms, CORDIC algorithms or polynomial and rational approximations. Table-based methods can not only be applied perfectly in such applications as computer 3D graphics, artificial neural networks, and logarithm number systems, but also function in providing initial seed values to iterative methods, such as the Newton-Raphson algorithms for division and square root [98, 99].

Iterative Method

-Digit-recurrence method [88, 89, 90, 91, 92, 93, 94, 95] is also known as the digit-by-digit iterative method, which allows the use of a simple *shift-and-add* implementation in each iteration to achieve a fixed precision of accuracy of the result. As for the implementation cost, the digit-recurrence algorithm has its advantage in achieving high precision results in contract to the table-based and functional iteration method. In terms of execution time, since the digit-recurrence algorithm can only achieve radix-r digit in each iteration (linear convergence), it takes more execution time to achieve a certain precision of the accuracy compared with other methods. To develop and implement the efficient arithmetic unit based on this method, the radix $r = 2^b$ applied in this algorithm must be selected carefully [57]. Increasing the value of b can reduce the latency of algorithm, but it costs more implementations, leads to a slower cycle time, and increases the complexity of the selection function for the digits,

which is usually the most time-consuming part in the architecture.

-Functional iteration method includes the Newton-Raphson [88, 96, 97, 65, 98, 99] and the Goldschmidt algorithms [100, 101]. This type of method is iteration-based method based on the multiplication operation, which makes this method more advantageous to utilize the multiplier built in microprocessors. Unlike the digit-recurrence algorithm (linear convergence), this type of method can achieve double precision of results in each iteration (quadratic convergence). To achieve a certain precision of accuracy, the function iteration method usually adopts the table-based method to generate an initial approximation (seed value) at the very beginning, and then achieves the final accuracy by couple of iterations. Thus, the number of iterations of this type of method depends on the precision of the accuracy of the initial approximation. For example, with the precision of the initial approximation as 2^{-8}, the function iteration method requires three iterations to achieve a double precision of accuracy, 2^{-53} ($2^{-8} \rightarrow 2^{-16} \rightarrow 2^{-32} \rightarrow 2^{-64}$). Since the function iteration method can not directly obtain the remainder of the operation, the straightforward rounding is difficult to be achieved.

2.2.3 Considerations of Hardware Implementation

To design and implement the proposed decimal transcendental arithmetic in hardware, the straightforward idea is to take the well-established hardware-oriented algorithms, adopt the techniques used in BFP or BXP transcendental functions, and then transfer them with the necessary changes [41]. However, processing this idea would be major challenges that can not be overcome and implemented quickly because that all aspects between the decimal and binary transcendental computation are different in detail. In this section, some considerations in terms of hardware implementation of the proposed decimal transcendental arithmetic are listed as follows:

10's Complement Number

In the computer arithmetic, the complement representation method is a technique which allows subtracting one number from another using only addition of positive numbers. This

method has been commonly used in modern computer arithmetic, such as the 2's complement number. In this dissertation, all the intermediate variables in the hardware implementation are represented with 10's complement number in the BCD encoding. The reason for choosing 10's complement format is that the decimal subtraction operation can be replaced by a decimal addition in 10's complement format, and all decimal digits, including the sign digit, can be operated in the decimal addition or subtraction.

In mathematics, the 10's complement of a n-digit decimal number X can be obtained as $10^n - X$ [102]. In the hardware implementation, the 10's complement of a number is usually obtained by adding '1' to its 9's complement [28]. The 9's complement of a number X is obtained by subtracting each digit in X from 9, which is usually implemented by a very small look-up table. In the data path of the architecture, adding '1' to obtain the 10's complement can be done separately with the 9's complement conversion, which is mostly often added with the carry in the least significand digit (LSD) of the decimal addition.

Non-redundant and Redundant Number System

The number systems of the computer arithmetic design can be referred to the non-redundant and redundant number systems. In the non-redundant decimal number system, every decimal number has unique representation in the conventional *8421* BCD encoding; in other words, no two sequences represent the same numerical value. For example, a non-redundant representation of an n-digit unsigned integer X is given in decimal by a digit-vector: $X = (x_{n-1}, ...x_1, x_0)$, where $\lceil n = \log_{10}(X) \rceil$, and $X = \sum_{i=0}^{n-1} x_i 10^i$, with $x_i \in \{0, 1...8, 9\}$ for $0 \leq i \leq n-1$. The redundant decimal number system has multiple valid representation for the same numerical value, such as decimal carry save number system, decimal *5211* and *4221* signed-digit number system, and decimal *septa* signed-digit number system, which have been widely used in current basic decimal computer arithmetic designs [17, 18, 22, 23, 24, 27, 29]. For example, a redundant representation of X in carry-save form is given by two digit-vectors, (XS, XC), with $XS = (xs_{n-1}, ...xs_1, xs_0)$ and $XC = (xc_{n-1}, ...xc_1, xc_0)$, such that $X = (XS + XC) \bmod 10^n$, where the + symbol represents the decimal addition.

In both decimal and binary arithmetic, partial products in multipliers and partial remainders in dividers or transcendental arithmetic units are usually represented via a redundant

number system (e.g., binary signed-digit [94, 92], decimal carry-save [23, 29], and decimal signed-digit [25, 27, 18, 24]). The number of redundant digits is sufficiently more than that of the radix, which allows the carry-free addition and subtraction as the basic sub-operations implemented in the data-path. In this dissertation, the redundant carry-save representation presented in Chapter 5 and Chapter 6 is used to improve the hardware performance of the proposed DFP transcendental arithmetic in terms of the computation latency. It makes the addition time independent of the operand's precision in the proposed architecture. When performing accumulation of values or computation of partial remainders in the proposed architecture, it is desirable to keep the intermediate sum and carry in (XS, XC) carry-save form, and only convert the final result to the non-redundant decimal 10's complement number system at the end.

2.2.4 Related Basic Decimal Arithmetic

The computation of decimal transcendental arithmetic operations is based on the combination of the basic decimal computer arithmetic operations, such as shift operation, addition, substraction, multiplication etc. In order to develop and implement the architecture of the decimal transcendental arithmetic, we need to implement some decimal basic computer arithmetic (as the subcomponents of the proposed architecture) by our own work based on the basic binary arithmetic in the textbook [103, 58], as well as other recently published designs, such as the decimal adder [104, 105, 20, 16], the decimal multiplier [23, 20], the decimal multiple logic [23, 24], the decimal multiplier, and the BCD-to-binary and binary-to BCD converters [103] etc. The detailed information about these basic decimal arithmetic is presented in the architecture section of each chapter.

Part III

Table-based First-Order Polynomial Approximation

Chapter 3

A Dynamic Non-Uniform Segmentation Method

This chapter presents a dynamic non-uniform segmentation method for the first-order polynomial transcendental function approximation. The proposed method can approximate the transcendental function by the optimized linear approximation with very few non-uniform segments. While the previous method is based on a static bit-width analysis, the proposed method is mainly based on the dynamic bit-width analysis and capable of reducing the number of segments, which in turn can significantly reduce the memory size occupied in hardware. The proposed dynamic method can approximate the function to satisfy accuracy requirement by employing a process of linear approximation, in which the input, coefficients, and intermediate values are rounded to the least bit-width that can not be achieved by previous static non-uniform segmentation methods.

3.1 Introduction

Transcendental functions, such as logarithm, exponential, trigonometric, square root etc, and combinations of these functions (compound functions) are essential in digital signal processing, computer 3D graphics, scientific computing and so on. Computing the transcendental functions effectively and accurately has been one of the major goals in computer arithmetic. The piecewise polynomial approximation algorithm is an attractive method because all transcendental or compound functions can be evaluated by a set of simple linear or quadratic approximations. Based on the polynomial approximation algorithm, the function evaluation in hardware [74, 75, 106, 107, 108, 109, 110, 111] has been employed on a field-programmable gate array (FPGA).

To achieve a required accuracy, the interval of the function can be split into a set of

segments with a same size. Such an approach is called uniform segmentation method [74, 75, 106, 107, 108, 109]. However, the shortcoming of this approach is that numerous segments make the size of the look-up table become too large to be actually practical. By way of contrast, a more effective approach is to determine the segment that has the largest size while maintaining the specified approximation accuracy. Such an approach in which segments have different width is called a non-uniform segmentation method [110, 111].

Main challenges for designing a polynomial structure based on the non-uniform segmentation method stem from the following three aspects. The first challenge is to determine the minimum number of the bit-width for internal signals in the fixed-point data path. The most commonly used approach for bit-width optimization is a dynamic method in which the bit-width of each signal is gradually adjusted to a point where all inputs meet the precision requirement [34, 37, 112, 113]. In [114], a static bit-width optimization approach (MiniBit) is proposed, which is adopted in the static non-uniform segmentation methods [110, 111] to compute the bit-width for each signal in mathematical manner. The second challenge is to select the best scheme to partition the interval of the function to the fewest number of segments, which can produce a minimum look-up table size for storing the coefficients and a relative simple segment index encoder (SIE). In [110], the domain of the function is partitioned at the point where the maximum absolute error occurs. A two-level hierarchical partition scheme is adopted in [111]. The third challenge is to compute the best-fit linear approximation with finite precision internal signals in each segment so that a faithful rounding can be guaranteed for accuracy.

In this chapter, we propose a new dynamic non-uniform segmentation method for linear approximation function evaluation. The main advantages of this method are:

- it can optimize the uniform fractional bit-width (UFB) determined by MiniBit to less one by a dynamic bit-width analysis;
- it can limit the number of non-uniform segments to minimum by a binary search partition scheme (BSPS) [115];
- it can compute the best-fit optimized linear approximation in which internal signals are rounded to finite precision in each segment.

This chapter is organized as follows: Section 3.2 presents the notations and analyzes the minimum maximum (minimax) linear approximation in one segment. In Section 3.3, we present the proposed dynamic non-uniform segmentation method for linear approximation function evaluation. Section 3.4 shows a hardware architecture for function evaluation. In Section 3.5, we present experimental results, which are then compared with the results obtained from the previous static method. Section 3.6 gives summary. While the proposed dynamic non-uniform segmentation method is evaluated by approximating binary transcendental functions in this chapter, it is then modified and applied to the designs and implementations of decimal logarithmic and antilogarithmic converters based on the piecewise linear approximation algorithms in the following Chapter 4.

3.2 Minimax Polynomial Approximation

3.2.1 Notations

In this chapter, we deal with the linear approximation evaluation of the function $f(x)$ with its input and output in the BXP format. The input value of x is a m-bit BXP number in the domain $[a, b]$; the function evaluation result is a n-bit BXP number. To achieve a specified accuracy, the interval of x is typically split into a set of subintervals, $[a_i, b_i]$, where $a \leq a_i < b_i \leq b$, and i is the segment index. According to [32], in each subinterval, there are many straight lines, defined as P_1 that can evaluate the function $f(x)$, and of which only $p^*(x)$ is the best-fit linear approximation, $c_{0i}x+c_{1i}$, for achieving the minimax absolute error:

$$\|f(x)-p^*(x)\|_\infty = \min_{p(x)\in \mathrm{P}_1} \max_{a_i \leq x \leq b_i} |f(x) - p(x)| \tag{3.1}$$

With the piecewise linear approximation, errors are produced in three ways. The first one is the maximum linear approximation error, ε_a, resulted from the difference of the function $f(x)$ and its minimax linear approximation:

$$\varepsilon_a = \max_{a_i \leq x < b_i} |f(x) - (c_{0i} \times x + c_{1i})| \tag{3.2}$$

The second one is the quantization error, ε_q, as shown in (3.3), produced by the finite precision of rounded inputs, $x' = round(x)$, coefficients, $c'_{0i} = round(c_{0i})$ and $c'_{1i} = round(c_{1i})$,

and intermediate values, $D'_i = round(c'_{0i} \times x')$, in the hardware implementation. Note that in this chapter, x, c_{0i}, c_{1i} and $D_i = c_{0i} \times x$ represent infinite precision inputs, coefficients and intermediate values respectively, while x', c'_{0i}, c'_{1i} and D'_i represent rounded inputs, coefficients and intermediate values respectively.

$$\varepsilon_q = (c_{0i} \times x + c_{1i}) - (D'_i + c'_{1i}) \tag{3.3}$$

The third one is the final output rounding error, ε_r, whose maximum value is 0.5 unit in the last place (*ulp*). In order to obtain a n-bit accuracy, the following condition must be satisfied:

$$\varepsilon_t = \varepsilon_a + \varepsilon_q + \varepsilon_r \leq 2^{-n} \tag{3.4}$$

3.2.2 Minimax Error Analysis in One Segment

In each segment, the best-fit straight line can be found by Chebyshev theorem [32] which gives a characterization of the minimax approximations to a function.

Chebyshev Theorem: p^* is the minimax degree-n approximation to f on $[a_i, b_i]$, if and only if there are at least $n+2$ values, $a_i \leq x_0 < x_1 < ... < x_n < x_{n+1} \leq b_i$, such that:

$$p^*(x_i) - f(x_i) = (-1)^i [p^*(x_0) - f(x_0)] = \pm \|f - p^*\|_\infty \tag{3.5}$$

In this section, we analyze the evaluation of $f(x)$ in one segment, $[a_i, b_i]$, by its minimax linear approximation. Based on Chebyshev theorem, there are at least three values, x_0, x_1 and x_2, where the minimax approximation error, ε_a, is kept balanceable and reached with alternate signs. The convexity of the function $f(x)$ implies that the differences between $f(x)$ and $p^*(x)$ at the *starting* ($x_0 = a_i$), *ending* ($x_2 = b_i$) and *tangent* ($f'(x_1) = c_{0i}$) points are equal, and represent the minimax error. Thus, we obtain:

$$\begin{cases} f(a_i) - (c_{0i} \times a_i + c_{1i}) &= -\varepsilon_a \\ f(x_1) - (c_{0i} \times x_1 + c_{1i}) &= \varepsilon_a \\ f(b_i) - (c_{0i} \times b_i + c_{1i}) &= -\varepsilon_a \\ f(x_1)' - c_{0i} &= 0 \end{cases} \tag{3.6}$$

According to (3.6), the coefficients c_{0i} and c_{1i}, the value x_1 and the minimax error ε_a are computed so that the best-fit minimax linear approximation in $[a_i, b_i]$ is determined. Since,

Algorithm 1 Determination of Coefficients c'_{0i} and c''_{1i} in One Segment

Inputs: 1) Function, $f(x)$; 2) one segment, $[a_i, b_i]$; 3) precision of c'_{0i} in bits, q; 4) precision of c'_{1i} and $D'_i(x')$ in bits, p; 5) precision of x in $f(x)$, m.

Outputs: 1) Best-fit coefficients c'_{0i} and c''_{1i} ; 2) Minmax error, $|\varepsilon'_a|$

1: $[c_{0i}, c_{1i}] \leftarrow \text{Chebyshev}(f(x), [a_i, b_i])$
2: $c'_{0i} \leftarrow \text{round}(c_{0i}, q)$
3: $x' \leftarrow \text{round}(x, p)$
4: $c'_{1i} \leftarrow \text{round}(c_{1i}, p)$
5: $D'_i(x') = \text{round}(c'_{0i} \times x', p)$
6: $max \leftarrow \max\limits_{\{x_i=a_i \text{ to } b_i \text{ steps } 2^{-m}\}} (f(x_i) - D'_i(\text{round}(x_i, q)))$
7: $min \leftarrow \min\limits_{\{x_i=a_i \text{ to } b_i \text{ steps } 2^{-m}\}} (f(x_i) - D'_i(\text{round}(x_i, q)))$
8: $c''_{1i} \leftarrow \dfrac{max + min}{2}$
9: $|\varepsilon'_a| \leftarrow \dfrac{max - min}{2}$

the infinite precision input, coefficients and intermediate values have to be rounded to the finite precision x', c'_{0i}, c'_{1i} and D'_i in hardware. As a result, a quantization error, ε_q, is produced, and the best-fit linear approximation line obtained by Chebyshev theorem is moved to $p^*(x')$ as shown in (3.7), which is not a minimax linear approximation anymore, and the minimax approximation errors are not balanceable.

$$\begin{aligned} D_i &= c'_{0i} \times x' \\ p^*(x') &= D'_i + c'_{1i} \end{aligned} \tag{3.7}$$

To redetermine a new best-fit linear approximation, $p^*_r(x')$, with these rounded values of x', c'_{0i}, c'_{1i} and D'_i, we keep the value of c'_{0i} and adjust the value of c'_{1i} to c''_{1i} according to (3.8):

$$c''_{1i} = \frac{\max(f(x) - D'_i) + \min(f(x) - D'_i)}{2} \tag{3.8}$$

Thus, a new best-fit linear approximation $p^*_r(x')$ is obtained, which leads to reoccupy a balanceable minimax approximate error, ε'_a:

$$p^*_r(x') = D'_i + c''_{1i} \tag{3.9}$$

The approach, with the aim of achieving coefficients c'_{0i} and c''_{1i} which lead to the best-fit linear approximation, $p^*_r(x')$, in one segment is summarized in *Algorithm 1* and illustrated by *Example 1*. In *Algorithm 1*, the symbols of $\leftarrow$ indicate assignments; $\text{Chebyshev}(f(x), [a_i, b_i])$

is the application of the Chebyshev's theorem according to (3.6); round(x, y) is the rounding of the value of x to y bits; $\{x_i = a_i$ to b_i steps $2^{-m}\}$ means traversing all the numbers in the range of $[a_i, b_i]$ with the step of 2^{-m}; and values of parameters p and q are achieved based on the MiniBit approach presented in Section 3.3.1.

Example 1: Based on the assumption that the evaluation of the logarithm function $f(x) = \ln(1 + x)$, here x is a 16-bit BXP number, and $[a_i, b_i]$ is in the domain of $[0, 1-2^{-16}]$, we obtain the best-fit linear approximation, $p^*(x)$, with infinite precision of x, c_{0i}, c_{1i} and D_i according to (3.6), and the minimax error $\varepsilon_a = 0.0298$. After rounding c'_{0i} to the finite precision of 2^{-3}, x' to 2^{-5}, c'_{1i} and D'_i to 2^{-8}, $p^*(x)$ is changed to $p^*(x') = D'_i + 0.03125$, where $D'_i = round((0.75 \times x'), 8)$, and then the maximum absolute error becomes 0.0959. To re-determine the best-fit $p^*_r(x')$, we keep $c'_{0i} = 0.75$ and adjust the value of c'_{1i}. First, we compute all the differences between $f(x_i)$ and D'_i in $[0, 1-2^{-16}]$; and then the maximum value of $(\max(f(x) - D'_i))$ and the minimum value of $(\min(f(x) - D'_i)$ are obtained. According to (3.8), rounded c''_{1i} is achieved, $c''_{1i} = -0.0078125$. Thus, a new best-fit linear approximation $p^*_r(x')$ is obtained, $p^*_r(x') = D'_i - 0.0078125$, where $D'_i = round((0.75 \times x'), 8)$, and a new balanceable minimax approximate error is achieved, $\varepsilon'_a = 0.0572$.

3.3 A Non-Uniform Segmentation Method

The proposed dynamic non-uniform segmentation method for the linear approximation function evaluation is summarized in *Algorithm 2*, where *Minibit*$(f(x), [a, b], \varepsilon_{con})$ is the MiniBit approach presented in Section 3.3.1; and *BoundaryWidth*$(f(x), [a, b])$ is the method to evaluate the bit-width of the segment boundary described in Section 3.3.2. First, the design specifications, which include 1) a transcendental or compound function $f(x)$ to be evaluated, where x is a m-bit precision BXP operand; 2) a domain $[a, b]$ for the x; and 3) a required accuracy $\varepsilon_{con} = 2^{-n}$ for evaluation results, are supplied by the user. Second, we adopt MiniBit approach to evaluate the initial UFB, p-bit, of x', c'_{0i}, c'_{1i} and D'_i. Meanwhile, the bit-width of the segment boundary, r-bit, is determined by evaluating the function $f(x)$ in the most gradient segment, where the maximum value of the first derivative of function, $f(x)'$, occurs. Third, since the value if r is smaller than m, the value z is set as $z = 2^{-r} - 2^{-m}$ to

Algorithm 2 A dynamic non-uniform segmentation method

Inputs: 1) function $f(x)$ to be evaluated; 2) a domain $[a, b]$; 3) a required accuracy $\varepsilon_{con} = 2^{-n}$; 4) precision of x in $f(x)$, m.

Outputs: 1) non-uniform segment boundaries, $[a_0, b_0], ..., [a_i, b_i]$; 2) linear approximation coefficients in each segment, $[c'_{00}, c''_{10}], ..., [c'_{0i}, c''_{1i}]$.

1: $p \leftarrow \text{Minibit}(f(x), [a, b], \varepsilon_{con})$
2: $r \leftarrow \text{boundaryWidth}(f(x), [a, b])$
3: $z \leftarrow 2^{-r} - 2^{-m}$
4: Manually select q, such that $q \leq p$
5: $k_n \leftarrow a$
6: $l_n \leftarrow \frac{a+b}{2}$
7: $low_n \leftarrow k_n$
8: $up_n \leftarrow l_n$
9: $i \leftarrow 0$
10: **repeat**
11: $[a_i, b_i] \leftarrow [0, 0]$
12: **repeat**
13: $k_c \leftarrow k_n$
14: $l_c \leftarrow l_n$
15: $low_c \leftarrow low_n$
16: $up_c \leftarrow up_n$
17: $[[c'_{0i}, c''_{1i}], |\varepsilon_t|] \leftarrow \text{Alg1}(f(x), [k_c, l_c + z], q, p)$
18: $[[k_n, l_n], low_n, up_n, [a_i, b_i], [c'_{0i}, c''_{1i}]] \leftarrow \text{Alg3}([k_c, l_c], \varepsilon_{con}, |\varepsilon_t|, low_c, up_c, [a_i, b_i], [c'_{0i}, c''_{1i}])$
19: **until** $|l_n - l_c| < 2^{-r}$
20: **if** $[a_i, b_i] = [0, 0]$ **then**
21: the step size 2^{-r} is too large, so increment r and go back to step 2.
22: **end if**
23: $k_n \leftarrow b_i + 2^{-r}$
24: $l_n \leftarrow b$
25: $low_n \leftarrow k_n$
26: $up_n \leftarrow l_n$
27: $i \leftarrow i + 1$
28: **until** $k_n \geq b$

supplement the right boundary of segments in order to traverse all the BXP input operands. The precision of c'_{0i}, q-bit, is also selected so that the value of q is always smaller than or eauql to that of p. Fourth, the new segment boundary k_n and l_n, and the new lower and upper limit, low_n and up_n, (as the initial outputs of BSPS method, refer to *Algorithm 3*) are set as the point of a and $\frac{a+b}{2}$, respectively. Moveover, the initial obtained boundary is set as $[a_i, b_i] = [0, 0]$ to detect the valid bit-width of the segment boundary, r-bit. Fifth, the current segment boundary k_c and l_c, and the current lower and upper limit, low_c and

up_c, are set as the values of the initial outputs of *Algorithm 3*. Then, the current tested segment boundary $[k_c, l_c + z]$, and the precision of c'_{0i} and c''_{1i} in bits, p and q, as the inputs of *Algorithm 1*, are sent to achieve the best-fit coefficients, $[c'_{0i}, c''_{1i}]$, and the minimax error $|\varepsilon_t|$ in the current tested segment. Sixth, the current tested segment boundary, $[k_c, l_c]$, the current lower limit and up limit, $[low_c, up_c]$, the required accuracy, ε_{con}, the minimax error, $|\varepsilon_t|$, and initial $[a_i, b_i]$, as the input of *Algorithm 3*, are sent to generate the new boundary k_n and l_n, and the new lower and upper limit, low_n and up_n, the updated boundary, $[a_i, b_i]$, and the updated $[c'_{0i}, c''_{1i}]$. If $|l_n - l_c| < 2^{-r}$ is satisfied, the first segment $[a_i, b_i]$ and corresponding coefficients $[c'_{0i}, c''_{1i}]$ are obtained, and *Algorithm 2* sets the new segment boundary k_n and l_n, and the new lower and upper limit, low_n and up_n, as the value of b_i+2^{-r} and b respectively, then *Algorithm 2* begins to search the second segment and the corresponding coefficients. Finally, once $k_n > b$, *Algorithm 2* stops. As a result, 1) the boundaries of all segments, $[a_i, b_i]$, are determined; and 2) the optimized coefficients, c'_{0i} and c''_{1i} with less bit-width in each segment are obtained.

3.3.1 Determination of Initial UFB by MiniBit Approach

A static bit-width optimization approach, MiniBit [114], can compute the required integer and fractional bits for each signal. This approach can quickly determine the UFB for each signal to guarantee a faithful rounding (1 *ulp*) for approximated results, but it can cause a problem in which a lager (suboptimal) bit-width for each signal obtained by this approach makes the hardware implementation for function evaluation more complicated. Thus, the proposed dynamic method uses the MiniBit approach to obtain the initial p-bit UFB of x', c'_{0i}, c'_{1i} and D'_i, and then optimizes the bit-width of c'_{0i} to a less q-bit one.

We denote the rounding errors of x', c'_{0i}, c'_{1i}, and D'_i as ε_x, $\varepsilon_{c_{0i}}$, $\varepsilon_{c_{1i}}$ and ε_{D_i} respectively. According to (3.7), ε_{D_i} is computed as:

$$\varepsilon_{D_i} = c_{0i}\varepsilon_x + x\varepsilon_{c_{0i}} + \varepsilon_{c_{0i}}\varepsilon_x + 0.5 \times 2^{-FB_{D_i}} \tag{3.10}$$

In (3.10) $0.5{\times}2^{-FB_{D_i}}$ is the maximum rounding error of D_i. Thus, the error analysis of $\varepsilon_{p^*(x')}$ is represented as:

$$\varepsilon_{p^*(x')} = \varepsilon_{D_i} + \varepsilon_{c_{1i}} + |\varepsilon_r| + |\varepsilon_a| \tag{3.11}$$

Where the $\varepsilon_{D_i}+\varepsilon_{c_{1i}}$ is the quantization error, $|\varepsilon_q|$; $|\varepsilon_a|$ is the linear approximation error; and $|\varepsilon_r|$ is the final rounding error. To guarantee a n-bit accuracy, the maximum absolute error needs to satisfy:

$$\max(\varepsilon_{p^*(x')}) \leq 2^{-n} \tag{3.12}$$

To compute the maximum absolute error of $\varepsilon_{p^*(x')}$, we consider the following worst cases: 1) the maximum rounding error of $\varepsilon_x = 0.5\times 2^{-FB_x}$, $\varepsilon_{c_{0i}} = 0.5\times 2^{-FB_{c_{0i}}}$ and $\varepsilon_{c_{1i}} = 0.5\times 2^{-FB_{c_{1i}}}$; 2) the maximum absolute value of x, $|x^{\max}|$; 3) the maximum absolute value of c_{0i}, $|c_{0i}^{max}|$; 4) the maximum final rounding error, 0.5×2^{-n} (0.5 *ulp*). Under the worst conditions, we substitute (3.11) to (3.12) and obtain:

$$\begin{aligned} &|c_{0i}^{max}|\times 2^{-FB_x-1}+|x^{\max}|\times 2^{-FB_{c_{0i}}-1}+ \\ &2^{-FB_x-FB_{c_{0i}}-2}+2^{-FB_{D_i}-1}+2^{-FD_{c_{1i}}-1}\leq 2^{-n-1}-|\varepsilon_a| \end{aligned} \tag{3.13}$$

In (3.13), we consider a simple solution by using *UFB* for all signals. Then, (3.13) can be written as:

$$(|c_{0i}^{\max}|+|x^{\max}|+2+2^{-UFB-1})\times 2^{-UFB-1}\leq 2^{-n-1}-|\varepsilon_a| \tag{3.14}$$

Since $|\varepsilon_a|>0$, the initial *UFB* of x', c'_{0i}, c'_{1i} and D'_i, p-bit, is computed in order to satisfy the condition (3.14).

3.3.2 Evaluation of Bit-Width of Segment Boundary

Less bit-width of the segment boundary can not only speedup the dynamic non-uniform segmentation method, but also simplify the hardware implementation of the SIE circuit. However, if the bit-width of the segment boundary is too small, the minimal distance of the segment is too large to allow the best-fit optimized linear approximation, $p_r^*(x')$, to guarantee the accuracy in this segment. Thus, we propose an approach to evaluate the minimal bit-width of the segment boundary. First, we compute the point, x_h, where $|c_{0i}^{\max}|$ occurs, in the domain $[a, b]$. Second, the minimum distance that we can obtain in the most gradient segment of the function $f(x)$ is 2^{-r}, the domain of $[x_h, x_h+2^{-r}]$ or $[x_h-2^{-r}, x_h]$. Third, we evaluate the function by *Algorithm 1* in which x', c''_{1i}, D'_i are rounded to the precision of 2^{-p}, and c'_{0i} to 2^{-q}. Then, we increase the distance, 2^{-r}, by decreasing the value of r-bit,

the bit-width of the segment boundary. Note that the value of r-bit is decreased from the value of m-bit bit by bit. Once the approximation results, $p_r^*(x')$, can not guarantee ε_{con}, we can obtain a r-bit bit-width of the segment boundary. Since ε_q is produced by rounding errors, the value of r-bit we obtained would be too small to make $p_r^*(x')$ satisfy ε_{con} in other segments. To avoid such conditions, the bit-width of segment boundary, r-bit, needs to be further increased.

3.3.3 Partition of Non-Uniform Segments by BSPS

By BSPS, summarized in *Algorithm 3*, the proposed dynamic non-uniform segmentation method can evaluate the function $f(x)$ by the minimum number of non-uniform segments $[a_i, b_i]$, because BSPS is an efficient traversal method which can partition the domain $[k_c, l_c]$ to all the segments. In this section, we present an example, *Example 2*, to illustrate the process of partitioning non-uniform segments by BSPS:

Example 2: Evaluate the function $f(x) = \ln(1+x)$ in the domain $[0, 1)$ to guarantee a required accuracy, $\varepsilon_{con} = 2^{-8}$, where x is a 16-bit BXP operand.

To guarantee a 2^{-8} accuracy, the initial p-bit UFB of x', c'_{0i}, c'_{1i} and D'_i computed by MiniBit is 11-bit, because $|c_{0i}^{max}| = 1$ and $|x^{max}| = 1 - 2^{-16}$ (refer to (3.14)). Since $|c_{0i}^{max}|$ is achieved at the point, $x_h = 0$, the most gradient segment of the function, $\ln(1+x)$, is $[0, 2^{-r}]$. Then, we evaluate $\ln(1+x)$ by *Algorithm 1* in $[0, 2^{-r}]$ in which x', c''_{1i}, D'_i are rounded to the precision of 2^{-11}, and c'_{0i} to 2^{-4}. When r-bit is decreased to 3-bit, $p_r^*(x')$ can not guarantee the 2^{-8} accuracy. Thus, the minimum bit-width of the segment boundary, r-bit, is 4-bit. Since, 4-bit segment boundary is too small to make $p_r^*(x')$ satisfy ε_{con} in all segments, we increase it to 5-bit in the non-uniform segmentation.

To partition the non-uniform segments, first, c_{01} is obtained in the initial segment $[k_c, l_c + z] = [0, 0.5+z]$, where $z = 2^{-5} - 2^{-16}$; second, c'_{01} is rounded to the precision of 2^{-4}, $c'_{01} = 0.8125$, and x' and D'_i are rounded to the precision of 2^{-11}, where $D_i = 0.8125 \times x'$; third, the optimized $c''_{11} = 0.00732421875$ with the precision of 2^{-11} is computed by *Algorithm 1*, thus, $p_r^*(x') = D'_1 + 0.00732421875$ in the domain $[0, 0.5+z]$; fourth, since after the final rounding, $|\varepsilon_t| = 0.014646$, the maximum absolute error between $f(x)$ and $p_r^*(x')$ is larger than the required accuracy, $\varepsilon_{con} = 2^{-8}$, we reduce the segment $[0, 0.5]$ to $[k_n, l_n] = [0, 0.25]$ based on

Algorithm 3 Binary search partition scheme (BSPS)

Inputs: 1) The current segment boundary, $[k_c, l_c]$; 2) a required accuracy, ε_{con}; 3) the current accuracy, $|\varepsilon_t|$; 4) the current lower limit, low_c; 5) the current upper limit, up_c; 6) the currently approved boundary, $[a_i, b_i]$.

Outputs: 1) The new segment boundary, $[k_n, l_n]$; 2) the new lower limit, low_n; 3) the new upper limit, up_n; 4) the newly approved boundary, $[a_i, b_i]$;

1: **if** $|\varepsilon_t| \geq \varepsilon_{con}$ **then**
2: $k_n \leftarrow k_c$
3: $l_n \leftarrow \frac{low_c + l_c}{2}$
4: $low_n \leftarrow low_c$
5: $up_n \leftarrow l_c$
6: $[a_i, b_i] \leftarrow [a_i, b_i]$
7: keep $[c'_{0i}, c''_{1i}]$
8: **else** // $|\varepsilon_t| < \varepsilon_{con}$
9: $k_n \leftarrow k_c$
10: $l_n \leftarrow \frac{l_c + up_c}{2}$
11: $low_n \leftarrow k_c$
12: $up_n \leftarrow up_c$
13: $[a_i, b_i] \leftarrow [k_c, l_c]$
14: updated $[c'_{0i}, c''_{1i}]$
15: **end if**

BSPS, and re-compute $p^*_r(x')$ and $|\varepsilon_t|$ in the segment $[0, 0.25]$. Fifth, after evaluating $\ln(1+x)$ by *Algorithm 1* in segments $[k_n, l_n]$ ($[0, 0.5]$, $[0, 0.25]$, $[0, 0.125]$, $[0, 0.1875]$ and $[0, 0.15625]$) obtained by *Algorithm 3*, the first segment [0,0.125] and the coefficient $c'_{00} = 0.9375$ and $c''_{01} = 0.00048828125$ are obtained. Sixth, we change the initial domain to $[k_c, l_c] = [0.15625, 1]$, and the proposed method starts to partition the second segment. Thus, the function $ln(1+x)$ is split into 6 segments, in each of which c'_{0i} with the precision of 2^{-4}, and x', c''_{1i} and D'_i with the precision of 2^{-11} can satisfy the required accuracy, 2^{-8}.

3.4 Hardware Architecture

Figure 3.1 shows the hardware architecture for function evaluations, which mainly consists of six units: 1) an input rounding register for rounding m-bit x to p-bit x'; 2) a SIE circuit for producing the segment index, i, according to r-bit most significant bits (MSBs) of x; 3) a coefficients look-up table for storing the q-bit c'_{0i} and p-bit c''_{1i} in each segment; 4) a multiplier for computing p-bit $D'_i = Round(c'_{0i} \times x')$; 5) an adder for computing p-bit $p^*_r(x') = D'_i + c''_{1i}$; and

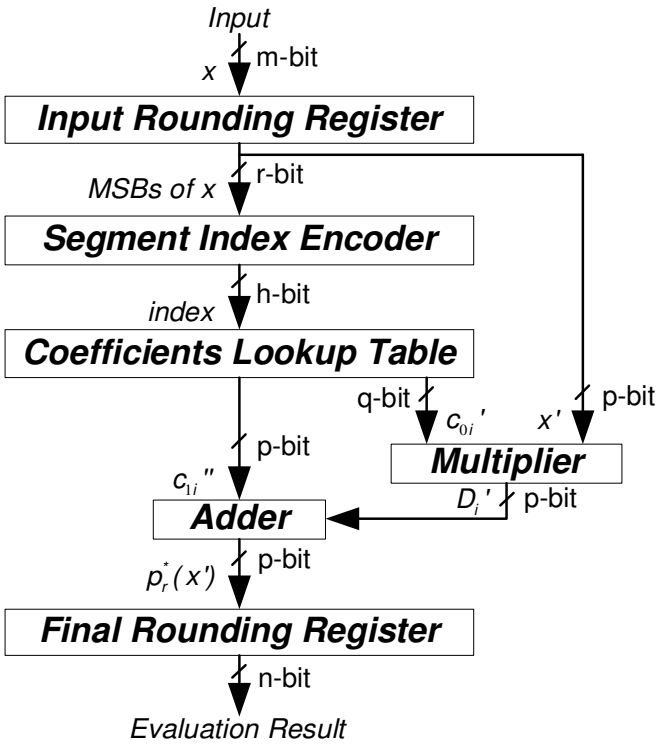

Figure 3.1: FPGA implementation for function evaluations.

6) a final output rounding register for rounding p-bit $p_r^*(x')$ to n-bit exact evaluation results. All signals in the data-path are represented by the 2's complement number. The BXP input/output rounding register, multiplier and adder in the architecture can be realized by Xilinx intellectual property (IP) core and look-up tables (LUTs) on FPGA. The coefficients look-up table and SIE circuit are implemented by the read only memory (ROM) on FPGA.

3.4.1 Segment Index Encoder

The most complicated unit in the architecture is the SIE circuit. The function of SIE is shown in (3.15):

$$SIE_{func}(x) : \{0,1\}^r \rightarrow \{0,1,...,i-1\} \tag{3.15}$$

In (3.15), the input of the SIE function is r-bit MSBs of m-bit x, and the output is the segment index, i, a h-bit BXP number, applied to the address of the coefficients look-up table.

The SIE circuit can be realized by the LUT cascade as shown in Figure 3.2. The LUT

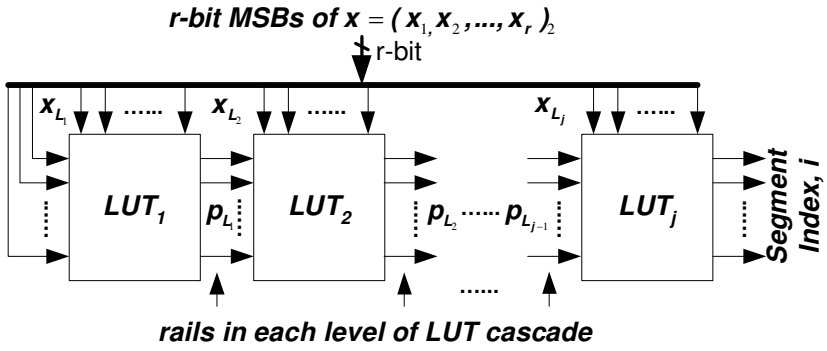

Figure 3.2: SIE circuit by the LUT cascade.

cascade is achieved by a functional decomposition based on the multi-terminal binary decision diagram (MTBDD) approach [110]. The architecture of the SIE circuit is a combinational logic in which each LUT can be realized by a block RAM on FPGA. In the leftmost LUT_1, all the inputs come from r-bit MSBs of x, x_{L_1}. For the other LUT_j, some inputs come from r-bit MSBs of x, x_{L_j}, and the others come from the output of the previous LUT_{j-1} stage, $p_{L_{j-1}}$, which is called rails. The outputs of the rightmost LUT_j then represent h-bit segment index, i.

3.4.2 Estimation of Memory Sizes

In [110], each LUT is restricted to $h+2$-bit inputs and h-bit outputs, where $h = \lceil \log_2(i) \rceil$, and i is the number of non-uniform segments. In this chapter, we use the same LUT cascade approach as [110] to implement the SIE circuit. Thus, the estimated memory size of the LUT cascade (ROM_1) is determined by the number of segment, i, and the bit-width of segment boundary, r-bit:

$$ROM_1 = 2^{\lceil \log_2(i) \rceil + 1} \times \lceil \log_2(i) \rceil \times (r - \lceil \log_2(i) \rceil) \tag{3.16}$$

In Figure 3.1, the coefficients look-up table has 2^h words, in which the coefficients, q-bit c'_{0i} and p-bit c''_{1i}, are stored respectively. The memory size of the coefficients look-up table (ROM_2) is estimated as follow:

$$ROM_2 = 2^{\lceil \log_2(i) \rceil} \times (p+q) \tag{3.17}$$

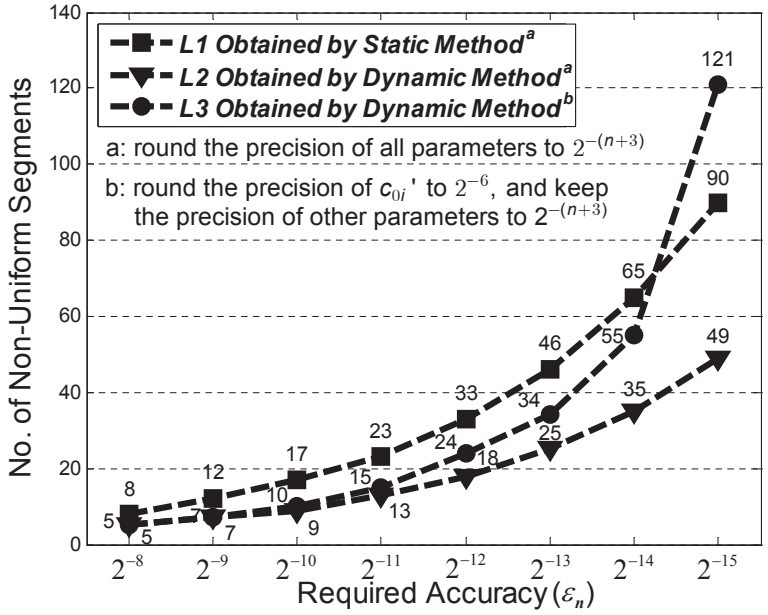

Figure 3.3: Non-uniform segmentation results vs. varied ε_n.

Thus, the total memory size (ROM_t) for implementing the coefficients look-up table and the SIE circuit is:

$$ROM_t = 2^{\lceil \log_2(i) \rceil} \times (2r \times \lceil \log_2(i) \rceil - 2 \lceil \log_2(i) \rceil^2 + p + q) \tag{3.18}$$

3.5 Experimental Results

3.5.1 Comparison Results

The previous static non-uniform segmentation methods presented in [110], [111] mainly have the following four features: 1) the function $f(x)$ is divided into non-uniform segments by a specified partition scheme; 2) the infinite best-fit linear approximation is determined by Chebyshev theorem in each segment; 3) the optimized UFB of the internal signals is analyzed using MiniBit in order to satisfy the required accuracy; 4) the internal signals are rounded to the finite precision of 2^{-UFB} in the hardware implementation.

In this section, we evaluate the function $\ln(1+x)$ of *Example 2* based on the previous static non-uniform segmentation method [110] (namely *Static Method*) and the proposed dynamic non-uniform segmentation method (namely *Dynamic Method*) to satisfy different required accuracy in two experiments. Note that the domain of the function is partitioned

at the point where the maximum absolute error occurs in *Static Method*. According to the MiniBit approach, the precision of the internal signals has to be at least $2^{-(n+3)}$ in order to guarantee the 2^{-n} accuracy. In the first experiment, the internal signals are rounded to the precision of $2^{-(n+3)}$, and then we evaluate the function $\ln(1+x)$ by *Static Method* and *Dynamic Method* to obtain *L1* and *L2* respectively as shown in Figure 3.3. It is proved that *Dynamic Method* obtains less segments than *Static Method* when the internal signals are rounded to the same precision. In the second experiment, we round the precision of c'_{0i} to 2^{-6} and keep the precision of x', c''_{1i} and D'_i to $2^{-(n+3)}$. As a result, we acquire another line, *L3*, as shown in Figure 3.3, which indicates that *Dynamic Method* still satisfy the required accuracy, even though the precision of internal signals is smaller than 2^{-UFB}, which is not achievable from the previous *Static Method*.

3.5.2 Evaluation Results for More Functions

To further analyze the performance of the proposed dynamic non-uniform segmentation method, we evaluate more elementary and compound functions (chosen based on [110]) with two required accuracy cases: 2^{-15} (namely *case 1*) and 2^{-23} (namely *case 2*). By way of contrast, we evaluate the functions through two experiments. In the first experiment, since the rounding error of x is not considered in [110] ($\varepsilon_x = 0$), and the precision of the internal signals has to be rounded to the precision of $2^{-(n+3)}$ based on the MiniBit approach, we keep m-bit x to the precision of 2^{-m}, and round the precision of c'_{0i}, c''_{1i} and D'_i to $2^{-(n+3)}$ (2^{-18} in *case 1*, 2^{-26} in *case 2*) to evaluate all the functions by *Dynamic Method* for a fair comparison. In the second experiment, we round the precision of c'_{0i} to 2^{-6}, c''_{1i} and D'_i to 2^{-16} in *case 1*; c'_{0i} to 2^{-10}, c''_{1i} and D'_i to 2^{-24} in *case 2*; and keep m-bit x to the precision of 2^{-m} to evaluate the functions by *Dynamic Method*.

In Table 3.1, the function evaluation results of the two experiments by *Dynamic Method* are compared with the results in [110]. The results indicate that 1) *Dynamic Method* can evaluate all the functions by less segments in both *case 1* and *case 2* in the first experiment. The average ratio are $R1_{ave} = 47\%$ in *case 1* and $R1_{ave} = 52\%$ in *case 2*; 2) although the precisions of c'_{0i}, c''_{1i} and D'_i are rounded to a number that smaller than 2^{-n-3} for all the functions in the second experiment, *Dynamic Method* still meet the accuracy requirement

for both cases, which is impractical in *Static Method*. The number of segments obtained by *Dynamic Method* in the second experiment is larger than the one in the first experiment, but for the most of functions, it is still smaller than the one obtained in *Static Method* [110]. The average ratio are $R2_{ave} = 69\%$ in *case 1* and $R2_{ave} = 81\%$ in *case 2*. *Dynamic Method* has its own advantages because: 1) it is a non-uniform segmentation method mainly based on a dynamic bit-width analysis; 2) it can determine the best-fit linear approximation, in which x', c'_{0i}, c''_{1i} and D'_i can be rounded to the less bit-width in each segment.

3.5.3 Memory Sizes for Two Methods

The memory resources on FPGA are occupied by two blocks in the proposed architecture for function evaluations: the SIE circuit and the coefficients look-up table. The memory sizes occupied by these two blocks are determined by the following parameters: 1) the number of segments, i; 2) the bit-width of segment boundary, r-bit; and 3) the bit-width of coefficients, q-bit c'_{0i} and p-bit c''_{1i}, which are achieved by the proposed dynamic non-uniform segmentation method. In Table 3.2, we compare the estimated memory sizes for *Dynamic Method* with those for *Static Method* [110]. By way of contrast with *Static Method*, we estimate the memory sizes for two required accuracy, 2^{-15} in *case 1* and 2^{-23} in *case 2* through two experiments in Section 3.5.1. The values of estimated memory sizes are achieved according to equation (3.18) derived in Section 3.4.2.

The comparison results indicate that 1) the memory sizes for all functions evaluated by *Dynamic Method* are smaller than those for *Static Method* in both *case 1* and *case 2* in the first experiment. The average ratio are $R1_{ave} = 48\%$ in *case 1* and $R1_{ave} = 38\%$ in *case 2*; 2) the memory sizes for *Dynamic Method* in the second experiment are larger than those in the first experiment, but most of which are still smaller than the memory size in [110]. The average ratio are $R2_{ave} = 56\%$ in *case 1* and $R2_{ave} = 71\%$ in *case 2*. *Dynamic Method* needs less memory sizes due to the following reasons: 1) the proposed *Dynamic Method* can evaluate functions using fewer segments, i; 2) it can determine the minimum bit-width of segment boundary, r-bit; 3) it enables c'_{0i}, c''_{1i} stored in coefficients look-up table, to have the minimum bit-width in each segment.

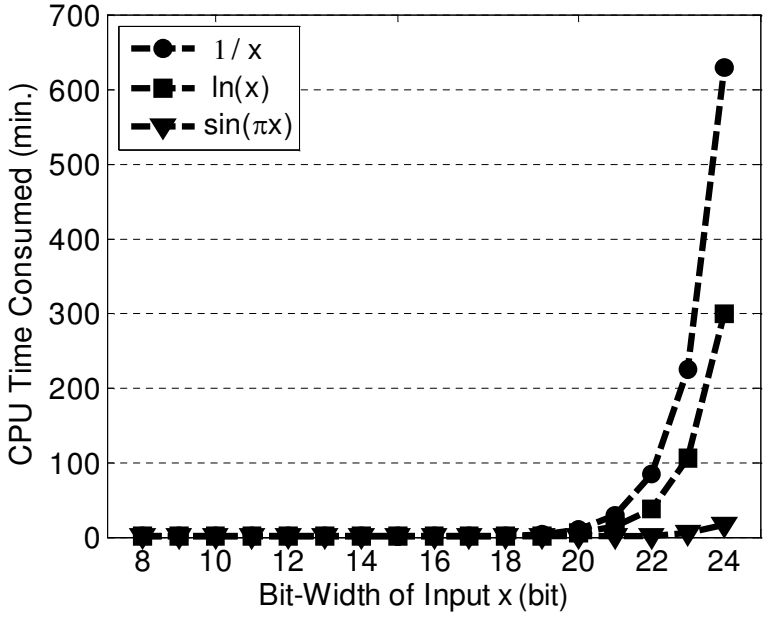

Figure 3.4: CPU time analysis of *Dynamic Method*.

3.5.4 CPU Time Consumed

The CPU time consumed by *Dynamic Method* to evaluate the functions in the first experiment is demonstrated in Table 3.1. Compared with *Static Method* [110], the CPU time consumed by *Dynamic Method* is much longer due to its basis of a dynamic bit-width analysis, in which the design is simulated over all input values and the maximum absolute errors at the outputs are monitored. Note that the CPU time is evaluated in different computer systems and software compilers. Figure 3.4 shows the CPU time, consumed by evaluating the most time-consuming functions, $1/x$, the least time-consuming function, $\sin(\pi x)$, and the average time-consuming function, $\ln(x)$ (refer to Table 3.1), for different bit-widthes of the input x, m-bit. The specified accuracy in this experiment for evaluating results is 2^{-m+1}. According to Figure 3.4, if the bit-width of the input, x, is less than 20-bit, the CPU time consumed by *Dynamic Method* is within 10 *min.* When the bit-width of the input x is 20-bit, the CPU time consumed by evaluating functions $1/x$, $\sin(\pi x)$ and $\ln(x)$ are 9.93, 0.23 and 4.5 *min.* respectively. If the bit-width of the input, x, is larger than 20-bit, the CPU time consumed by *Dynamic Method* increase drastically. Thus, Figure 3.4 indicates that if the bit-width of the input is relatively small, for example, less than 20-bit in this

experiment, *Dynamic Method* is efficient for function evaluations in hardware or a specific computer arithmetic design.

3.6 Summary

In this chapter, we develop a dynamic non-uniform segmentation method for the first-order polynomial-based function evaluation. First, we analyze the minimax linear approximation in one segment. Second, we present this method in detail, and illustrate it by an example. Third, we provide the architecture for function evaluations, and analyze the memory size occupied by the SIE circuit and the coefficients look-up table. Fourth, we evaluate more functions through two experiments to compare the performance of the proposed dynamic non-uniform segmentation method and the previous static method. Compared with the static method [110], the proposed method can achieve fewer segments, and at the same time, satisfy the required accuracy. Also, the input, coefficients, and intermediate values in each segment can be rounded to a smaller bit-width, which is impractical in the previous static method. The advantage of the proposed method leads to a significant reduction of memory size for a simpler and faster function evaluations or a specific computer arithmetic implementation on FPGAs.

Table 3.1: Number of non-uniform segments obtained by *Static Method* and *Dynamic Method*.

Function $f(x)$	**Domain** $[a,b]$	**Case 1:** *Accuracy Constraint* 2^{-15} (*x is a 16-bit BXP Number; $\varepsilon_x=0$*)							**Case 2:** *Accuracy Constraint* 2^{-23} (*x is a 24-bit BXP Number; $\varepsilon_x=0$*)						
		No. of Segments			R1	R2	Time (sec)		No. of Segments			R1	R2	Time (sec)	
		SM	DM_1	DM_2	%	%	*SM*	*DM*	*SM*	DM_1	DM_2	%	%	*SM*	*DM*
e^x	$[0,1)$	128	90	197	70	154	0.01	0.67	2,048	1,478	3,113	72	152	0.06	2,476
$1/x$	$[1/32,1]$	1,721	766	965	45	56	0.07	7.80	28,010	14,319	19,430	51	69	1.04	37,684
$1/\sqrt{x}$	$[1/32,1]$	620	319	448	51	72	0.03	3.36	9,946	5,227	8,282	53	83	0.31	14,206
$\sqrt{x}$	$[0,1]$	231	118	272	51	117	0.01	1.32	3,941	2,039	4,801	52	122	0.12	5,484
$\ln(x)$	$[1/256,1]$	726	337	476	46	66	0.04	4.03	11,761	5,820	8,931	49	76	0.48	17,982
$x\ln(x)$	$(0,1)$	282	132	232	47	82	0.01	1.44	4,535	2,184	4,210	48	93	0.13	5,697
$\sqrt{-\ln(x)}$	$(0,1)$	584	286	439	49	75	0.04	3.11	12,089	6,044	9,303	50	77	1.71	17,279
$\sin(\pi x)$	$[0,1/2]$	127	94	138	74	109	0.01	0.34	2,027	1,407	2,489	69	123	0.07	974
$\arcsin(x)$	$[0,1]$	260	128	272	49	105	0.02	0.67	4,415	2,268	5,036	51	114	0.17	1,628
$\tan(\pi x)$	$[0,31/64]$	1,328	579	701	44	53	0.10	1.21	20,770	11,427	14,681	55	71	1.34	3,738
Avg.	*null*	601	285	414	47	69	0.03	2.40	9,954	5,221	8,017	52	81	0.54	10,715

[#] *SM*: *Static Method* [110]; DM_1 and DM_2: *Dynamic Method* in experiment 1 and 2; $R1$: $\frac{DM_1}{SM}\times 100$; $R2$: $\frac{DM_2}{SM}\times 100$;

[*] *Time*: CPU time for *Static Method* and *Dynamic Method* conducted on the following environments respectively: *Static Method*: System: Sun Blade 2500 (Sliver), CPU: UltraSPARC-IIIi 1.6GHz, Memory: 6GB, OS: Solaris 9, C complier: gcc -O2 [8]. *Dynamic Method*: System: Lenovo ThinkCentre M55-8811, CPU: Intel(R) Core(TM)2 CPU 6600 2.4GHz, Memory: 3GB, Os: Microsoft Windows XP, Matlab complier: R2009a.

Table 3.2: Estimated memory sizes (bits) obtained based on *Static Method* and *Dynamic Methods*.

Function $f(x)$	**Case 1:** *Accuracy Constraint* 2^{-15} (x *is a* 16*-bit BXP Number;* $\varepsilon_x=0$)							**Case 2:** *Accuracy Constraint* 2^{-23} (x *is a* 24*-bit BXP Number;* $\varepsilon_x=0$)						
	Estimated ROM Sizes (bit)			R1	R2	BTSB (r-bit)		Estimated ROM Sizes (bit)			R1	R2	BTSB (r-bit)	
	SM	DM_1	DM_2	%	%	DM_1	DM_2	*SM*	DM_1	DM_2	%	%	DM_1	DM_2
e^x	20,096	9,984	17,920	50	89	10	11	681,984	241,664	630,784	35	92	14	17
$1/x$	317,440	159,744	145,408	50	46	16	16	11,108,352	3,604,480	7,012,352	32	63	20	21
$1/\sqrt{x}$	168,960	55,296	57,344	33	34	13	14	5,718,016	1,703,936	3,309,568	30	58	19	20
$\sqrt{x}$	44,288	20,736	75,776	47	171	16	16	1,462,272	692,224	2,621,440	47	179	24	24
$ln(x)$	168,960	73,728	75,776	44	45	15	16	5,718,016	2,129,920	3,768,320	37	66	21	21
$xln(x)$	78,336	33,792	34,340	43	44	14	15	2,695,168	1,196,032	2,408,448	44	89	22	23
$\sqrt{-\ln(x)}$	168,960	82,944	75,776	49	45	16	16	5,718,016	2,768,896	5,144,576	48	90	24	24
$\sin(\pi x)$	20,096	8,192	22,016	41	110	9	12	681,984	241,664	827,392	35	121	14	19
$\arcsin(x)$	87,552	20,736	75,776	24	87	16	16	2,908,160	1,392,640	2,621,440	48	90	24	24
$\tan(\pi x)$	227,328	159,744	145,408	70	64	16	16	9,142,272	3,604,480	4,227,072	39	46	20	22
Avg.	130,202	62,490	72,554	48	56	14	15	4,583,424	1,757,593	3,257,139	38	71	21	22

SM: *Static Method* [110]; DM_1 and DM_2: *Dynamic Method* in experiment 1 and 2; $R1$: $\frac{DM_1}{SM}\times 100$; $R2$: $\frac{DM_2}{SM}\times 100$; *BTSB*: bit-width of segment boundary, r-bit.

Chapter 4

Decimal Logarithmic and Antilogarithmic Converters

This chapter presents decimal logarithmic and antilogarithmic converters based on the decimal first-order polynomial (linear) approximation algorithm. The proposed approach is mainly based on a look-up table, followed by a decimal linear approximation step. Compared with a binary-based decimal linear approximation algorithm (Alg. 1), the proposed algorithm (Alg. 2) is error-free in the conversion between decimal and binary format. The proposed architectures are implemented only by the combinational logic in the binary coded decimal (BCD) encoding on FPGAs. In this chapter, we analyze the tradeoff of the hardware performance, scale up the proposed logarithmic architecture to achieve a higher accuracy. Finally, we compare the hardware performance of Alg. 1 and Alg. 2, the results show that although it occupies 1.14 times more area, the proposed decimal logarithmic converter (Alg. 2) can run 2.15 times faster than the binary-based decimal logarithmic converter (Alg. 1).

4.1 Introduction

The piecewise linear approximation algorithm is an attractive method because logarithm and antilogarithm functions can be evaluated by a set of simple linear approximations. In particular, if the objective is to use the logarithmic unit to implement high-speed and low-power applications, which have some tolerance for errors. In this chapter, the decimal piecewise linear approximation algorithm based approach is proposed to implement decimal logarithmic and antilogarithmic converters. The number format used in the converter is a 32-bit decimal DFP storage format specified in IEEE 754-2008, Decimal32, in which the

decimal significand is defined as a 7-digit non-normalized DXP number based on the BCD encoding. Thus, we focus on the algorithm and architecture of the 7-digit DXP logarithmic and antilogarithmic converters that can compute the required accuracy of faithful logarithm and antilogarithm results. The architectures of the proposed converters are combinational logics which take a single clock cycle to compute a decimal result. As far as we know, this work is the first study for the decimal logarithmic and antilogarithmic converters based on the decimal piecewise linear approximation algorithm.

This chapter is organized as follows: Section 4.2 describes two straight-line approximation algorithm based on approaches for computing the decimal logarithm and antilogarithm. In Section 4.3, we present a dynamic non-uniform segmentation method (modified method based on Chapter 3) to evaluate the decimal logarithm and antilogarithm function based on linear approximation. Section 4.4 describes the comparison in terms of the error analysis for two algorithms for the decimal logarithm computation. In Section 4.5, the architecture of the proposed decimal logarithmic and antilogarithmic converters are presented. Section 4.6 analyzes the implementation and comparison results on FPGA. Section 4.7 gives conclusions. This chapter is an extension of the research presented in [116, 117].

4.2 Decimal Logarithm and Antilogarithm Conversion

4.2.1 Binary-based Decimal Logarithm Conversion (Alg. 1)

Mitchell [118] presented a straight-line approximation method to obtain logarithms of a binary operand. Based on Mitchell's algorithm, several techniques to compute binary logarithm results based on a piecewise linear approximation and the hardware implementations are presented in [119, 120, 82, 85, 87]. The computation of $\log_{10}(dec)$ can be achieved by a simple transformation from the approximation of $\log_2(bin)$ to $\log_{10}(bin)$ as shown in (1), where *dec* and *bin* are used to present a decimal and BXP number respectively.

$$\log_{10}(dec) \approx \log_{10}(bin) = \log_2(bin) \times \log_{10}(2) \tag{4.1}$$

However, it is evident that the method described above is error-prone because many fractions such as 0.1 can not be exactly represented as binary numbers. Using this approach,

the errors generated by conversion between decimal and binary format can not be avoided. Therefore, there is a need for a new decimal logarithm algorithm which is error-free in the conversion between decimal and binary format. The binary-based decimal linear approximation algorithm (referred to as Alg. 1) is simulated using MATLAB as a benchmark to compare with the following decimal linear approximation algorithm in Section 4.4.

4.2.2 Decimal Logarithm Conversion (Alg. 2)

The linear approximation algorithm to compute $\log_{10}(dec)$ is summarized as follows: since the decimal significand, N, is a 7-digit non-normalized DXP number in the range of $0 \leq N \leq 10^7$–1, N can be represented by:

$$N = \sum_{i=j}^{k} 10^i z_i \tag{4.2}$$

Where, z_i is a decimal digit, $'0', '1', ..., '8', '9'$, and $0 \leq j \leq k \leq 6$. Since N is a 7-digit non-normalized DXP operand, it should be adjusted into the range of $[0.1, 1)$ before it is computed. Thus, N can be written as:

$$N = 10^{k+1} \sum_{i=j}^{k} 10^{i-k-1} z_i \tag{4.3}$$

Factoring by 10^{k+1}, (4.3) becomes:

$$N = 10^{k+1}(0 + \sum_{i=j}^{k} 10^{i-k-1} z_i) \tag{4.4}$$

Let the term

$$\sum_{i=j}^{k} 10^{i-k-1} z_i = m \tag{4.5}$$

Where m is in the interval $0.1 \leq m < 1$, thus,

$$N = 10^{k+1}(m) \tag{4.6}$$

The logarithm of N is computed as:

$$R = \log_{10}(N) = k + (1 + \log_{10}(m)) \tag{4.7}$$

Since $1+\log_{10}(m)$ are in the range of $[0,1)$, the integer portion of the logarithm or characteristic is k, and the fraction portion of the logarithm or mantissa is only a function of m. Thus, the term $1+\log_{10}(m)$ can be computed by its piecewise linear approximation:

$$1+\log_{10}(m) \approx c_{0i} \times m + c_{1i} \tag{4.8}$$

Where c_{0i} and c_{1i} are coefficients in each piecewise segment, and i represents the index of different segments. Thus, the logarithm result, R, is achieved as:

$$R = \log_{10}(N) = k + (1+\log_{10}(m)) \approx k + c_{0i} \times m + c_{1i} \tag{4.9}$$

4.2.3 Decimal Antilogarithm Conversion (Alg. 3)

Let R be the logarithm of a decimal operand in which the characteristic of the logarithm is represented by k and the mantissa is represented by $c_{0i} \times m + c_{1i}$. R is given in as equation (4.9). The decimal antilogarithm calculations are based on piecewise approximations to the antilogarithm curve of the decimal logarithm. A linear approximation algorithm to calculate the antilogarithm of R ($Antilog_{10}(R)$) is summarized as follows:

$$Antilog_{10}(R) = 10^R \tag{4.10}$$

According to (4.9), we obtain,

$$10^R = Antilog_{10}(R) = 10^k \times 10^{c_{0i} \times m + c_{1i}} \tag{4.11}$$

To achieve the antilogarithm approximation, 10^k is obtained by shifting the result since characteristic k is a 1-digit integer. The approximation of logarithm $c_{0i} \times m + c_{1i}$ is a 6-digit fraction bounded by $0 \leq c_{0i} \times m + c_{1i} < 1$, so $10^{c_{0i} \times m + c_{1i}}$ is obtained by approximation:

$$Antilog_{10}(R) = 10^k (d_{0i} \times m + d_{1i}) \tag{4.12}$$

Assuming that a decimal logarithm R is 4.514562, it represents the logarithm of a 7-digit DXP number. According to the algorithm described above, the formula should be $10^R = Antilog_{10}(R) = 10^k 10^{c_{0i} \times m + c_{1i}} = 10^4 10^{0.514562}$, where $10^{c_{0i} \times m + c_{1i}}$ is achieved by the linear approximation $(d_{0i} \times m + d_{1i})$, and 10^k is obtained by a shift operation.

4.3 Piecewise Linear Approximation Method

4.3.1 Notations

In this section, we deal with the linear approximation evaluation of the function, $f(m) = 1+\log_{10}(m)$, with its input and output in a 7-digit DXP format. With the piecewise linear approximation, errors are produced in three ways. The first one is the maximum linear approximation error, ε_a, because of the difference of the function $1+\log_{10}(m)$ and its minimax linear approximation:

$$\varepsilon_a = \max_{a_i \le m < b_i} |(1 + \log_{10}(m)) - (c_{0i} \times m + c_{1i})| \tag{4.13}$$

The second one is the quantization error, ε_q, as shown in (4.14), produced by the finite precision of rounded inputs, $m' = truncate(m)$, coefficients, $c'_{0i} = round(c_{0i})$ and $c'_{1i} = round(c_{1i})$, and intermediate values, $D'_i = truncate(c'_{0i} \times m')$, in the hardware implementation. Considering a truncation mode for inputs, m', and intermediate values, D'_i, instead of a rounding mode is because that the truncation mode can reduce the rounding delay resulting from an extra decimal addition in the hardware architecture. Note that in this work, m, c_{0i}, c_{1i} and $D_i = c_{0i} \times m$ represent infinite precision inputs, coefficients and intermediate values, while m', c'_{0i}, c'_{1i} and D'_i represent rounded coefficients, truncated inputs and intermediate values.

$$\varepsilon_q = (c_{0i} \times m + c_{1i}) - (D'_i + c'_{1i}) \tag{4.14}$$

The third one is the final output rounding error, ε_r, whose maximum value is 0.5 unit in the last place (*ulp*). In order to obtain a n-digit accuracy, the following condition must be satisfied:

$$\varepsilon_t = \varepsilon_a + \varepsilon_q + \varepsilon_r \le 10^{-n} \tag{4.15}$$

4.3.2 Decimal Minimax Error Analysis in One Segment

In each segment, the best-fit straight line can be found by Chebyshev theorem [32] which gives a characterization of the minimax approximations to a function. The detail of the minimux error analysis is given in Section 3.2.2 by *Algorithm 1.* Figure 4.1 demonstrates

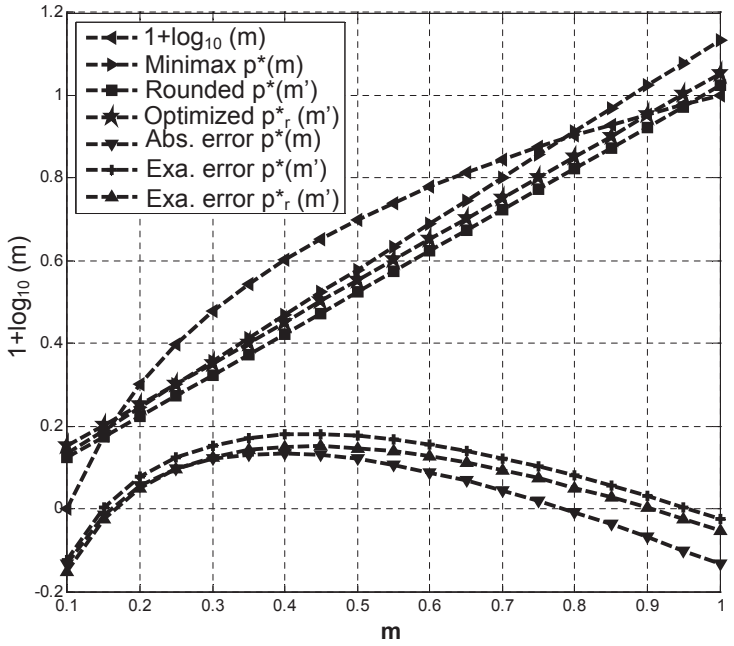

Figure 4.1: Optimization of linear approximation in one segment of decimal logarithm.

the optimization of linear approximation for minimax absolute error in the largest segment $[0.1, 1)$ for the decimal logarithm. Assuming m', c'_{1i} and D'_i are rounded to the finite digit-width of 4-digit, and c'_{0i} is rounded to the finite digit-width of 1-digit, the original minimax linear approximation $p^*(m)$, achieved by Chebyshev theorem based method is rounded to $p^*(m')$, $1\times m'+0.0233$, and the maximum absolute error is increased from the original linear approximations' 0.134 to 0.180. Thus, the value of c'_{0i} is kept and the value of c'_{1i} is adjusted to c''_{1i}, which is 0.0518. Then, the new linear approximation, $p^*_r(m') = 1 \times m' + 0.0518$, is determined and the maximum absolute error is decreased from 0.180, given by the linear approximations, to 0.152.

4.3.3 Decimal Dynamic Non-Uniform Segmentation Method

To achieve a specific accuracy, the interval of m can be split into a set of segments with the same size. Such an approach is called uniform segmentation method [106, 109]. However, by this method, the numerous segments make using look-up table for storing the coefficients impractical. A more effective approach is to determine the segment that has the largest size while maintaining the specified approximation accuracy. Such an approach with different

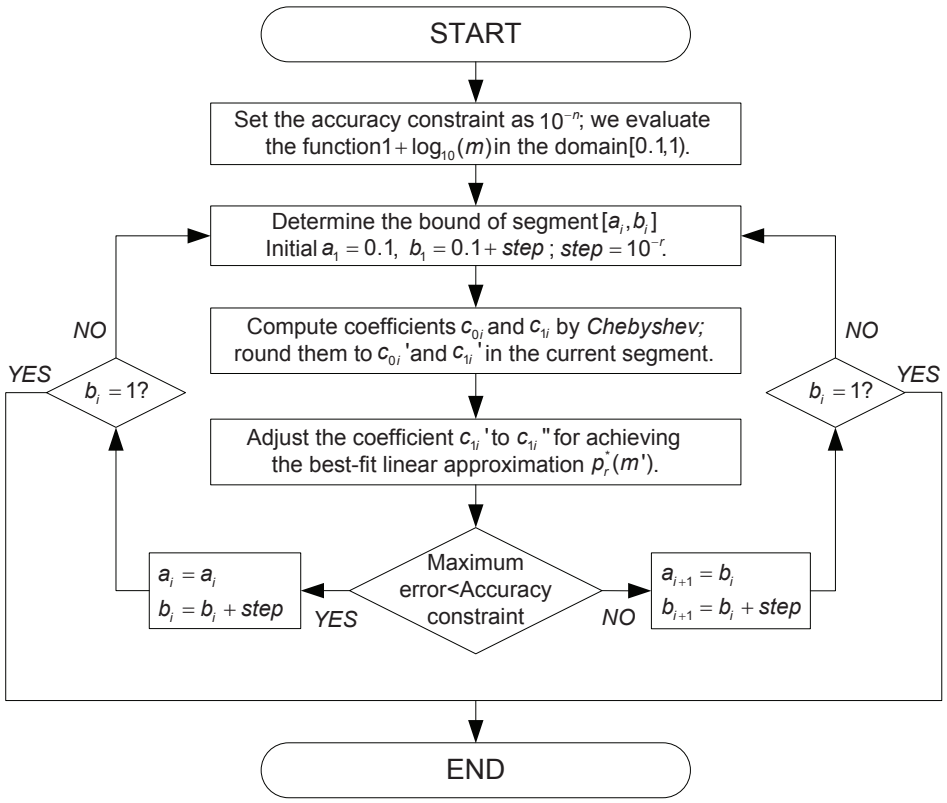

Figure 4.2: Proposed dynamic non-uniform segmentation method.

widthes is called a non-uniform segmentation method and the examples are presented in [110, 111] in detail.

In this work, a new dynamic non-uniform segmentation method is proposed to obtain the linear approximation of the decimal logarithm and antilogarithm operations as shown in Figure 4.2. Since the proposed method uses a dynamic error analysis instead of a static error analysis [114], it can determine the least number of segments as well as give a smaller digit-width of the coefficients in each segment. The first step of this method is to set the accuracy constraint. Any accuracy constraint can be set in the proposed method, for instance, we can set the accuracy constraint as 10^{-n} to achieve the n-digit fractional accuracy of logarithm and antilogarithm results. The second step is to determine the current bound of segment $[a_i, b_i]$, initialized as $a_1 = 0.1$ and $b_1 = 0.1 + step$, where $step$ is defined as 10^{-r} to adjust the segment region in each iteration, and the value of r represents the digit-width of the segment boundary. The third step is to obtain the best coefficients c_{0i} and c_{1i} by Chebyshev theorem based method in the current segment, then c'_{0i} and c'_{1i} are rounded to the finite digit-width

of q-digit and p-digit respectively. The fourth step is to adjust the rounded coefficient c'_{1i} to optimized c''_{1i}, which can achieve the minimax absolute error operated with the rounded input, m', and intermediate values, D'_i. Then, the linear approximation of decimal logarithm, $D'_i + c''_{1i}$, is finally rounded to the specific n-digit and the final rounding error is considered. In the fifth step, if the maximum absolute error, $|\varepsilon_t|$, is smaller than accuracy constraint, the current segment, $[a_i, b_i]$, is enlarged to $[a_i, b_i + step]$, and the method goes back to the second step. Otherwise, the next segment, $[a_{i+1}, b_{i+1}]$, is set as $[b_i, b_i + step]$ and one bound of segment, $[a_i, b_i - step]$, is obtained. Finally, when $b_i = 1$ or $b_{i+1} = 1$, the method is completed. The decimal logarithm curve is divided into several segments in each of which the rounded input m', optimized coefficients c'_{0i} and c''_{1i}, and intermediate value D'_i can acquire the best-fit linear approximation to satisfy the accuracy constraint.

4.3.4 Approximation Results for Decimal Logarithm

To compute the decimal logarithm operation, we evaluate the function $1 + \log_{10}(m)$ on the interval $[0.1, 1)$ using the proposed dynamic non-uniform segmentation method. First, the accuracy constraint, 10^{-3}, is set in order to guarantee the precision of 10^{-3} decimal accuracy, which has the same dynamic range as the precision of 2^{-9} binary accuracy. Second, the initial bound of segment $[a_1, b_1]$, is set as $[0.1, 0.101]$, in which the best-fit infinite coefficients c_{00} and c_{01} are obtained by Chebyshev theorem based method. Third, the coefficient c'_{01}, the input value x', and intermediate values D'_0 are rounded to the finite digit-width of 4-digit, and the coefficient c'_{00} is rounded to the finite digit-width of 3-digit. Fourth, the value of c'_{01} is optimized to c''_{01} for achieving the best-fit linear approximation, $p^*_r(m')$. Fifth, since the maximum absolute error, $|\varepsilon_t|$, is smaller than the error constraint, 10^{-3}, we enlarge the current segment to $[0.1, 0.102]$ and and re-compute $p^*_r(m')$ and $|\varepsilon_t|$. After several iterations, the first segment $[0.1, 0.107]$ and the coefficients, $c'_{00} = 4.18$ and $c''_{01} = -0.4176$, are obtained. Sixth, we change the initial domain to $[0.108, 0.109]$, and the proposed method starts to partition the second segment. Finally, the function $1 + \log_{10}(m)$ on the interval $[0.1, 1)$ is split into 20 segments. Table 4.1 demonstrates the optimized coefficients, the boundaries of each segment, and the maximum absolute errors of the linear approximation for a decimal logarithmic converter.

Table 4.1: Parameters of decimal logarithm linear approximation (Alg. 2).

Index	**Segments**	c'_{0i}	c''_{1i}	$\lvert\varepsilon_t\rvert$
1	[0.100, 0.107]	4.18	-0.4176	0.875209E-3
2	[0.108, 0.119]	3.81	-0.3775	0.995479E-3
3	[0.120, 0.132]	3.44	-0.3331	0.950724E-3
4	[0.133, 0.148]	3.08	-0.2852	0.985012E-3
5	[0.149, 0.168]	2.74	-0.2345	0.958700E-3
6	[0.169, 0.189]	2.42	-0.1805	0.932669E-3
7	[0.190, 0.210]	2.17	-0.1331	0.923420E-3
8	[0.211, 0.236]	1.94	-0.0845	0.985260E-3
9	[0.237, 0.266]	1.73	-0.0348	0.976984E-3
10	[0.267, 0.302]	1.53	0.0185	0.993057E-3
11	[0.303, 0.341]	1.35	0.0729	0.990850E-3
12	[0.342, 0.388]	1.19	0.1276	0.973767E-3
13	[0.389, 0.439]	1.05	0.1820	0.949947E-3
14	[0.440, 0.501]	0.923	0.2379	0.988938E-3
15	[0.502, 0.568]	0.812	0.2936	0.964066E-3
16	[0.569, 0.647]	0.715	0.3488	0.961491E-3
17	[0.648, 0.733]	0.629	0.4045	0.956102E-3
18	[0.734, 0.833]	0.555	0.4588	0.990199E-3
19	[0.834, 0.952]	0.487	0.5156	0.999405E-3
20	[0.953, 0.999]	0.445	0.5551	0.564950E-3

Based on the proposed 20-segment straightforward decimal logarithm linear approximation algorithm, a MATLAB simulation model is established, which is completely consistent with the hardware implementation of the proposed 7-digit DXP logarithmic converter. Furthermore, 100,000 7-digit DXP operands are simulated as test vectors in the MATLAB simulation model for this algorithm. The exact error of the decimal logarithm operation is demonstrated in Figure 4.3, and it is in the range of $-0.000998 \leq \varepsilon_t \leq 0.000988$. Thus, the final rounded outputs of $D'_i(4\text{-}digit) + c''_{1i}(4\text{-}digit)$ $(D_i = c'_{0i}(3\text{-}digit) \times m'(4\text{-}digit))$ can satisfy

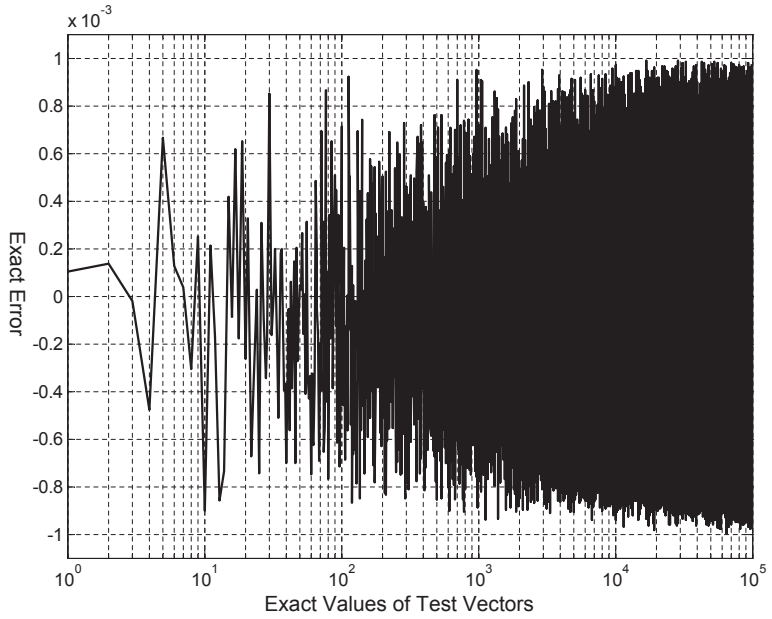

Figure 4.3: Exact error analysis of linear approximation of decimal logarithm.

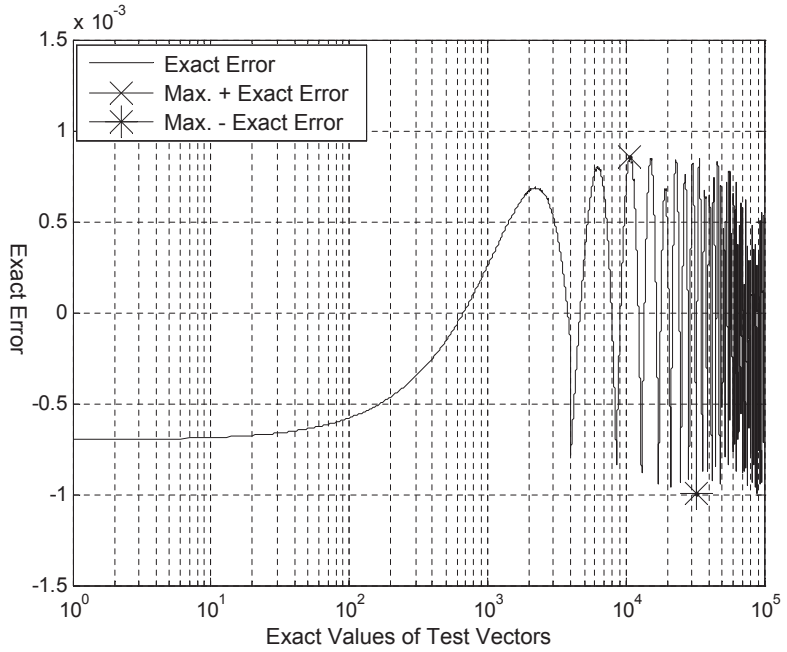

Figure 4.4: Exact error analysis of linear approximation of decimal antilogarithm.

the precision of 10^{-3} decimal logarithm results for any 7-digit DXP operand. The parameters shown in Table 4.1 are adopted for a design example in Section 4.5.7. Also, we analyze the tradeoff between the digit width of c'_{0i} and the number of segments in terms of the hardware implementation of the decimal logarithmic converter in Section 4.6.2.

4.3.5 Approximation Results for Decimal Antilogarithm

To compute the decimal antilogarithm operation, we evaluate the function 10^m on the interval $[0, 1)$ using the proposed dynamic non-uniform segmentation method. To minimize the complexity of the implementation of the antilogarithmic unit while keeping the precision of 10^{-3} accuracy of antilogarithm results, the function 10^m on the interval $[0, 1)$ is split into 45 segments to achieve the precision of 10^{-3} accuracy of antilogarithm results with the coefficients, with d_{0i} and d_{1i} being truncated to 5-digit respectively. The MATLAB simulation models based on decimal linear approximation algorithm is thus set up. It is completely consistent with a hardware implementation of a 7-digit DXP antilogarithmic converter. Furthermore, the 100,000 7-digit decimal logarithm results as test vectors are tested in the MATLAB simulation model based on this algorithm. The exact error of the decimal antilogarithm operation is demonstrated in Figure 4.4. The maximum error obtained by using this algorithm ranges over $-0.000999 \leq E_{absolute} \leq 0.000857$. Thus, the final rounded outputs of $D'_i(5\text{-}digit)+d''_{1i}(5\text{-}digit)$ $(D_i = d'_{0i}(5\text{-}digit) \times m'(5\text{-}digit))$ can satisfy the precision of 10^{-3} decimal antilogarithm results for any 7-digit DXP operand. The details of the optimized coefficients, the boundaries of each segment, and the maximum absolute errors of the linear approximation for a decimal antilogarithmic converter can be found in [117]. The hardware implementation of the decimal antilogarithmic converter is given in Section 4.5.3.

4.4 Error Analysis of Two Algorithms

Using Alg. 1 to operate a decimal logarithm computation, we consider a straightforward approach: 1) convert decimal operands to the binary format; 2) perform a binary-based decimal logarithm operation; and 3) convert binary-based decimal logarithm results back to the decimal format. The errors in this approach are produced in three ways. First, since the fractional part of most decimal operands can not be exactly represented by the finite width of binary numbers, the initial decimal-to-binary conversion generates a conversion error, $|\varepsilon_{c1}|$. Second, the binary-based decimal logarithm operation gives an computational error, $|\varepsilon_t|$. Third, the final binary-to-decimal conversion produces another conversion error,

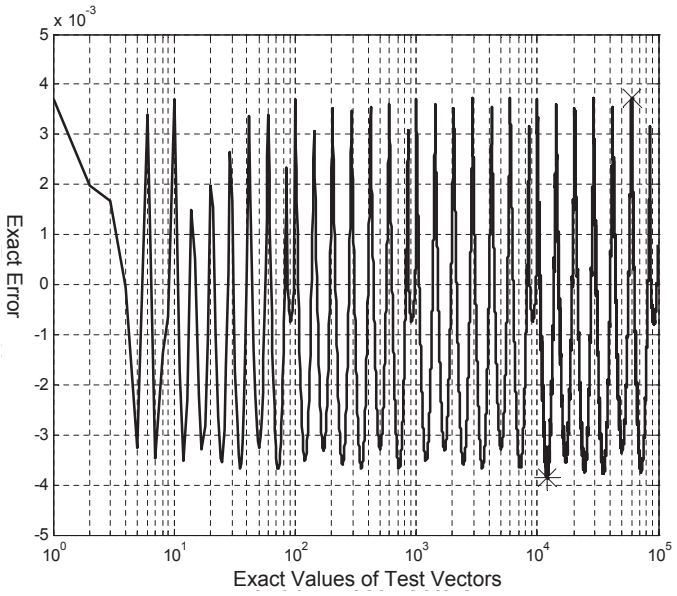

Figure 4.5: Exact error analysis of Alg. 2 for integer and fraction cases.

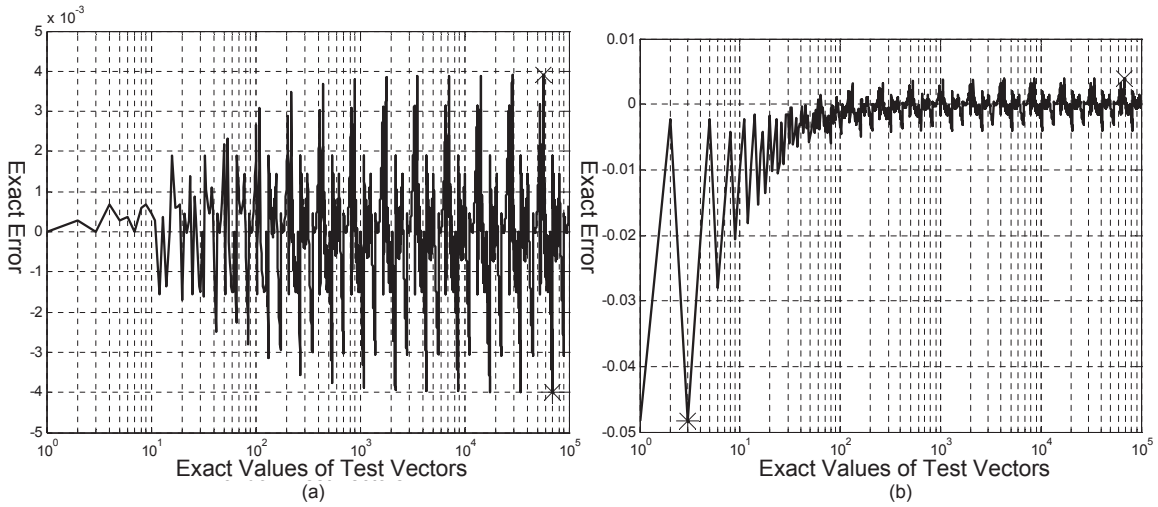

Figure 4.6: Exact error analysis of Alg. 1: a)integer case; b) fraction case.

$|\varepsilon_{c2}|$, because the binary fractional part of most decimal logarithm results also can not be exactly represented by the finite width of decimal numbers. However, compared with Alg. 1, the straightforward decimal logarithm conversion (Alg. 2) does not have any conversion error, $|\varepsilon_c|$, produced because it can directly compute decimal operands without the binary and decimal format conversion. To conduct a fair comparison, 1) we convert the 7-digit decimal non-normalized significand, N, to the 24-bit BXP number because they have similar dynamic ranges for the normalized mantissa ($2^{23} < 10^7 < 2^{24}$); 2) we constrain the same

Table 4.2: Comparsion of two decimal logarithm algorithms.

	Algorithm 2		**Algorithm 1**	
	case 1	**case 2**	**case 1**	**case 2**
No. of Segments	7		6	
Max +Exa. Error	0.367E-2		0.390E-2	0.390E-2
Max -Exa. Error	-0.388E-2		-0.398E-2	-0.483E-1
Exa. Error Range	0.755E-2		0.788E-2	0.522E-1

case 1: 7-digit decimal integer; case 2: 7-digit decimal fraction.

maximum absolute computational error, $|\varepsilon_t|$, for decimal logarithm conversions based on Alg. 1 and Alg. 2. Two MATLAB simulation models based on Alg. 1 and Alg. 2 are set up respectively. For Alg. 1, the conversion between 7-digit decimal operands and 24-bit binary numbers is realized based on the shift-and-add algorithm in [103]. A 24-bit binary logarithm conversion is implemented according to the 6-segment error correction algorithm described in [85] as a key computational component to compute binary-based decimal logarithm operation. For Alg. 2, since the maximum absolute computational error of the binary-based decimal logarithm conversion in Alg. 1 is equal to 0.00399, we evaluate the decimal logarithm function to keep the same maximum absolute computation error, $|\varepsilon_t|$, by the proposed dynamic non-uniform segmentation method for Alg. 2.

Thus, the decimal logarithm function, $1+\log_{10}(m)$, on the interval $[0.1, 1)$ is split into number of 7 segments. Then, 100,000 7-digit DXP operands as test vectors are tested in the MATLAB simulation model based on Alg. 2 for two cases (*case 1*: 7-digit integer; *case 2*: 7-digit fraction), while the corresponding 24-bit binary numbers, after being converted from 7-digit decimal operands in *case 1* and *case 2*, are tested in MATLAB simulation model based on Alg. 1. The exact errors of calculation of $1+log_{10}(m)$ based on Alg. 1 and Alg. 2 are demonstrated in Figure 4.5 and Figure 4.6 respectively, while the comparison results are given in Table 4.2. The comparison results indicate that 1) the maximum exact error resulted from Alg. 2, $-0.00388 \leq \varepsilon_t \leq 0.00367$, keeps the same for both integer and fraction cases; 2) the exact error range of the 7-segment Alg. 2 is constrained to 0.00755, which is similar to Alg. 1's error range of 0.00778 for *case 1*, but is much less than Alg. 1's error

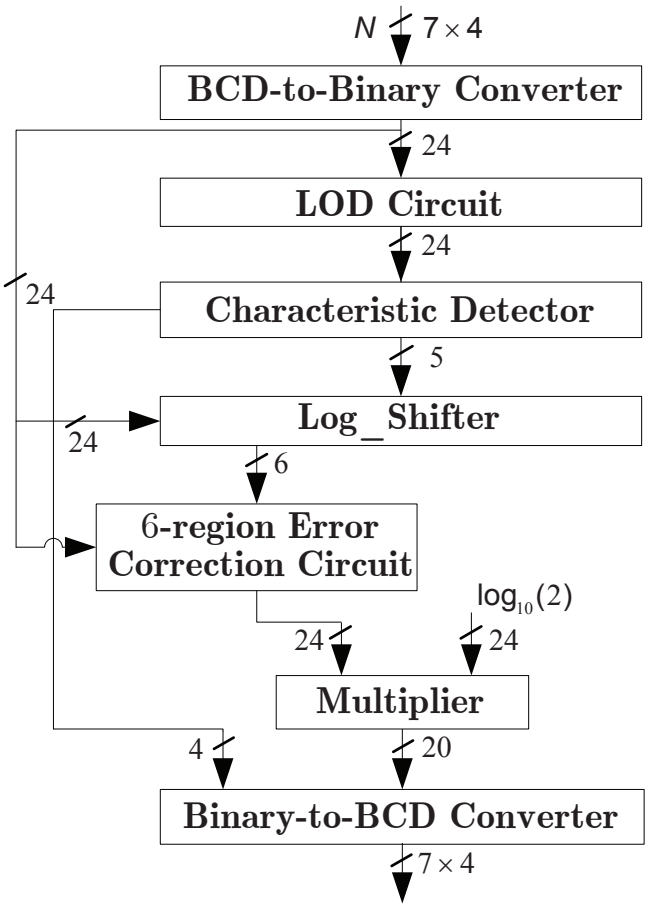

Figure 4.7: Block diagram of binary-based decimal logarithmic converter.

range of 0.0522 for *case 2* because the conversion error is unavoidable. Thus, it is shown that the proposed straightforward decimal logarithm conversion (Alg. 2) is error-free in the conversion between decimal and binary format.

4.5 Hardware Architecture

4.5.1 Binary-based Decimal Logarithmic Converter (Alg. 1)

Figure 4.7 shows the hardware architecture of the binary-based decimal logarithmic converter based on the binary-based decimal linear approximation (Alg. 1). The binary-based converter consists of four stages. First, a 7-digit non-normalized integers N is converted to a 24-bit BXP number by a combinational BCD-to-Binary converter. Second, the 24-bit BXP

number is computed in a 24-bit BXP logarithmic converter which is implemented based on the 6-region error correction algorithm described in [85]. This binary logarithmic converter is constructed by a leading one detector (LOD) circuit, a characteristic detector, a log_shifter, and a 6-region error correction circuit. For more details, we would refer the reader to [85]. Third, the 24-bit results of the 6-region error correction circuit are multiplied with a 24-bit constant $\log_{10}(2)$ to obtain decimal logarithm results, $(\log_{10}(dec))$. The decimal logarithm results obtained by the combinational multiplier are truncated to 20-bit, and then combined with 4-bit characteristic to achieve 24-bit binary-based decimal logarithm results, where the combinational multiplier is implemented by the multiplier IP core on FPGA. Fourth, the 24-bit results are converted back to decimal BCD representation by a combinational binary-to-BCD converter. The combinational BCD-to-binary and binary-to-BCD converters are implemented based on shift-and-add algorithms represented in [103]. This binary-based decimal logarithmic converter, as a benchmark, is implemented on FPGA to compare with the proposed decimal logarithmic converter in Section 4.6.4.

4.5.2 Decimal Logarithmic Converter (Alg. 2)

Figure 4.8 shows the hardware architecture of the proposed 7-digit decimal logarithmic converter based on the decimal linear approximation (Alg. 2). The hardware implementation of the proposed converter mainly consists of 1) a leading-zero detector (LZD) to compute the 1-digit decimal logarithm characteristic, k; 2) a decimal normalizer to shift non-normalized integers N to normalized DXP operands m; 3) a segment index encoder (SIE) to produce the segment index, i, according to r-digit most significant digits (MSDs) of m; 4) a coefficients look-up table to store the coefficients, q-digit c'_{0i} and p-digit c''_{1i}, in each segment; and 5) a decimal linear approximation unit (DLAU) to compute n-digit accurate approximation results of decimal logarithm mantissa. Thus, the accurate n+1-digit final decimal logarithm result, $\log_{10}(N)$, is achieved by combining 1-digit decimal logarithm characteristic and n-digit decimal logarithm mantissa in the final output register.

All signals in the architecture are represented with 10's complement number system. Each digit of positive DXP number is represented by 4-bit BCD code, whereas each digit of negative number is represented by its 10's complement format. The reason for choosing

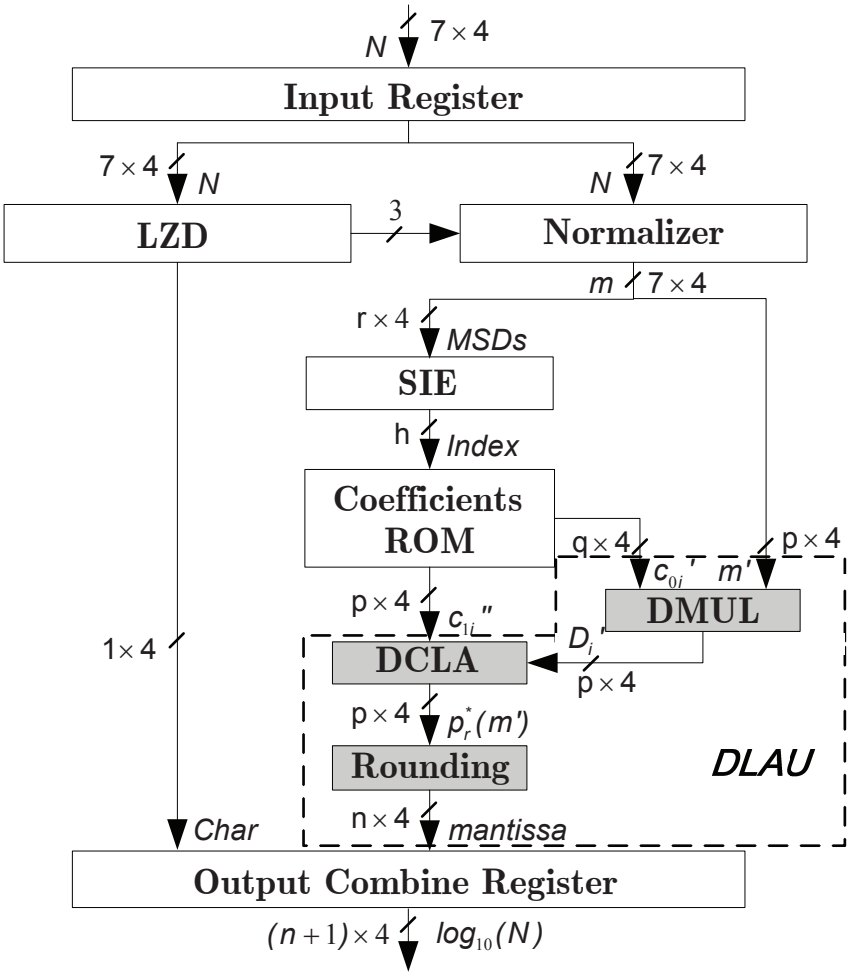

Figure 4.8: Block diagram of the proposed decimal logarithmic converter.

10's complement format is the decimal subtraction operation can be replaced by a decimal addition in 10's complement format, and all decimal digits, including the sign digit, can be operated in the decimal addition or subtraction.

4.5.3 Decimal Antilogarithmic Converter (Alg. 3)

Figure 4.9 shows the hardware architecture of the proposed 7-digit decimal antilogarithmic converter based on the decimal linear approximation. The hardware implementation of the proposed converter mainly consists of 1) a characteristic decoder that tells the shifter register how many digits should be shifted according to the characteristic part of logarithm results; 2) a coefficient look-up table that stores the coefficients, q-digit d'_{0i} and p-digit d''_{1i}, in each segment; 3) a decimal linear approximation unit (DLAU) that computes n-digit accurate

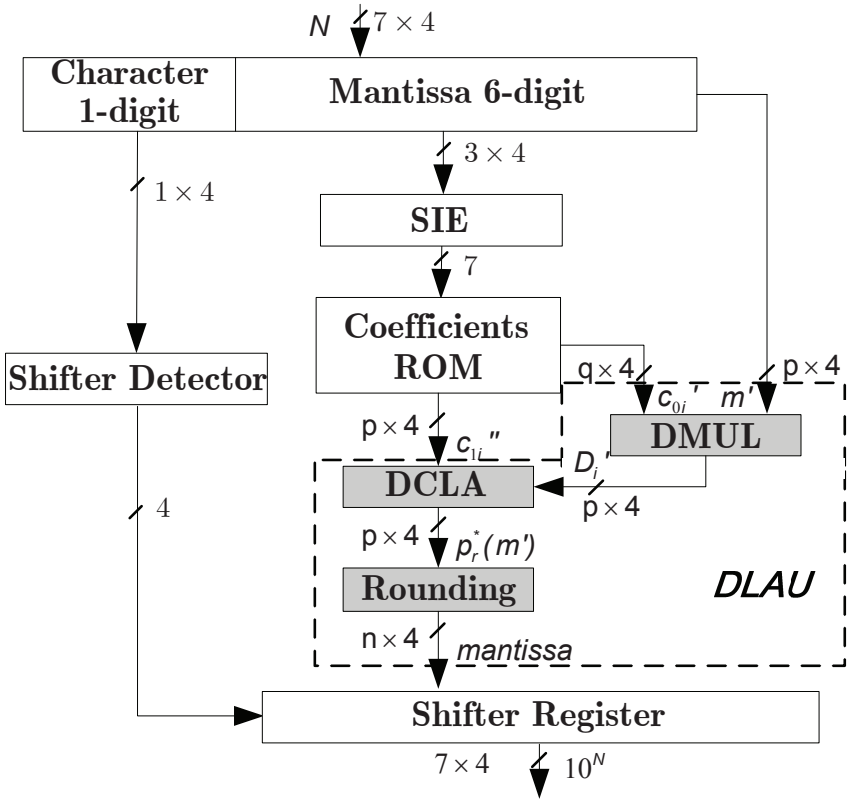

Figure 4.9: Block diagram of the proposed decimal antilogarithmic converter.

approximation results of decimal antilogarithm mantissa. 4) a right shifter that can shift the linear approximation result to the correct antilogarithm.

4.5.4 Decimal Segment Index Encoder

The most complicated unit in the proposed architecture is the SIE circuit, in particular when the decimal logarithm function is divided into large number of segments for a higher computational accuracy. The function of SIE is shown in:

$$SIE_{func}(m) : \{0,1,...,8,9\}^r \rightarrow \{0,1,...,i-1\} \tag{4.16}$$

In (4.16), the input of the SIE function is r-digit MSDs of 7-digit m, and the output is the segment index, i, a h-bit BXP number, applied to the address of the coefficients look-up table. The SIE circuit can be realized by the LUT cascade given in Figure 3.2. The LUT cascade is achieved by a functional decomposition based on the multiterminal decimal decision diagram (MTBDD) approach [110]. The architecture of the SIE circuit is a combinational logic in

which each LUT can be realized by the distributed memory on FPGA. In the leftmost LUT_1, all the inputs come from most significant bits (MSBs) of $r\times$4-bit x, x_{L_1}. For the other LUT_j, some inputs come from x, x_{L_j}, and the others come from the output of the previous LUT_{j-1} stage, $p_{L_{j-1}}$, which are called rails. The outputs of the rightmost LUT_j then represent h-bit segment index, i.

In [110], each LUT is restricted to h+2-bit inputs and h-bit outputs, where $h=\lceil\log_2(i)\rceil$, and i is the number of non-uniform segments. In this work, we adopt the same LUT cascade approach to evaluate the distributed memory size of the SIE circuit. Thus, the estimated memory size of the LUT cascade (ROM_1) is determined by the number of segment, i, and the digit-width of the segment boundary, $r\times$4-bit:

$$ROM_1=2^{\lceil\log_2(i)\rceil+1}\times\lceil\log_2(i)\rceil\times(4r-\lceil\log_2(i)\rceil) \tag{4.17}$$

Since it is impractical to restrict the same bit-width as the estimated LUT cascade in the actual implementation of the SIE circuit, we proposed an optimum decomposition approach to determine the actual LUT cascade strategy, using the minimum size of the distributed memory, and to implement the SIE circuit. Note that the actual distributed memory size is a bit larger than the estimated memory size.

4.5.5 Coefficients Look-up Table

In Figure 4.8, the coefficients look-up table has 2^h words, in which the coefficients, q-digit c'_{0i} and p-digit c''_{1i}, are stored respectively. Since the values of c'_{0i} are same in several adjacent segments when the digit-width of c'_{0i}, q-digit, is rounded to a relative small value, the memory size occupied by the coefficients look-up table can be reduced by only storing one c'_{0i} for these adjacent segments. In this work, we consider the maximum estimated memory size of the coefficients look-up table, (ROM_2), which is:

$$ROM_2=2^{\lceil\log_2(i)\rceil}\times(p+q)\times4 \tag{4.18}$$

Thus, the total estimated memory size (ROM_t) for implementing the coefficients look-up table and SIE circuit is:

$$ROM_t=2^{\lceil\log_2(i)\rceil+1}\times(4r\times\lceil\log_2(i)\rceil-\lceil\log_2(i)\rceil^2+2p+2q) \tag{4.19}$$

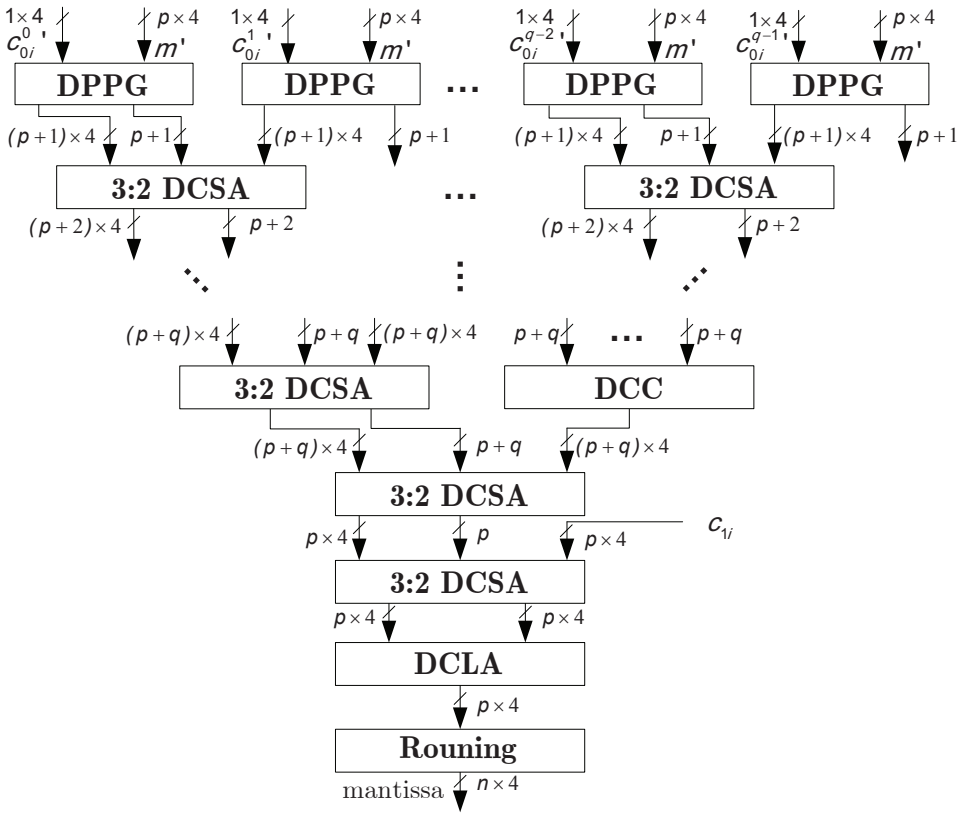

Figure 4.10: Details of decimal linear approximation unit.

4.5.6 Decimal Linear Approximation Unit

The hardware architecture of the DLAU circuit in the proposed decimal logarithmic and antilogarithmic converters is shown in Figure 4.8 (dotted line). It mainly consists of three units from the function view: 1) a DXP combinational multiplier[1] (DMUL) for computing p-digit $D'_i = truncate(c'_{0i} \times m')$; 2) a DXP carry-look-ahead adder (DCLA) for computing p-digit $p^*_r(m') = D'_i + c''_{1i}$; and 3) an output rounding unit for rounding p-digit $p^*_r(m')$ to the precision of 10^{-n} exact logarithm mantissas.

To decrease the carry propagation delay resulted from the decimal carry-propagating adder (DCPA) in the final level of DMUL, we replace this DCPA by a decimal carry-save adder (DCSA) in order to fuse DMUL and DCLA in the architecture of DLAU. The details of the DLAU architecture are shown in Figure 4.10, where $c_{0i}^{q-1\prime}$ represents the qth-digit of

[1]Note that when the digit-width of c'_{0i} is rounded to 1-digit, DMUL can be simplified to a decimal multiple logic.

c'_{0i}. Thus, the DLAU circuit mainly consists of five parts: 1) the decimal partial product generation (DPPG) units to multiply inputs m' by each digit of c'_{0i}; 2) a $\lceil \log_2(q) \rceil + 2$-level DCSA tree, comprised of the 3:2 DCSA counters and a decimal carry counter (DCC), to compute p-digit $p_r^*(m')$; 3) a p-digit DCLA to convert $p_r^*(m')$ from the decimal carry-save representation to the BCD representation; 4) a final rounding logic to obtain n-digit exact logarithm mantissas. The DPPG unit and DCSA tree are implemented based on a decimal combinational multiplier in [23]. The DCLA and DCSA are implemented based on the 1-digit DCLA adder described in [104]. The subtraction operations in the algorithm are carried out by this DCLA adder due to the 10's complement decimal format used in the architecture.

4.5.7 Design Example

Based on the evaluation results demonstrated in Table 4.1, we present a decimal logarithmic converter as a design example to compute the precision of 10^{-3} decimal logarithm results for any 7-digit DXP operand. The proposed design example can be implemented based on the block diagram of the proposed decimal logarithmic converter as shown in Figure 4.8. Since the decimal logarithm function is partitioned into $i = 20$ segments; and the digit-width of the segment boundary is $r = 3$-digit (refer to Table 4.1), we obtain a 4-level LUT cascade for the SIE circuit based on the optimum decomposition approach. According to (4.17), the estimated memory size of the LUT cascade for implementing the SIE circuit is obtained:

$$ROM_1 = 2^{5+1} \times 5 \times (12-5) = 2240 \ bits \tag{4.20}$$

The coefficients look-up table has 2^5 words, in each of which the coefficients, $q = 3$-digit c'_{0i} and $p = 4$-digit c''_{1i}, are stored respectively. According to (4.18), the estimated memory size of the coefficients look-up table is:

$$ROM_2 = 2^5 \times (3+4) \times 4 = 896 \ bits \tag{4.21}$$

Thus, the total estimated memory size of the decimal logarithmic converter is $ROM_t = 3136$ bits.

To compute the decimal linear approximation results in the DLAU circuit, first, 4-digit m' is multiplied by each digit of 3-digit coefficient c'_{0i} in the DPPG; second, 7-digit intermediate

Table 4.3: Details of combinational delay.

Reg	LZD	Norm	SIE	ROM	DLAU	Comb	**Total** (ns)
1.01	1.66	1.28	3.76	1.04	8.77	0.39	17.9

value, D_i, is computed in a 3-level DCSA tree; third, D_i is truncated to 4-digit D'_i, and then added with the 4-digit coefficient c''_{1i} to achieve the 4-digit $p^*_r(m')$ in another DCSA; fourth, 4-digit $p^*_r(m')$ in the decimal carry-save representation is converted to the BCD representation in a DCLA; fifth, 4-digit $p^*_r(m')$ is rounded to 3-digit accurate logarithm mantissa in the final rounding logic. Thus, the accurate 4-digit final decimal logarithm result, $\log_{10}(N)$, is obtained by combining 1-digit decimal logarithm characteristic and 3-digit decimal logarithm mantissa in the output register.

4.6 Experimental Results and Analysis

4.6.1 Implementation Results and Analysis

The proposed 7-digit DXP logarithmic converter, to compute the precision of 10^{-3} accurate logarithm results (refer to the design example), is modeled with VHDL and implemented on Virtex5 XC5VLX110T FPGA configuration [121]. The discussion about how we scale up the proposed converter to compute a higher precision of decimal logarithm results is presented in Section 4.6.3.

The proposed 7-digit DXP logarithmic converter is implemented only by a combinational architecture, which is firstly synthesized with XST, and then placed and routed by Xilinx ISE 11.3. The circuits of LZD, decimal normalizer, and DLAU in the proposed architecture are realized by look-up tables (LUTs) on FPGA, while the SIE circuit and coefficients look-up table are implemented by the distributed memory on FPGA. The implementation results on FPGA show that the proposed combinational architecture occupies 1 out of 32 GCLK I/O block, 46 out of 680 I/O blocks, and 384 out of 17,280 slices; and it takes a single clock cycle, running at 55.9 MHz, consuming 40 mW dynamic power, to achieve a precision of 10^{-3} logarithm result. The combinational delays of each block in the proposed architecture

are available in Table 4.3. It is evident that the two most time consuming blocks in the proposed architecture are SIE and DLAU circuits. The reasons are that 1) the SIE circuit is realized by a multi-level LUT cascade in which each level of LUT is implemented by a distribute memory on FPGA; 2) the DLAU circuit consists of a more complex DPPG, DCAS tree, DCLA adder and a final rounding block implemented based on BCD encoding.

The proposed architecture based on decimal linear approximation algorithm for DXP antilogarithmic converter is modeled in VHDL and implemented using the same Virtex5 XC5VLX110T FPGA device as that used for DXP logarithmic converter. The implementation results show that the proposed combinational architecture occupies 406 out of 17,280 slices; and it takes a single clock cycle, running at 59.3 *MHz* to achieve a precision of 10^{-3} antilogarithm result.

4.6.2 Tradeoff Analysis of Hardware Implementation

To reduce the combinational delay of the DLAU circuit in the proposed architecture, we keep the digit-width of m', c''_{1i} and D'_i as 4-digit and decrease the digit-width of c'_{0i} from 4-digit to less in order to adopt a faster less-level DCSA tree to compute the linear approximation, $p^*_r(m')$ of decimal logarithm results. However, when the digit-width of c'_{0i} is decreased, the decimal logarithm function has to be partitioned into more segments in order to keep the accuracy constraint. This result leads to a more-level LUT cascade and larger size of distribute memories occupied on FPGA for implementing the SIE circuit.

Table 4.4 shows the evaluation and hardware implementation results on Virtex5 XC5VLX 110T FPGA for the different digit-width of the coefficient c'_{0i}. The results indicate that 1) the proposed architecture, in which the digit-width of c'_{0i} is 2-digit, is 1.02 times faster and 1.47 times smaller than the architecture, and the digit-width of c'_{0i} is 4-digit; 2) it is 1.09 times faster and 1.21 times smaller than the architecture, in which the digit-width of c'_{0i} is 3-digit; 3) it is 1.05 times faster and 1.07 times smaller than the architecture, in which the digit-width of c'_{0i} is 1-digit; and 4) the dynamic power consumption is reduced with the decrease of the digit-width of c'_{0i}.

The reasons are that 1) since the number of level of DCSA tree in DLAU, which is equal to $\lceil \log_2(q) \rceil + 2$, is decreased accordingly when the digit-width of c'_{0i} is rounded from 4-digit

Table 4.4: Tradeoff analysis of hardware implementation.

Accuracy constraint	**Digit-width of q-digit c'_{0i}**			
precision of 10^{-3}	4-digit	3-digit	2-digit	1-digit
Segments (No.)	19	20	23	94
Actual memory size (Byte)	448	488	584	3,112
Occupied slices (No.)	490	384	317	338
Combinational delay (ns)	16.8	17.9	16.4	17.3
Power consumption (mW)	45	40	35	33

to 1-digit, the combinational delay and area of DLAU are reduced. However, 2) since the decimal logarithm function is partitioned into more segments, the number of level and size of the LUT cascade obtained by the optimum decomposition approach for implementing the SIE circuit, and the size of the coefficient look-up table for storing the coefficients is increased, which lead to an increase of the combinational delay and the distribute memory size. Note that the number of level of DCSA tree are 4, 4, 3 and 2; and the number of level of LUT cascades are 3, 4, 4 and 6 when the digit-width of c'_{0i} is rounded from 4-digit to 1-digit respectively.

4.6.3 Scale up to a Higher Required Accuracy

Since any accuracy constraint, 10^{-n}, can be set in the proposed non-uniform segmentation method for the decimal logarithm computation by Alg. 2, we can easily scale up the proposed architecture to achieve a higher fractional accuracy of logarithm results.

In this section, we analyze the function evaluation and hardware implementation results of the scaled-up architecture to achieve a higher precision of decimal logarithm results. During the decimal logarithm function evaluation by the proposed dynamic non-uniform segmentation method, first, we set the accuracy constraint as 10^{-n}, where the value of n is set as from 4 to 7 in order to achieve a higher fractional accuracy from the precision of 10^{-4} to 10^{-7} respectively; second, we constrain the digit-width of the rounded q-digit c'_{0i} as 4-digit, the digit-width of the rounded p-digit m', c'_{1i} and D'_i as $n+1$-digit. Note that since

Table 4.5: Implementation results in different accuracy constraints.

Accuracy constraint	10^{-4}	10^{-5}	10^{-6}	10^{-7}
Segments (No.)	66	204	662	2,310
Segments boundary (r-digit)	3	4	4	5
LUT cascade (No.)	3	5	5	6
Actual memory size (Byte)	3,136	7,264	17,120	127,584
Occupied slices (No.)	753	884	1,848	4,059
Combinational delay (ns)	20.1	25.6	29.9	35.7
Power consumption (mW)	51	55	73	112

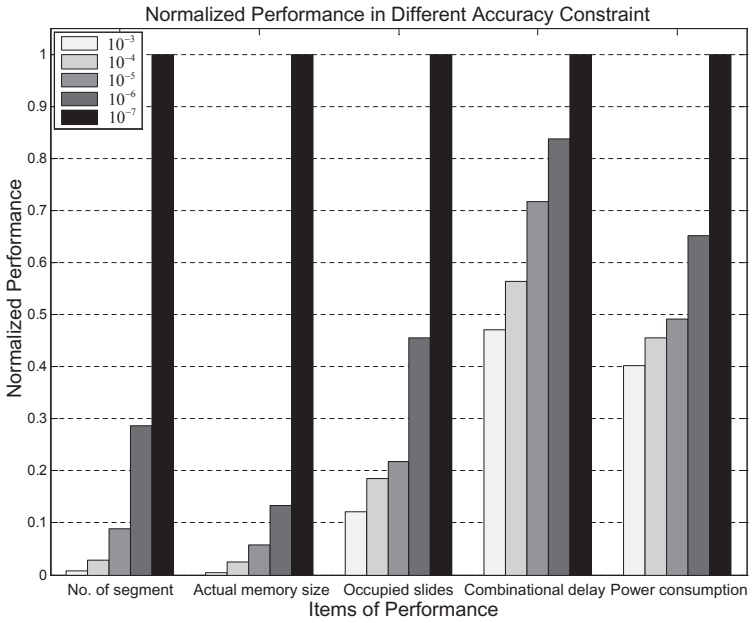

Figure 4.11: Normalized hardware performance in different accuracy constraints.

the full digit-width of the input m is 7-digit, the rounding error of m' is not considered when the accuracy constraint is set as 10^{-7}.

The scaled-up architectures of decimal logarithmic converters are modeled with VHDL and implemented on Virtex5 XC5VLX110T FPGA. Table 4.5 gives the evaluation and hardware implementation results of scaled-up architectures in the different accuracy constraints. Figure 4.11 shows a normalized hardware performance comparison. The results indicate that 1) the number of segment is increased with the more precision of the required fractional

Table 4.6: Hardware performance comparison in two algorithms.

Approaches	**Algorithm 1**	**Algorithm 2**
Segments (No.)	6	7
Occupied slices (No.)	300	343
Max. frequency (MHz)	29.4	63.3
Clock cycles (No.)	1	1
Latency (ns)	34.0	15.8
Power consumption (mW)	25.8	36.4

accuracy, which leads to the increase of actual memory size occupied by the SIE circuit and coefficient look-up table; 2) the number of level of LUT cascade obtained by the optimum decomposition approach [110] is increased with more segments and the digit-width of the segment boundary, which leads to an increment of the combinational delay of the SIE circuit; 3) since the digit-width of the rounded p-digit m', c'_{1i} and D'_i is increased 1-digit with one more precision of fractional accuracy, which lead to a more complex architecture of the DLAU so that its combinational delay, area and dynamic power consumption are increased accordingly. Therefore, with the increasing precision of the required accuracy, the memory size occupied by the SIE circuit and the coefficient look-up table could become too large to be implemented by the limited resources on FPGA.

4.6.4 Hardware Performance Comparison in Two Algorithms

For further analysis, we compare the hardware performance of the proposed architecture (Alg. 2) and the binary-based architecture (Alg. 1), which is described in Section 4.5.1. In order to conduct a fair comparison, we scale up the two architectures to compute with the same maximum absolute computation error, then synthesize, place and route them using the same tools on Virtex5 XC5VLX110T FPGA. Table 4.6 shows hardware performances of the two architectures. The comparison results indicate that the proposed architecture based on the decimal linear approximation algorithm (Alg. 2) only occupies 1.14 times more slices, and consumes 1.41 times more dynamic power, but runs 2.15 times faster than the binary-

based decimal logarithmic converter. The reasons lie in that 1) the form of BCD encoding used in the proposed architecture (Alg .2) needs more resource to be implemented compared with binary format used in the binary-based architecture (Alg. 1); 2) the proposed decimal logarithmic converter (Alg. 2) can directly compute the decimal logarithm results, and that Alg. 2's architecture needs neither combinational BCD-to-binary nor binary-to-BCD converters, the factors that slow down the performance of Alg. 1.

4.7 Summary

In this chapter, we present a new approach to compute the decimal logarithm and antilogarithm function based on the decimal first-order polynomial approximation in hardware. This new approach is error-free in the conversion between decimal and binary format, and can be scaled up to compute any accuracy of decimal logarithm and antilogarithm results in the BCD encoding. Moreover, we analyze the tradeoff of the hardware performance in order to determine the most suitable digit-width of coefficients so that the proposed architecture can achieve the smallest area and fastest speed. Finally, we make a hardware performance comparison between the two algorithms, and its results show that the proposed decimal logarithmic converter (Alg. 2) runs significantly faster than the binary-based decimal logarithmic converter (Alg. 1). As long as enough distributed memory is available on FPGA, the precision of accuracy can be increased by using the proposed architecture that has the potential of achieving a higher speed and throughput on the decimal logarithm computation.

Part IV

Digit-Recurrence with Selection by Rounding

Chapter 5

Decimal Floating-Point Logarithmic Converter

In this chapter, we present the algorithm and architecture of the decimal floating-point (DFP) logarithmic converter, based on digit-recurrence algorithm with selection by rounding. The proposed converter can compute faithful DFP logarithm results for any one of the three DFP formats, which are specified in the IEEE 754-2008 standard. In order to optimize the latency for the proposed design, we integrate the following novel features: 1) using signed-digit redundant digits, and redundant carry-save representation of the data-path; 2) reducing the number of iterations by determining the number of initial iteration; and 3) retiming and balancing the delay of the proposed architecture. The proposed architecture is synthesized with STM 90-nm standard cell library, and results show that the critical path delay and latency of the proposed Decimal64 logarithmic converter are 1.55 *ns* (34.4 FO4) and 19 clock cycles respectively; and the total hardware complexity is 43572 NAND2 gates. The rough delay estimation results of the proposed architecture show that its latency is close to that of the binary radix-16 logarithmic converter, and that it has a significant decrease on latency in contrast to a recently published high performance CORDIC implementation.

5.1 Introduction

In Chapter 4, a low-accuracy DXP logarithmic converter, based on linear approximation algorithm, is described. Although the logarithmic converter based on linear approximation method is fast and consumes low power, it will produce some errors that lead to inaccurate computation results. The hardware-oriented algorithms based on digit-recurrence with selection by rounding are introduced for high-radix binary division, square-root [89, 90, 91], CORDIC [93], logarithm [94] and exponential [92] operations respectively. This method can

efficiently decrease the cost of implementation, and the complexities of the selection function for redundant digits in particular. In this chapter, a radix-10 digit-recurrence algorithm based on the selection by rounding approach is presented to implement the DFP logarithmic converter in order to achieve faithful logarithm results of DFP operands, specified in IEEE 754-2008 standard. The design described in this chapter is an improved work based on our previous research [122], and includes the following new features:

1. signed-digit redundant digits;
2. redundant carry-save representation of the data-path;
3. reducing the number of iterations by determining the number of initial iteration;
4. selecting redundant digits by rounding estimated residuals;
5. retiming and balance the delay of the proposed architecture;
6. novel implementations of operation blocks in the carry-save data-path;
7. normalization, parallel final addition and rounding operation to display DFP logarithm results.

This design makes the first attempt to analyze and implement a DFP logarithmic converter that can compute the DFP based-10 logarithm operation specified in the IEEE 754-2008 standard.

This chapter is organized as follows: In Section 5.2 we give an overview of the DFP logarithm operation. Section 5.3 presents the proposed algorithm and error analysis for a DFP logarithm computation. Section 5.4 describes the improved architecture based on the redundant data-path with details of its hardware implementation. In Section 5.5, first, we analyze the area-delay evaluation results of the proposed architecture; second, we compare the performance of the proposed design with the binary radix-16 converter [94], our original design [122], the recent decimal CORDIC design [49], and the software implementation [41] in terms of the latency; third, we discuss the various characters of the DFP logarithmic converter for three different DFP formats. Section 5.6 gives conclusions. In Chapter 5 and Chapter 6, for details about the DFP operands and rounding for decimal based-10 logarithm operation, please refer to Section 2.1.1 and Section 2.2.1.

5.2 DFP Logarithm Operation

5.2.1 Exception Handling

A valid DFP logarithm operation is defined as:

$$R = \log_{10}(v) = \log_{10}(10^e) + \log_{10}(\mathit{significand}). \tag{5.1}$$

In (5.1), the exponent is in the range of $(emin-q+1) \leq e \leq (emax-q+1)$, and the decimal significand is a q-digit non-normalized integer in the range of $1 \leq \mathit{significand} \leq 10^q-1$. There are some exceptional cases that need to be dealt with during a DFP logarithm operation:

- v must be a positive DFP operand ($S = 0$), otherwise the DFP logarithm operation simply returns NaN and signals the invalid operation exception;
- if v is a NaN, the DFP logarithm operation returns NaN and signal the invalid operation exception;
- if v is ± 0, the DFP logarithm operation returns $-\infty$ and signals the divideByZero exception;
- if the DFP logarithm result is inexact, the DFP logarithm operation signals the inexact exception;
- if v is $+\infty$ or $+1$, the DFP logarithm operation returns $+\infty$ or $+0$ respectively with no exception;
- since all of valid DFP logarithm results are obtained in the representable range of the DFP numbers, the overflow, underflow and subnormal exceptions are not produced during the DFP logarithm operation.

5.2.2 Range Reduction

The computation of $\log_{10}(\mathit{significand})$ is a q-digit DXP logarithm operation. Since the decimal significand of DFP operand is defined as a non-normalized integer, it should be normalized into the range of $[0.1, 1)$ before it is computed by the proposed algorithm. As a result, (5.2) is obtained:

$$R = \log_{10}(v) = e + k + \log_{10}(m). \tag{5.2}$$

In (5.2), k is the number of significant digits of the decimal significand, and equal to q less the number of leading zeros of the decimal significand, where $1 \leq k \leq q$; m is a normalized decimal fractional number that consists of the k-digit significant digits of the decimal significand followed by $(q-k)$-digit zeros, where $0.1 \leq m < 1$. After range reduction, the logarithm operation, $\log_{10}(m)$, may produce at most q-digit leading zeros or $q-1$-digit leading nines, when the value of m is very close to 1 or 0.1. Since the target is a DFP logarithm operation, the DFP logarithm operation, $\log_{10}(v)$, should be capable of achieving a q-digit decimal significand without leading zeros.

A straightforward approach, similar to [123], is required to guarantee at least a twice precision of $2q$-digit DXP logarithm results at first, and then the normalization can be performed by shifting the leading zeros of $2q$-digit results to the left to obtain q-digit results without leading zeros, which then represent the decimal significand of DFP logarithm results. However, this approach makes the computational delay extremely large, since it doubles the number of iterations and the data-path width. To avoid the computation of the leading zeros for DFP logarithm results, we use a similar approach as [49], which start the first iteration with the index $j = j_{init}$ (refer to Section 5.3.3), and only compute the $q+1$-digit decimal logarithm result $\log_{10}(m)$ with at most 1-digit leading zero in the number of $q+1$ iterations. For example, the proposed approach can achieve a q+1-digit result with the $2q$-digit accuracy from the number of q^{th} to $2q^{th}$ iteration for an operand, m, whose fractional digits are all equal to nine.

5.3 Digit-Recurrence Algorithm for Logarithm

5.3.1 Overview

A digit-recurrence algorithm to calculate $\log_{10}(m)$ is derived based on [94], which is summarized as follows:

$$\log_{10}(m) = \log_{10}(m \prod f_j) - \sum \log_{10}(f_j) \tag{5.3}$$

where $0.1 \leq m < 1$. If the following condition is satisfied:

$$\lim_{j\to\infty}\{m\prod_{j=1}^{j} f_j\} \to 1 \tag{5.4}$$

Then

$$\lim_{j\to\infty}\{\log_{10}(m\prod_{j=1}^{j} f_j)\} \to 0 \tag{5.5}$$

Thus,

$$\log_{10}(m) = 0 - \sum_{j=1}^{\infty}\log_{10}(f_j) \tag{5.6}$$

f_j is defined as $f_j = 1+e_j 10^{-j}$ by which m is transformed to 1 through successive multiplications. This form of f_j allows the use of a decimal *shift-and-add* implementation.

The corresponding recurrences for transforming m and computing the logarithm are presented in (5.7) and (5.8), where $j \geq 1$, $E[1] = m$ and $L[1] = 0$.

$$E[j+1] = E[j](1 + e_j 10^{-j}) \tag{5.7}$$

$$L[j+1] = L[j] - \log_{10}(1 + e_j 10^{-j}) \tag{5.8}$$

The digits e_j are selected so that $E(j+1)$ converges to 1. A 1-digit accuracy of the calculation result is, therefore, obtained in each iteration. After performing the last iteration of recurrence, the results are:

$$E[N+1] \approx 1 \tag{5.9}$$

$$L[N+1] \approx \log_{10}(m) \tag{5.10}$$

Although the proposed algorithm is designed for the computation of DXP logarithm results in base 10, the logarithm results in any base β can also be achieved by simply changing the base '10' to 'β' as shown in (5.11):

$$L[j+1] = L[j] - \log_{\beta}(1 + e_j 10^{-j}) \tag{5.11}$$

Thus, after performing the last iteration of recurrence, the final results are transformed to:

$$L[N+1] \approx \log_{\beta}(m) \tag{5.12}$$

To have the selection function for e_j dependent on the same digit positions in all iterations, a scaled remainder is defined as:

$$W[j] = 10^j(1 - E[j]) \tag{5.13}$$

Thus,

$$E[j] = 1 - W[j]10^{-j} \tag{5.14}$$

To substitute (5.14) in (5.7), the recurrence on E is replaced by the residual recurrence:

$$W[j+1] = 10(W[j] - e_j + e_j W[j]10^{-j}) \tag{5.15}$$

According to (5.15), the digits e_j are selected as a function of leading digits of the scaled residual in a way that the residual $W[j]$ remains bounded.

5.3.2 Selection by Rounding

The selection of the digits e_j is achieved by rounding to the integer part of the scaled residual. To reduce the delay of selection function, the rounding is performed on an estimated $\widehat{W}[j]$, which is obtained by truncating $W[j]$ to t fractional digits (truncating $W[j]$ at the position 10^{-t}). The selection function is indicated as:

$$e_j = round(\widehat{W}[j]) \tag{5.16}$$

In (15), *round* indicates that if the digit of $\widehat{W[j]}$ at the position 10^{-1} is larger than or equal to 5, the digit e_j is obtained by adding the integer part of $\widehat{W[j]}$ and 1; otherwise it is directly obtained by the integer part of $\widehat{W[j]}$. In this work, the selection by rounding is performed with the maximum redundant set $e_j \in \{-9, -8, ..., 0, ...8, 9\}$. Since $|e_j| \leq 9$,

$$-9.5 < \widehat{W}[j+1] < 9.5 \tag{5.17}$$

thus,

$$-9.5 - 10^{-t} < W[j+1] < 9.5 + 10^{-t} \tag{5.18}$$

and,

$$-0.5 - 10^{-t} < W[j] - e_j < 0.5 + 10^{-t} \tag{5.19}$$

equation (5.15) can be represented as:

$$W[j+1] = 10(W[j] - e_j) + e_j 10^{1-j}(W[j] - e_j + e_j) \tag{5.20}$$

According to (5.18), (5.19) and (5.20), the numerical analysis is processed as follows:

$$-5 - 10^{-t+1} + e_j 10^{1-j}(-0.5 - 10^{-t} + e_j) > -9.5 - 10^{-t} \tag{5.21}$$

$$5 + 10^{-t+1} + e_j 10^{1-j}(0.5 + 10^{-t} + e_j) < 9.5 + 10^{-t} \tag{5.22}$$

The results in numerical analysis show that if and only if $j \geq 3$, $t \geq 1$ the conditions (5.21) and (5.22) are satisfied. In doing so, the selection by rounding is only valid for iterations $j \geq 3$, and e_1 and e_2 can be only achieved by look-up tables. However, using two look-up tables for $j = 1, 2$ will significantly increase the overall hardware implementations. Therefore, the restriction for e_1 is defined so that e_2 can be achieved by selection by rounding and one look-up table will be saved. Because $W[1] = 10(m-1)$, $W[2]$ can be achieved as:

$$W[2] = 100 - 100 \times m - 10e_1 \times m \tag{5.23}$$

When the value of j equates to 2, the value of e_2 is in the range of $-6 < e_2 < 6$ so that (5.21) and (5.22) are satisfied. Substituting $-6 < e_2 < 6$, and $t = 1$ in (5.19) yields:

$$-5.6 < W[2] < 5.6 \tag{5.24}$$

According to (5.23) and (5.24), we obtain:

$$100 - 100 \times m - 10e_1 \times m > -5.6 \tag{5.25}$$

$$100 - 100 \times m - 10e_1 \times m < 5.6 \tag{5.26}$$

The results in numerical analysis of (5.25) and (5.26) show that 1) the decimal input operand m is restricted in the range of $0.5 \leq m' \leq 1.05$ so that e_2 can be achieved by selection by rounding; 2) 3-digit MSDs of the input operand m are necessary to address the initial look-up table for selection of the digit e_1. The look-up table I for selection of e_1 is shown in Table 5.1. Since the value of m is in the range of $0.1 \leq m < 1$, if the input operand is in the range of $0.1 \leq m < 0.5$, it is necessary to be scaled to the range of $0.525 \leq m' < 1.05$ by multiplying with 2, 4, 5 or 10 for the proper recurrence; otherwise, it keeps the same value.

Table 5.1: Digit e_1 selection of DFP logarithm.

The range of m'	**e_1 (BCD)**
$[0.95, 1.05)$	0(0000)
$[0.86, 0.95)$	1(0001)
$[0.79, 0.86)$	2(0010)
$[0.73, 0.79)$	3(0011)
$[0.68, 0.73)$	4(0100)
$[0.63, 0.68)$	5(0101)
$[0.60, 0.63)$	6(0110)
$[0.56, 0.60)$	7(0111)
$[0.53, 0.56)$	8(1000)
$[0.50, 0.53)$	9(1001)

The input operand, in the range of $0.1 \leq m < 0.105$, is scaled by multiplying with 10 instead of 5 because scaling by 5 would not produce $\log_{10}(m')$ with any leading nines that only happens after subtracting $\log_{10}(5)$ to obtain the final result. To avoid the recurrence of these leading nines of the final result in the proposed algorithm, the operand $(0.1 \leq m < 0.105)$ needs to be scaled to the range of $1 \leq m' < 1.05$. The scaled numbers m' which are in the range of $0.5 \leq m' < 1.05$ are computed by the proposed algorithm. Then, the final logarithm result of $\log_{10}(m)$ is achieved by subtracting $\log_{10}(m')$ with the scaled constant (0, 1, $\log_{10}(2)$, $\log_{10}(4)$ or $\log_{10}(5)$).

5.3.3 Index of Initial Iteration

When the value of $e+k$ is equal to zero or one, the q-digit decimal logarithm operation, $\log_{10}(m')$, is required to compute the decimal logarithm result with at most $2q$-digit accuracy, so that the q-digit decimal significand without leading zeros for a DFP logarithm operation can be achieved. To avoid the computation of leading zeros, the proposed algorithm starts the first iteration with the index $j = j_{init}$ so that the value of the digit $e_{j_{init}}$, obtained in the first iteration, is not zero. The index of the initial iteration, $j = j_{init}$, is determined based on the value of the q-digit scaled operand m' as shown in Table 5.2. If the initial iteration

Table 5.2: Selection of Index j_{init}.

The range of q-digit m'	j_{init}
$[0.5000...000, 0.9500...000]$	1
$[0.9500...001, 0.9950...000]$	2
...	...
$[0.9999...951, 0.9999...995]$	$q-1$
$[0.9999...996, 1.0000...004]$	q
$[1.0000...005, 1.0000...049]$	$q-1$
...	...
$[1.0005...000, 1.0049...999]$	3
$[1.0050...000, 1.0499...999]$	2

is not started from the first iteration, $j_{init} \neq 1$, the digit $e_{j_{init}}$ is obtained by rounding the estimated value of $W[j_{init}] = 10^{j_{init}} \times (1-m')$; if the initial iteration is started from the first iteration, $j_{init} = 1$, the digit $e_{j_{init}}$ is obtained from *look-up table I*. The corresponding $L[j_{init}]$, $\log_{10}(1+e_{j_{init}}10^{-j_{init}})$ without $j_{init}-1$-digit leading zeros, as the first item of $L[j]$, is selected from the look-up table II to achieve the q-digit decimal signifincand of DFP logarithm results. Since the value of the digit $e_{j_{init}}$ may equal to ± 1 or ± 2, the corresponding $L[j_{init}]$, may still include at most 1-digit leading zero after shifting out $j_{init}-1$-digit leading zeros. Thus, to avoid this 1-digit leading zero, the proposed algorithm needs to operate at least $q+1$ iterations to obtain a q-digit decimal significand for a DFP logarithm operation.

5.3.4 Approximation of Logarithm

The logarithm result can be achieved by accumulating the values of $-\log_{10}(1+e_j 10^{-j})$ in each iteration, and these values are stored in the look-up table II. With the increasing number of iteration, however, the size of the table will become prohibitively larger and larger. Therefore, there is a need for a method that can reduce the table size and achieve a significant reduction in the overall hardware requirement. A Taylor series expansion of the logarithm function $\log_{10}(1+x)$ is demonstrated in (5.27):

$$\log_{10}(1+x) = (x - \frac{x^2}{2} + ...)/\ln(10) \tag{5.27}$$

After $j=h$ iterations, the values of $\log_{10}(1+e_j10^{-j})$ can be approximated by $e_j10^{-j}/\ln(10)$. Since an at most $2q$-digit accuracy needs to be guaranteed for achieving the q-digit decimal significand of DFP logarithm results, the series approximation can be used in the iterations when the following constraint is satisfied,

$$\frac{e_j^2 10^{-2j}}{2\ln(10)} < 10^{-(2q+2)} \tag{5.28}$$

Considering the case ($e_j=9$ or -9), the numerical analysis of (5.28) shows that:

$$h=\lfloor q+1.6 \rfloor = q+1 \tag{5.29}$$

Therefore, after $h = q+1$ iterations, while the values of $-\log_{10}(1+e_j10^{-j})$ do not need to be stored in the look-up table II, the values of $-e_j10^{-j}/\ln(10)$, instead, will be used for approximation. Note that the values of h are equal to 8, 17 and 35 for Decimal32, Decimal64 and Decimal128 formats respectively in the proposed logarithm digit-recurrence algorithm.

5.3.5 Error Analysis and Evaluation

The errors in the proposed algorithm can be produced in four ways. The first type of error is the inherent error ε_i resulted from the difference between the logarithm results obtained from finite iterations and the exact results obtained from infinite iterations. The second one is the approximation error ε_a produced by approximating the values of $-\log_{10}(1+e_j10^{-j})$ with the values of $-e_j10^{-j}/\ln(10)$. The third one is the quantization error ε_q generated from the finite precision of the intermediate values in the hardware implementation. The fourth one is the final rounding error ε_r, whose maximum value is 0.5 ulp ($|\varepsilon_r| \leq 0.5\times10^{-2q}$). In order to achieve a q-digit decimal significand of the faithful DFP logarithm result, we analyze the maximum absolute error in terms of the worst case that requires $2q$-digit accuracy to be guaranteed by the DXP logarithm operation from the q^{th} to the $2q^{th}$ iteration. Thus, the following condition must be satisfied:

$$|\varepsilon_t| = |\varepsilon_i| + |\varepsilon_a| + |\varepsilon_q| \leq 0.5 \times 10^{-2q} \tag{5.30}$$

Inherent Error

Since the DXP logarithm result of the worst case is achieved after the $2q^{th}$ iteration, ε_i can be defined as:

$$\varepsilon_i = -\sum_{j=2q+1}^{\infty} \log_{10}(1+e_j 10^{-j}) \tag{5.31}$$

In order to use the static error analysis method, we choose the cases ($e_j = 9$ or -9) to analyze the maximum ε_i:

$$\varepsilon_i = -\sum_{j=2q+1}^{\infty} \log_{10}(1 \pm 9 \times 10^{-j}) \tag{5.32}$$

Using Taylor series expansion of (5.27), we obtain:

$$\varepsilon_i < \pm 9 \times (10^{-2q-1} + 10^{-2q-2} + ...)/\ln(10) \tag{5.33}$$

Then, the absolute maximum ε_i is in the range:

$$|\varepsilon_i| \leq 4.34 \times 10^{-2q-1} \tag{5.34}$$

Approximation Error

We use an approximated value, $e_j 10^{-j}/\ln 10$, to estimate $\log_{10}(1+e_j 10^{-j})$ from $q+2^{nd}$ to $2q^{th}$ iteration. According to the series expansion of logarithm function in (5.27), it produces an approximation error, ε_a:

$$\varepsilon_a = \sum_{j=q+2}^{2q} (-\frac{(e_j 10^{-j})^2}{2} + \frac{(e_j 10^{-j})^3}{3} - ...)/\ln(10) \tag{5.35}$$

Since

$$\sum_{j=q+2}^{2q} (\frac{(e_j 10^{-j})^3}{3} - ...)/\ln(10) \ll 10^{-2q-3} \tag{5.36}$$

we keep $-(e_j 10^{-j})^2/2\ln(10)$ to analyze ε_a:

$$\varepsilon_a \leq \sum_{j=q+2}^{2q} (-\frac{(e_j 10^{-j})^2}{2})/\ln(10) \tag{5.37}$$

Considering the case ($e_j = 9$ or -9) in (5.37), we obtain the maximum ε_a:

$$|\varepsilon_a| \leq 1.74 \times 10^{-2q-3} \tag{5.38}$$

Quantization Error

Since only those intermediate values that have finite precision are operated in the hardware implementation, three quantization errors occur. In this work, we define *FDs*-digit as the minimal data-width of fractional digits for each of the intermediate values. First, since logarithm results are achieved by accumulating *FDs*-digit rounded values of $-\log_{10}(1+e_j10^{-j})$ from the q^{th} to the $(q+1)^{th}$ iteration, the maximum rounding error of $-\log_{10}(1+e_j10^{-j})$ is $\pm0.5\times10^{-FDs}$ in each iteration, and the maximum ε_{q1} is:

$$|\varepsilon_{q1}| \leq \sum_{j=q}^{q+1} 0.5\times10^{-FDs} = 1\times10^{-FDs} \tag{5.39}$$

Second, the logarithm results are achieved by accumulating *FDs*-digit truncated values of $-e_j10^{-j}/\ln(10)$ from $q+2^{nd}$ to $2q^{th}$ iteration. *FDs*-digit rounded values $e_j/\ln(10)$ are stored in a look-up table, so the maximum quantization error of the value $e_j/\ln(10)$ is $\pm0.5\times10^{-FDs}$. When $e_j=9$ or -9, the maximum ε^1_{q2} is:

$$\left|\varepsilon^1_{q2}\right| \leq \sum_{j=q+2}^{2q} \pm9\times10^{-j}\times0.5\times10^{-FDs} \ll 10^{-FDs} \tag{5.40}$$

Another quantization error, ε^2_{q2}, is produced by the finite *FDs*-digit precision when truncating the value of $-e_j10^{-j}/\ln(10)$. In each iteration, the maximum truncating error of $-e_j10^{-j}/\ln(10)$ is $\pm1\times10^{-FDs}$, so the maximum ε^2_{q2} is:

$$\left|\varepsilon^2_{q2}\right| \leq \sum_{j=q+2}^{2q} 1\times10^{-FDs} = (q-1)\times10^{-FDs} \tag{5.41}$$

Third, the logarithm result $\log_{10}(m')$ is adjusted by a finite *FDs*-digit rounded scaled constant (0, $\log_{10}(2)$, $\log_{10}(3)$ or $\log_{10}(5)$) in the last iteration, so the quantization error, ε_{q3}, occurs. The maximum ε_{q3} is:

$$|\varepsilon_{q3}| \leq 0.5\times10^{-FDs} \tag{5.42}$$

Therefore, the maximum quantization error, ε_q, is:

$$|\varepsilon_q| \leq |\varepsilon_{q1}| + \left|\varepsilon^1_{q2}\right| + \left|\varepsilon^2_{q2}\right| + |\varepsilon_{q3}| \approx (q+0.5)\times10^{-FDs} \tag{5.43}$$

Table 5.3: Error Analysis for DFP Interchange Formats

Format Names	**Decimal32**	**Decimal64**	**Decimal128**
Significand (q-digit)	7	16	34
Num. of Iteration (q+1)	8	17	35
Accuracy ($2q$-digit)	14	32	68
FD-digit ($2q$+3-digit)	17	35	71
Max. Error ($\lvert\varepsilon_t\rvert \times 10^{-2q}$)	0.444	0.453	0.471

Error Evaluation

Since the DXP logarithm result is required to guarantee at least $2q$-digit accuracy, having ε_i, ε_a, ε_q obtained in (5.34), (5.38) and (5.43) respectively, we achieve the maximum total error ε_t as:

$$|\varepsilon_t| = |\varepsilon_i| + |\varepsilon_a| + |\varepsilon_q| \leq 0.436 \times 10^{-2q} + (q+0.5) \times 10^{-FD} \tag{5.44}$$

We substitute the digit-width of the decimal significand of the three DFP formats, $q=7$, 16 and 34, into (5.44) respectively. The results indicate that the maximum absolute errors $|\varepsilon_t|$ obtained in the three DFP formats are smaller than 0.5 ulp, which can satisfy the condition (5.30). Thus, the final rounded results are smaller than accuracy requirement within 1 ulp after considering the final rounding error. Table 5.3 shows the error analysis of the worst case for three different DFP interchange formats. The error analysis in Table 5.3 proves that only when the minimal data-width of the fractional digits for each intermediate value (*FD*-digit) is larger than or equal to $2q$+3-digit, the proposed algorithm can guarantee at most $2q$-digit accuracy for the DXP logarithm operation, and therefore a q-digit decimal significand of the faithful DFP logarithm result can be achieved.

5.3.6 Guard Digit of Scaled Residual

Since only the finite precision scaled residual $W[j]$ is operated in the hardware implementation, we need to analyze how many guard digits, g, are enough to prevent the truncation error of the residual, ε_w, from affecting the correct selection of digits e_j, where the digit-width of the fractional part of $W[j]$ is assumed as the $q+g$-digit. Since the digit-width of

the fractional part of $W[1]$ is q-digit, the digit-width of the fractional part of $W[j]e_j10^{-j}$ is $q+j$-digit in each iteration. According to (16), the truncation error, 10^{-q-g+1}, is produced when the $q+g$-digit fractional part of $W[j]$ is not enough to represent $q+j$-digit $e_jW[j]10^{-j}$. Thus, we conclude that from $j=g+1$ to $j=g+q+1$ iterations, the truncated error of $W[j]$, 10^{-q-g+1}, is produced in each iteration. Thus, before the last iteration, $j=g+q+1$, the truncation error of $W[j]$, ε_w, is obtained as:

$$\varepsilon_w = 10^{-q-g+q} + 10^{-q-g+q-1} ... + 10^{-q-g+1} \tag{5.45}$$

Since the digit e_j is selected by rounding the scaled residual $\widehat{W}[j]$ to its integer part in each iteration, ε_w needs to satisfy the condition of $\varepsilon_w < 0.1$ in order to prevent the truncation error of $W[j]$, ε_w, from affecting the value of $\widehat{W}[j]$. To satisfy such a condition, the guard digit, g, should be at least 2-digit for three different DFP interchange formats in order to guarantee the correct selection of digit e_j.

5.4 Architecture of DFP Logarithmic Converter

Figure 5.1 shows the architecture of the proposed DFP logarithmic converter in the top level. Since such issues as the exception handling, the packing and the unpacking from IEEE 754-2008 DFP format are straightforward, we focus particularly on the architecture for the computation of the sign bit (R_{sign}), the real exponent (R_{exp}) and the decimal significand ($R_{significand}$) of DFP logarithm results. To represent the signed decimal intermediate value, all variables in the architecture are represented with 10's complement number system in the BCD encoding. The reason for choosing 10's complement format is that the decimal subtraction operation can be replaced by a decimal addition in 10's complement format, and all decimal digits, including the sign digit, can be operated in the decimal addition or subtraction.

In the data-path of the proposed architecture, the residual $W[j]$ is represented by the $q+g+$ 2-digit intermediate value (including 1-digit sign, 1-digit integer, q-digit fraction, and g-digit guard digit); the decimal significand $L[j]$ is represented by the $q+4$-digit intermediate value (including only $q+4$-digit fraction); and the digit e_j is represented by a 5-bit intermediate

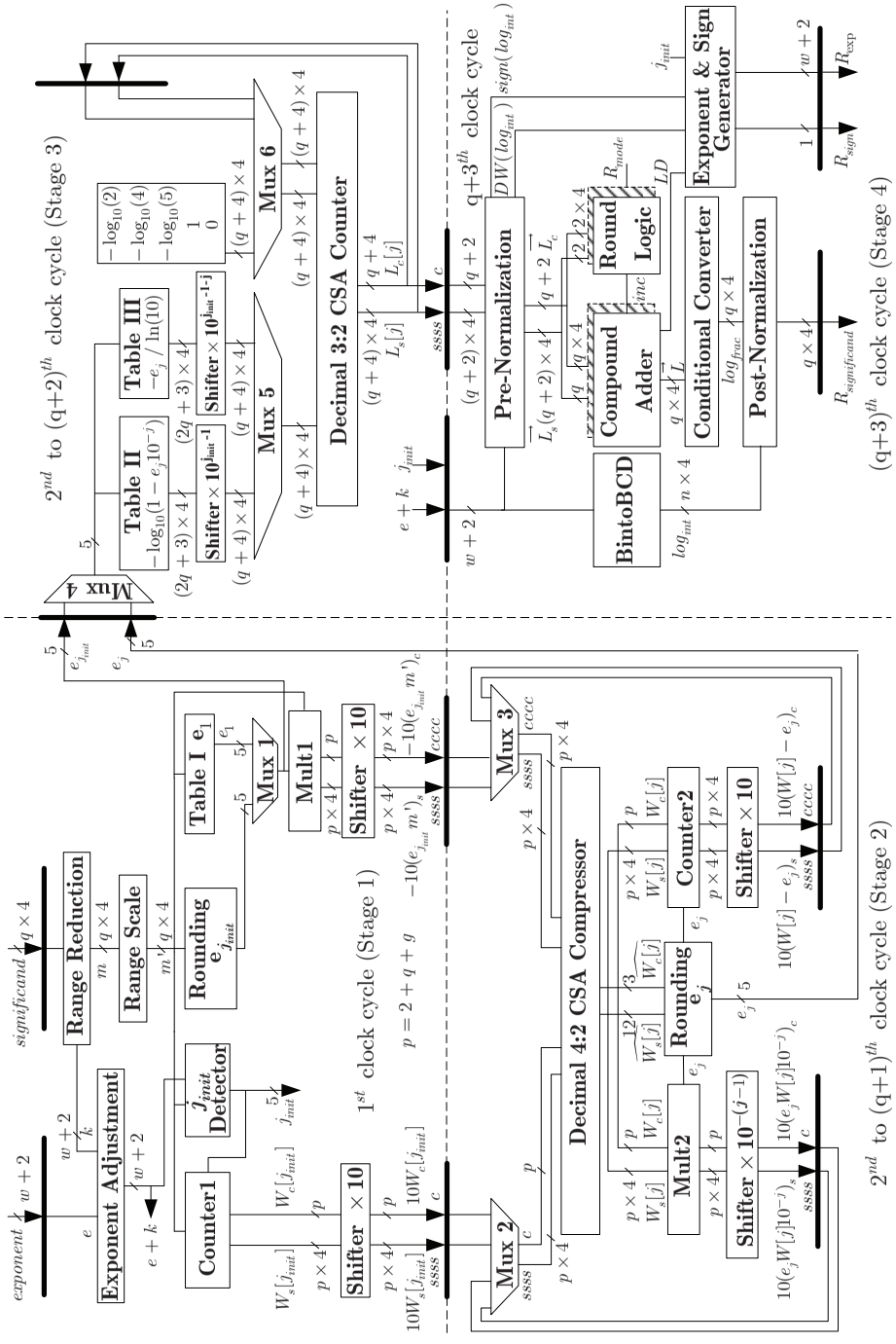

Figure 5.1: Improved architecture of DFP logarithmic converter.

value (including 1-bit sign and 1-digit absolute value of e_j). To speed-up the execution of recurrences, all intermediate values in the data-path are represented using the redundant decimal carry-save representation. For example, the residual $W[j]$ with the decimal carry-save representation is shown in Figure 5.2, where $ssss$ represents a 1-digit $W_s[j]$, c represents

$W_{s(\text{power of } 10)}$		$W_{s(0)}$	$W_{s(-1)}$	$\cdots$	$W_{s(-q-g-2)}$
	$sign$	$\widehat{W[j]}$ $t=1$			
$W_s[j]$	ssss	ssss.	ssss	$\cdots$	ssss
$W_c[j]$	c	c.	c	$\cdots$	c
$W_{c(\text{power of } 10)}$		$W_{c(0)}$	$W_{c(-1)}$		$W_{c(-q-g-2)}$

Figure 5.2: Decimal carry-save representation of $W[j]$.

a 1-bit $W_c[j]$, and $\widehat{W[j]}$ represents an estimated value of the residual $W[j]$ (including 3-digit most significand digits (MSDs) of $W[j]$). As a consequence of this representation, the delay of the addition and the multiply operation in the recurrence are independent of the computational precision.

5.4.1 Datapath

The data-path of the proposed architecture is pipelined and re-timed into four stages in order to minimize and balance the critical path delay. The initial processing stage (stage 1) aims to obtain the initial digit $e_{j_{init}}$, and then the digit recurrence stage (stage 2) is for achieving the remaining digits e_j. The logarithm computation stage (stage 3) is to achieve the q+4-digit decimal significand of the DFP logarithm result. Finally, 1-bit R_{sign}, w+2-bit R_{exp} and q-digit $R_{significand}$ of the DFP logarithm result are achieved in the final processing stage (stage 4). The cycle-based sequence of operations is summarized as follows:

- Stage 1, In 1^{st} clock cycle (In iteration ($j = j_{init}$)):

- The $w+2$-bit real exponent, and the q-digit non-normalized decimal significand, as input operands, are obtained from input registers. Since DFP input operands should be positive, the sign bit ($S=0$) is ignored.
- The q-digit decimal significand is normalized to a q-digit m in the range of $[0.1, 1)$ by a range reduction logic. Meanwhile, the value of $e+k$, as one of inputs of stage 4, is achieved in an exponent adjustment logic and then stored in a register.
- The normalized m is scaled to m' in the range of $0.5 \leq m' < 1.05$ (selected from m, $2m$, $4m$, $5m$ and $10m$) in a range scale logic.
- The index of the initial iteration, j_{init}, is obtained based on values of m' (refer to Table

2) and $e+k$ in a j_{init} detector.

- The initial residual $W[j]$ in the carry-save representation is achieved based on the value of j_{init} in a residual counter (counter1). If $j_{init}=1$, the digit $e_{j_{init}}$ is obtained from a look-up table I (refer to Table 5.1) based on the value of m'; otherwise, it is obtained by rounding 3-digit $\widehat{W[j]}$ in a rounding $e_{j_{init}}$ logic.

$$(W_s[j], W_c[j]) = 10^{j_{init}}(1-m')$$
$$e_j = mux(e_1, round(\widehat{W[j]}), j_{init})$$

- The $-e_j m'$ out from a multiply logic (Mult1) is shifted 1-digit to the left to achieve $-10e_j m'$, and $W[j]$ out from the residual counter is shifted 1-digit to the left to achieve $10W[j]$. The $10W[j]$ and $-10e_j m'$ are sent to the stage 2 by registers.

$$(10W_s[j], 10W_c[j]) = 10^{j_{init}+1}(1-m')$$
$$(-10e_j m'_s, -10e_j m'_c) = -10e_j \times m'$$

- Stage 2, From 2^{nd} to $(q+1)^{th}$ clock cycle (In iterations $j=j_{init}+1$ to $j=j_{init}+q$):

- In the 2^{nd} clock cycle, the residual $W[j]$ is obtained by adding $10W[j-1]$ (selected from Mux2) and $-10e_{j-1}m'$ (selected from Mux3) together in a decimal 4:2 CSA compressor. Thus, the digit e_j is obtained by rounding 3-digit $\widehat{W[j]}$ in a rounding e_j logic.

$$(W_s[j], W_c[j]) = 10W[j-1] - 10e_{j-1} \times m'$$
$$e_j = round(\widehat{W[j]})$$

- The intermediate value of $e_j W[j]$ out from a multiply logic (Mult2) is shifted j_{init}-digit to the right to obtain $10e_j W[j]10^{-j}$ in a barrel shifter, while the intermediate value of $W[j]-e_j$ out from a residual counter (counter2) is shifted 1-digit to the left to achieve $10(W[j]-e_j)$.

$$(10e_j W[j]10^{-j}_s, 10e_j W[j]10^{-j}_c) = 10e_j \times W[j]10^{-j}$$
$$(10(W[j]-e_j)_s, 10(W[j]-e_j)_c) = 10(W[j]-e_j)$$

- The intermediate values of $10(W[j]-e_j)$ and $10e_jW[j]10^{-j}$ are sent back to the stage 2 by registers for the recurrence in the next iteration.

$$(W_s[j+1], W_c[j+1]) = 10(W[j]-e_j+e_j \times W[j]10^{-j})$$
$$e_{j+1} = round(\widehat{W[j+1]})$$

After the $q+1^{st}$ clock cycle, all the digits e_j are achieved with the selection by rounding.

- Stage 3, From 2^{nd} to $(q+2)^{th}$ clock cycle (In iterations $j=j_{init}$ to $j=j_{init}+q$):

- In the 2^{nd} clock cycle, $e_{j_{init}}$ out from the stage 1, is sent to the stage 3, so that the corresponding value of $2q+3$-digit $-\log_{10}(1+e_j10^{-j})$, stored in a look-up table II, is obtained.
- To decrease the data width of the decimal addition in the stage 3, the value of $-\log_{10}(1+e_j10^{-j})$ is shifted $j_{init}-1$-digit to the left, so that $j_{init}-1$-digit leading zeros or leading nines of this value is eliminated.
- The $q+4$-digit MSDs of $-10^{(j_{init}-1)}\log_{10}(1+e_j10^{-j})$ and adjusted constant (0, -1, $-\log_{10}(2)$, $-\log_{10}(4)$ or $-\log_{10}(5)$) are selected by Mux5 and Mux6 respectively. Thus, the $q+4$-digit decimal significand of the DFP logarithm result is obtained in a $q+4$-digit decimal 3:2 CSA counter.

$$(L_s[j+1], L_c[j+1]) = L[j]-10^{(j_{init}-1)}\log_{10}(1+e_j10^{-j})$$

- From the 3^{rd} to $(q+2)^{th}$ clock cycle, the digits e_j out from the stage 2 are selected by Mux4, and the value of $L[j]$ is selected by Mux6 for the computation of $L[j+1]$ in the next iteration.
- From the number of $j=(q+2)^{th}$ to $j=(q+j_{init})^{th}$ iteration, the value of $2q+3$-digit $-e_j/\ln(10)$ is obtained from a look-up table III, and then shifted j-digit to the right to achieve the value of $-10^{-j}e_j/\ln(10)$ for approximating the value of $-\log_{10}(1+e_j10^{-j})$.
- The $q+4$-digit MSDs of $-10^{(j_{init}-j-1)}e_j/\ln(10)$ (after eliminating $j_{init}-1$-digit leading zeros) and $L[j]$ is added together to achieve $L[j+1]$ in the decimal 3:2 CSA counter.

$$(L_s[j+1], L_c[j+1]) = L[j]-10^{(j_{init}-1)}e_j10^{-j}/\ln(10)$$

After the $(q+2)^{th}$ clock cycle, the $q+4$-digit decimal significand of DFP logarithm results with at most one leading zero or one leading nine is obtained.

-Stage 4, In $(q+3)^{th}$ clock cycle:

- In the $(q+3)^{th}$ clock cycle, since the $R_{significand}$ consists of the integral part log_{int} (formed from the value of $e+k$) and the fractional part log_{frac} (formed from the value of $\log_{10}(m)$), the $q+2$-digit MSDs of the decimal significand ($L_s[j]$, $L_c[j]$) are pre-normalized based on the value of $e+k$ out from the stage 1 in order to determine the rounding position of $R_{significand}$. Figure 5.3 shows the data-path of the computation of the $\overrightarrow{L}$. The values of $L_s[j]$ and $L_c[j]$ (obtained from the stage 3) should be shifted to the right properly based on the digit width of the integral part, and then the rounding position is on the second last significand digit (LSD) or LSD (r_d or r'_d) of the shifted fractional part ($\overrightarrow{L_s}$, $\overrightarrow{L_c}$). The shift amount is obtained based on:

$$shift\_amount = \begin{cases} digit\_width(e+k-1) & \text{if } e+k>0 \\ digit\_width(e+k) & \text{if } e+k\leq 0 \end{cases}$$

  Then,

$$\overrightarrow{L_s} = r\_shift(L_s, shift\_amount)$$
$$\overrightarrow{L_c} = r\_shift(L_c, shift\_amount)$$

- The values of $\overrightarrow{L_s}$ and $\overrightarrow{L_c}$ are added together to obtain the value of L in a q-digit decimal compound adder. At the same time, the L is rounded to the faithful result based on the value inc of the rounding position in a rounding logic. Since we only consider the *roundTiesToEven* mode in this design, the rounding logic generates an increment inc based on:

$$inc = \begin{cases} 1 & \text{if } r_d>5 \text{ } or \text{ } (r_d=5 \text{ } and \text{ } LSB(L)=1) \\ 0 & \text{if } r_d<5 \text{ } or \text{ } (r_d=5 \text{ } and \text{ } LSB(L)=0) \end{cases}$$

  The other rounding modes can be implemented in a similar manner. To remove the possible one leading zero or nine in the L, a duplication of the decimal compound adder and the rounding logic are placed to obtain the value of L', and then the correct result, $\overrightarrow{L}$, is selected by the multiplexer based on the value of LD:

$$\overrightarrow{L} = \begin{cases} \overrightarrow{L_{s(16:1)}} + \overrightarrow{L_{c(16:1)}} + inc_0 & \text{if } LD=0 \\ \overrightarrow{L_{s(17:2)}} + \overrightarrow{L_{c(17:2)}} + inc_1 & \text{if } LD=1 \end{cases}$$

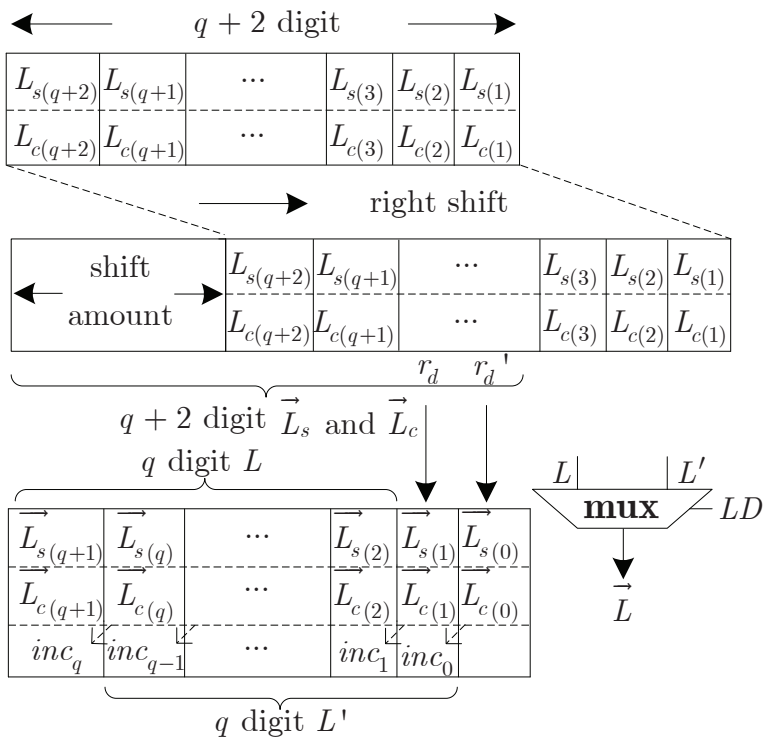

Figure 5.3: Data-path of the computation of L and L'.

Where, the value of LD is achieved based on:

$$LD = \begin{cases} 1 & \text{if } (e+k=1 \text{ and } \overrightarrow{L(\text{MSD})}=9) \text{ or} \\ & \quad (e+k=0 \text{ and } \overrightarrow{L(\text{MSD})}=0) \\ 0 & \text{otherwise} \end{cases}$$

- If the integral part of the decimal significand, $e+k \geq 1$, the fractional part of the decimal significand, log_{frac}, is obtained by the 10's complement conversion of $\overrightarrow{L}$, otherwise it is directly obtained from the value of $\overrightarrow{L}$ in a conditional converter.

$$log_{frac} = \begin{cases} 10\text{'s } com(\overrightarrow{L}) & \text{if } e+k \geq 1 \\ \overrightarrow{L} & \text{if } e+k < 1 \end{cases}$$

- The binary value of $w+2$-bit $e+k$ is converted to the n-digit BCD encoding as the integral part of the DFP logarithm result (log_{int}) in a binary to BCD converter. Note that n is equal to 3, 3 and 4 for Decimal32, Decimal64 and Decimal128 DFP formats respectively. Then, the values of log_{int} and log_{frac} are postnormalized to obtain the final decimal significand of the DFP logarithm result ($R_{significand}$).

- The 1-bit sign R_{sign} is obtained:

$$R_{sign} = \begin{cases} 0, & \text{if } e+k>0 \\ 1, & \text{if } e+k\leq 0 \end{cases}$$

Then, the $w+2$-bit exponent R_{exp} is obtained based on:

$$R_{exp} = \begin{cases} -(q+LD+j_{init}-1) & \text{if } e+k=0 \text{ or } 1 \\ -(q-DW(log_{int})) & \text{otherwise} \end{cases}$$

Table 5.4 shows some iterations of a 64-bit DFP logarithm operation executed in the proposed architecture.

5.4.2 Hardware Implementation

The details of the hardware implementation for the each stage of the proposed DFP logarithmic converter are presented in this section. In the rest of this section, the symbols of $\oplus$, $\wedge$, $\vee$ and & represent the logical-XOR, logical-AND, logical-OR and logical-concatenation respectively. The symbol of $(A)_x^y$ refers to the y-th bit in digit position, x, in a decimal number, A, where the least significant bit and the least significant digit have the index of 0. For example, $(W[j])_2^3$ is the third bit of the second BCD digit in $W[j]$. The symbol of $\overline{A}$ refers to the logic-NOT of a number of A.

Initial Processing Stage

Figure 5.4 shows the details of the hardware implementation of the initial processing stage (stage 1).

-In the stage 1: The range reduction logic consists of a leading-zero-counter (LZC) and a decimal barrel shifter. The LZC is applied to count the number of leading zeros (z-digit) of a non-normalized input decimal significand. These leading zeros are shifted z-digit to the left by the decimal barrel shifter to achieve a normalized m. While the LZC is implemented based on [124], the decimal barrel shifter that can shift a decimal significand by any amount from 0 to $q-1$ digits is implemented by a $\log_2(q)$ levels of multiplexors. The exponent adjustment logic is implemented by a binary CLA adder to achieve the value of $e+k$, where $k=q-z$ which is obtained from the LZC.

Table 5.4: Example of a Decimal64 logarithm operation.

$v = (-1)^0 \times 9986451368175534 \times 10^{-16}$, $R = k + e + \log_{10}(m) = 16 - 16 + \log_{10}(0.9986451368175534)$, $e + k = 0$
$\rightarrow m = 0.9986451368175534$, $m' = 0.9986451368175534$, $j_{init} = 3$ (Recurrence starts from 3^{rd} Iteration)

	$\widehat{W[j]}$	$W[j]$	$L[j]$
$W_s[3] = 10^3(1 - m')$	013	01.354863182446500000	Adjusted Constant
$W_c[3]$	000	00.000000000000100000	$= .0000000000000000000$
$\widehat{W[3]}$	+13	$\rightarrow$ $e_3 = +1$ $\rightarrow$ $10^2 \times \log_{10}(1 + e_3 10^{-3}) =$	$.0434077479318640668$
$10(e_3 m')_s$		99.986451368175534000	
$10(e_3 m')_c$		00.000000000000000001	
$10W_s[3]$		13.548631824465000000	
$10W_c[3]$		+ 00.000000000001000000	**In 1^{st} clock cycle (3^{rd} iteration)**
$W_s[4]$	035	03.551179455289354890	$L_s[3] = .0434077479318640668$
$W_c[4]$	000	00.011001001001111110	$L_c[3] = .0000000000000000000$
$\widehat{W[4]}$	+35	$\rightarrow$ $e_4 = +4$ $\rightarrow$ $10^2 \times \log_{10}(1 + e_4 10^{-4}) =$	$.0173683058464918822$
$10(W[4]e_4 10^{-4})_s$		00.014148711825151853	
$10(W[4]e_4 10^{-4})_c$		00.000100010000010010	
$10(W[4] - e_4)_s$		95.511794552893548900	
$10(W[4] - e_4)_c$		+ 00.110010010011111100	**In 2^{nd} clock cycle (4^{th} iteration)**
$W_s[5]$	956	95.635042283729711863	$L_s[4] = .0507650427772558480$
$W_c[5]$	000	00.001011001000110000	$L_c[4] = .0100110110011001011$
$\widehat{W[5]}$	−44	$\rightarrow$ $e_5 = -4$ $\rightarrow$ $10^2 \times \log_{10}(1 + e_5 10^{-5}) =$	$.9982627873279019177$
$10(W[5]e_5 10^{-5})_s$		00.014148711825151853	
$10(W[5]e_5 10^{-5})_c$		00.110010010011111100	
$10(W[5] - e_5)_s$		95.511794552893548900	
$10(W[5] - e_5)_c$		+ 00.001011001000110000	**In 3^{rd} clock cycle (5^{th} iteration)**
...	...	... ...	...
$W_s[18]$	007	00.762902058900277420	$L_s[17] = .0588085715297893522$
$W_c[18]$	000	00.000000000000000000	$L_c[17] = .0000000000000000000$
$\widehat{W[18]}$	+07	$\rightarrow$ $e_{18} = +1$ $\rightarrow$ $10^2 \times e_{18} 10^{-18}$ / $\ln(10) =$	$.0000000000000004342$
$10(W[18]e_{18} 10^{-18})_s$		00.000000000000000007	
$10(W[18]e_{18} 10^{-18})_c$		00.000000000000000000	
$10(W[18] - e_{18})_s$		97.629020590002774200	
$10(W[18] - e_{18})_c$		+ 00.000000000000000000	**In 16^{th} clock cycle (18^{th} iteration)**
$W_s[19]$	976	97.629020590002774207	$L_s[18] = .0588085715297897864$
$W_c[19]$	000	00.000000000000000000	$L_c[18] = .0000000000000000000$
$\widehat{W[19]}$	−24	$\rightarrow$ $e_{19} = -2$ $\rightarrow$ $10^2 \times e_{19} 10^{-19}$ / $\ln(10) =$	$.9999999999999999131$
			In 17^{th} clock cycle (19^{th} iteration)
In 18^{th} clock cycle			$L_s[19] = .9477977460418678699$
			$L_c[19] = .1111011111111111000$

In 19^{th} clock cycle $e + k = 0$ $\rightarrow$ $shift\_amount = 0$ $\rightarrow$ $\overrightarrow{L_s} = .9477977460418678699$ $\leftarrow r_d'$
$log_{int} = 0$ $log_{frac} = .5888085715297897$ $\leftarrow$ $\vec{L} = .5888085715297897$ $\leftarrow$ $\overrightarrow{L_c} = .1111011111111111000$
$R_{sign} = 1$ $R_{\exp} = -19$ ("1111101101") $R_{significand} = 5888085715297897$ + compound addition 1 $\leftarrow inc$

The range scale logic consists of a multiple generation block (generating m, $2m$, $4m$, $5m$ and $10m$), a selection control block and a 5-to-1 decimal multiplexer. Both $2m$ and $5m$ can be generated with only a few logic delays, since there is no carry propagation beyond the next more significant digit. Further, due to the simplicity of generating $2m$, $4m$ can be generated via connecting two doublers in series. The boolean equations for generating double and quintuple of the BCD number are presented in [125]. The control signal (sel) of the

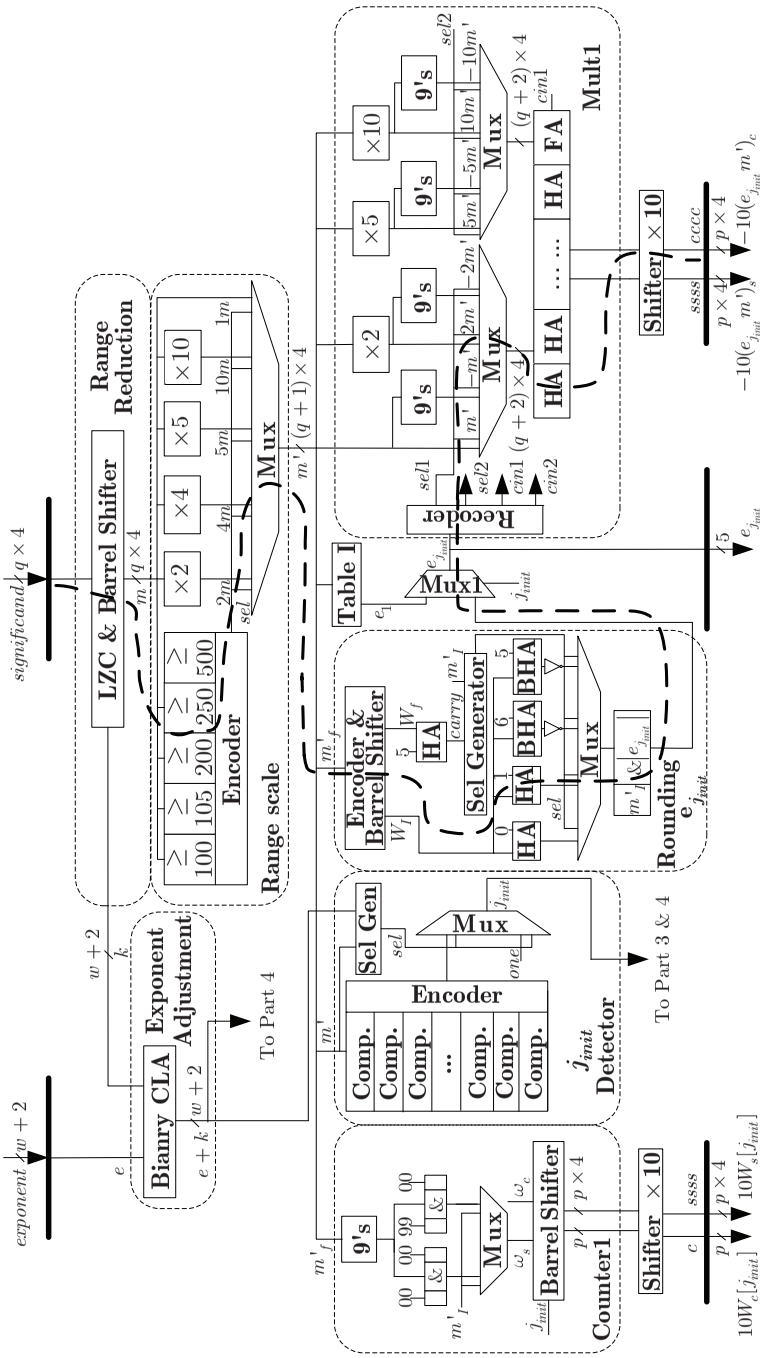

Figure 5.4: Hardware implementation of Stage 1 in DFP logarithmic converter.

decimal 5-to-1 multiplexer is generated by the selection control block which is implemented by five parallel decimal comparators and an encoder. If m' is in the range of $1 \leq m' < 1.05$, the MSD of m' (m'_I), is one, otherwise, it is zero. While the signal j_{init} is generated from the j_{init} detector that is mainly implemented by a series of parallel decimal comparators and

an encoder. The control signal (sel) is generated based on the value of $e+k$ in a selection generator. Thus, if $e+k=0$ or 1, the signal j_{init} is obtained from the encoder, otherwise, it is obtained from a constant of one.

The residual counter (counter1) consists of a $9's$ complement converter and a decimal barrel shifter. Since the value of $1-m'$ can be achieved based on:

$$1 - m' = \left| 9\text{'s } com(m'_f) + 1 \text{ } LSD \right|$$

where, m'_f represents the q-digit fraction part of m'. Thus, the signal of $(1–m')_s$ is represented by the q-digit signal of $9\text{'s } complement(m'_f)$, and the q-bit signal of $(1-m')_c$is represented by the constant of "00...01". If the signal $m'_I=1$, the q-digit $(1-m')_s$ is extended to the $q+g+2$-digit signal $w_s=99$ & $(1-m')_s$ & 00 by concatenating 1-digit sign ('9'), 1-digit integer part ('9') and g-digit ("00") guard digit, otherwise it is extended to $w_s=00$ & $(1–m')_s$ & 00. The corresponding q-bit signal of $(1–m')_c$is extended to $q+g+2$-bit signal $w_c=00$ & $(1–m')_c$ & 00. Then, the $q+g+2$-digit $(1-m')$ is shifted j_{init}-digit to the left to obtain $W_s[j_{init}]$ and $W_c[j_{init}]$ in the decimal barrel shifter.

The rounding $e_{j_{init}}$ logic for selecting digits $e_{j_{init}}$ is processed by rounding the residual $\widehat{W[j_{init}]}$. In order to decrease the delay of the stage 1, the rounding $e_{j_{init}}$ logic directly rounds the residual $\widehat{W[j_{init}]}$ instead of ($W_s[j_{init}]$, $W_c[j_{init}]$) obtained from the counter1. The signal m'_f is shifted j_{init}-digit (obtained from an encoder instead of from the j_{init} detector) to the left to achieve the signal $W[j_{init}]$ in a barrel shifter, and then the 2-digit MSDs of the signal $W[j_{init}]$ (W_I, W_f) is used to round the digits $e_{j_{init}}$. The 1-digit fraction W_f and the value of 5 are added together in a 1-digit decimal half adder to generate the signal $carry$ to determine the rounding operation. The signals $carry$ and m'_I are sent to selection generator to achieve a control signal sel of a 4-to-1 multiplexer. The value of $|e_{j_{init}}|$ is achieved in four parallel half adders (two 1-digit decimal adders, and two 4-bit binary adders) by adding the value of 0, 1, 6 and 5 with the signals of W_I respectively:

$$|e_{j_{init}}| = \begin{cases} W_I+0 & \text{if } m'_I=1 \wedge carry=0 \\ W_I+1 & \text{if } m'_I=1 \wedge carry=1 \\ \overline{W_I+6} & \text{if } m'_I=0 \wedge carry=1 \\ \overline{W_I+5} & \text{if } m'_I=0 \wedge carry=0 \end{cases}$$

Thus, the digit $e_{j_{init}}$ is obtained by concatenating 1-bit m'_I with 1-digit $|e_{j_{init}}|$.

The multiply logic (Mult1) is applied to compute the value of $-e_{j_{init}}m'$, where $e_{j_{init}}$ is a value in the range of $-9 \leq e_{j_{init}} \leq 9$, so the Mult1 is to obtain the results from $-9m'$ to $9m'$. The Mult1 is implemented based on the partial product generation logic presented in [23]. The multiples are formed by adding two of a initial multiple set $\{m', -m', 2m', -2m', 5m', -5m', 10m', -10m'\}$ (selecting by signals *sel1* and *sel2* generated by a recorder). To decrease the delay of the addition, a decimal CSA adder is implemented to develop multiples $-(e_{j_{init}}m')_s$ and $-(e_{j_{init}}m')_c$. The boolean equation for computing 1-digit decimal addition of the BCD number is presented in [20]. The signals *cin1* and *cin2* are generated by a recorder to supplement the LSD due to the 10's complement conversion. The signal *cin1* is added in the LSD of the CSA adder, while the signal *cin2* is set in the LSD of the signal $-(e_{j_{init}}m')_c$.

Digit Recurrence Stage

Figure 5.5 shows the details of the hardware implementation of the digit recurrence stage (stage 2).

- In the stage 2: The 4:2 decimal CSA compressor, applied to achieve the residual ($W_s[j]$, $W_c[j]$), is implemented by two levels of q+g+2-digit 3:2 CSA counters. Then, 1-digit sign (S_s, S_c), 1-digit integer part(I_s, I_c) and 1-digit fraction part ($f_{s_{MSD}}$, $f_{c_{MSB}}$) of the residual are sent to the rounding e_j logic for selecting digits e_j by rounding the residual ($\widehat{W_s[j]}$, $\widehat{W_c[j]}$). The sign of the digit e_j is obtained by the sign detector block which is implemented based on the equation:

$$sign = (S_s^0 \oplus S_c) \oplus (I_s^3 \wedge I_s^0 \wedge I_c)$$

The 1-digit fraction ($f_{s_{MSD}}$, $f_{c_{MSB}}$) and the value of 5 are added together in the 1-digit decimal full adder to generate the signal *carry* to determine the rounding operation. The signals *carry* and *sign* are sent to selection generator to achieve a control signal *sel* of a 4-to-1 multiplexer. The value of $|e_j|$ is achieved in four parallel full adders by adding the

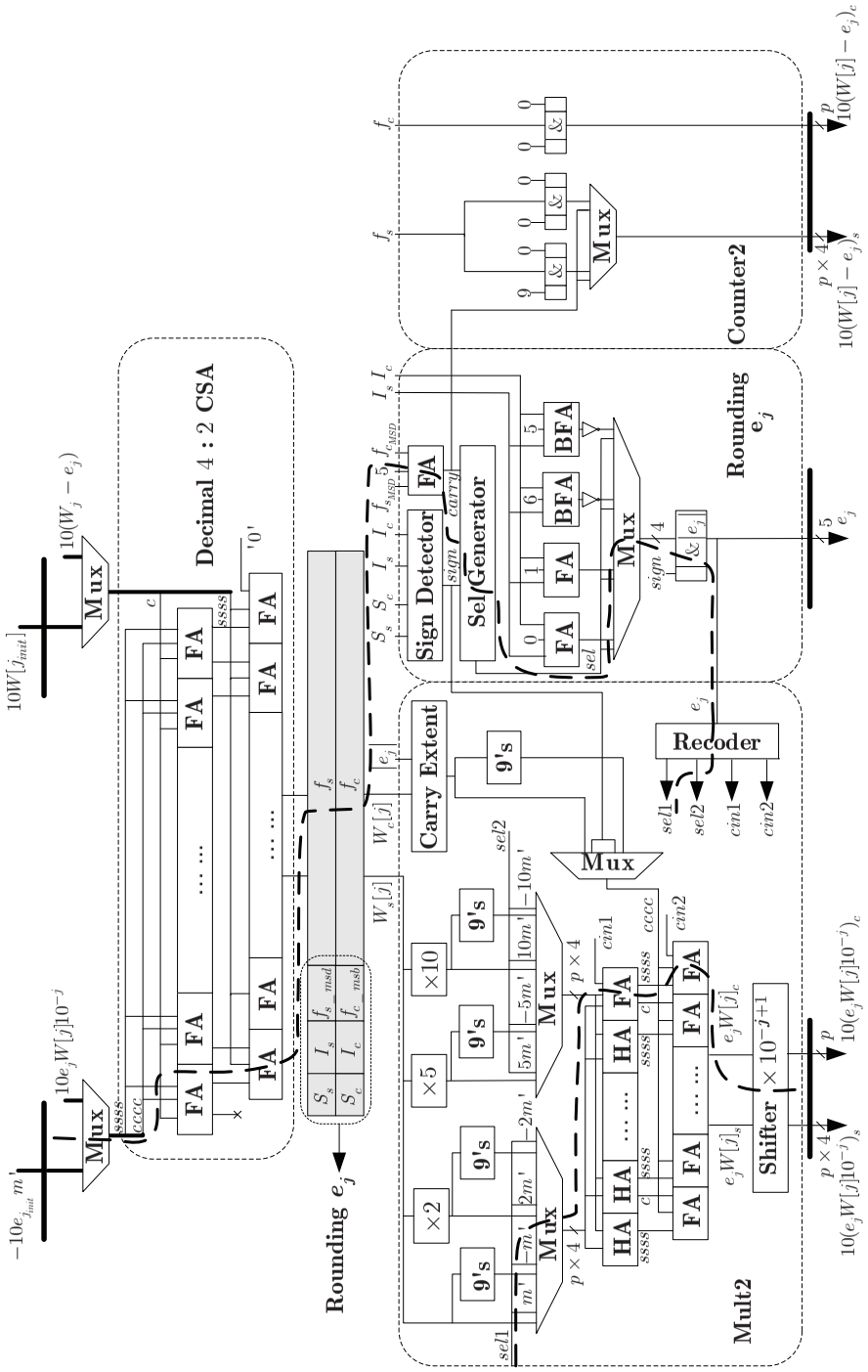

Figure 5.5: Hardware implementation of Stage 2 in DFP logarithmic converter.

value of 0, 1, 6 and 5 with the signals of $f_{s_{MSD}}$ and $f_{c_{MSB}}$ respectively:

$$|e_j| = \begin{cases} I_s + I_c + 0 & \text{if } sign = 0 \wedge carry = 0 \\ I_s + I_c + 1 & \text{if } sign = 0 \wedge carry = 1 \\ \overline{I_s + I_c + 6} & \text{if } sign = 1 \wedge carry = 0 \\ \overline{I_s + I_c + 5} & \text{if } sign = 1 \wedge carry = 1 \end{cases}$$

Thus, the digit e_j is obtained by concatenating 1-bit *sign* with 1-digit $|e_j|$.

The multiply logic (Mult2) is applied to compute the value of $W_s[j]e_j+W_c[j]e_j$, where e_j is a value in the range of $-9 \leq e_j \leq 9$. The multiple of $W_s[j]e_j$ is achieved by applying the same hardware implementation as the Mult1. Since each bit of $W_c[j]$ is only zero or one, the signal of $W_c[j]\,|e_j|$ can be achieved in a carry extend block which can be implemented by a series of logical-AND gates. If the digit $e_j < 0$ ($sign=1$), the signal of $W_c[j]e_j$ is obtained by the 9's complement conversion of $W_c[j]\,|e_j|$, and then the signal *sign* is supplemented in the LSD of the signal $(e_jW[j])_c$, otherwise, the signal of $W_c[j]e_j$ is directly obtained from the signal of $W_c[j]\,|e_j|$.

$$(W_c[j]e_j)_i^{3:0} = \begin{cases} W_c[j]_i \wedge e_j^{3:0} & \text{if } sign=0 \\ 9\text{'s } com(W_c[j]_i \wedge e_j^{3:0}) & \text{if } sign=1 \end{cases}$$

Thus, the value of $W[j]e_j$ ($e_jW[j]_s$, $e_jW[j]_c$) are achieved by adding the $W_s[j]e_j$ and $W_c[j]e_j$ in a decimal CSA adder. Finally, the signals of $10W[j]e_j10_s^{-j}$ and $10W[j]e_j10_c^{-j}$ are obtained in a decimal barrel shifter.

The residual counter (counter2) is applied to compute the value of $10(W[j]-e_j)$. Since the digit e_j is obtained by rounding the residue $W[j]$ to its integer part (I_s, I_c), the signal of $10(W[j]-ej)_s$ and $10(W[j]-e_j)_c$ can be achieved based on the fraction part (f_s, f_c) of the residue:

$$10(W[j]-e_j)_s = \begin{cases} 0 \;\&\; f_s \;\&\; 0 & \text{if } carry=0 \\ 9 \;\&\; f_s \;\&\; 0 & \text{if } carry=1 \end{cases}$$

$$10(W[j]-e_j)_c = 0 \;\&\; f_c \;\&\; 0$$

Thus, the counter2 is implemented by s set of simple concatenation block instead of a decimal subtracter.

Logarithm Computation Stage

Figure 5.6 shows the details of the hardware implementation for the logarithm computation stage (stage 3).

- In the stage 3: The look-up table II stores all the values of the $2q$+3-digit $-\log_{10}(1+e_j10^{-j})$

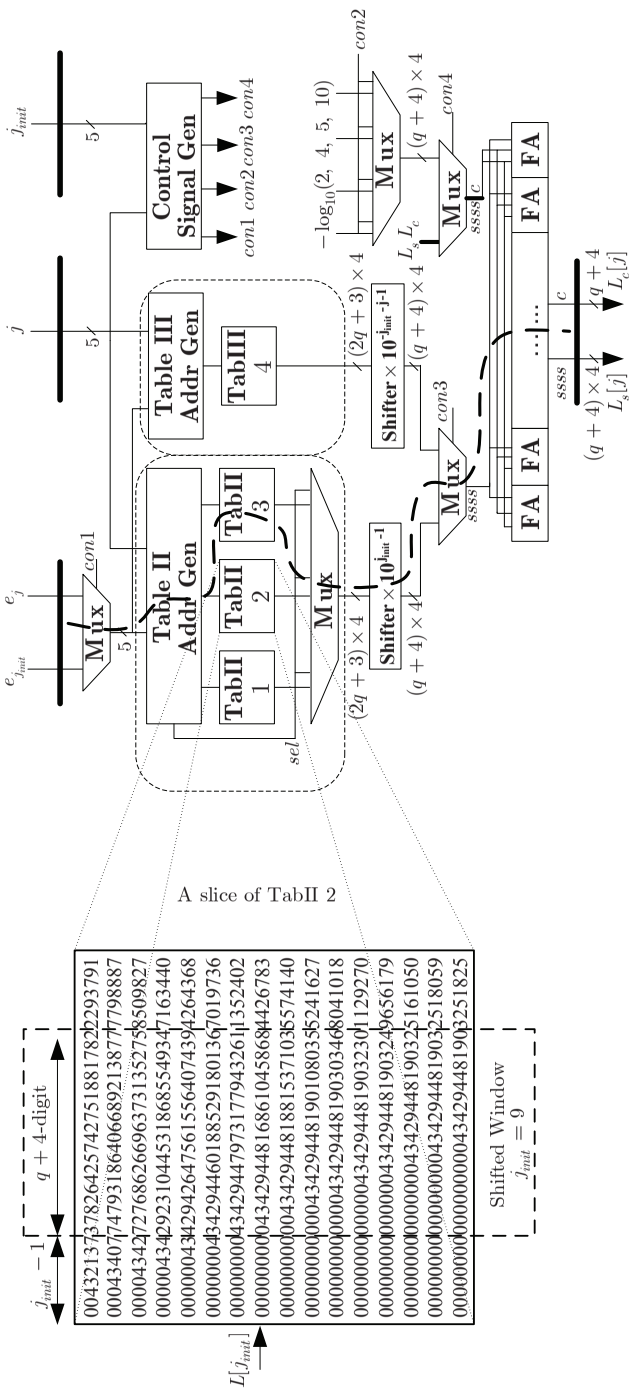

Figure 5.6: Hardware implementation of Stage 3 in DFP logarithmic converter.

for achieving at most $2q$-digit accuracy decimal logarithm results, where the number of iteration j is in the range of $1 \leq j \leq q+1$. Since the digit e_j is in the range of $-9 \leq e_j \leq 9$, there are a total of 18 different values (except for the value when $e_j = 0$) that need to be stored in the look-up table for each iteration. Since the address of the look-up table in hardware

must be restricted to a power of two, the size of the look-up table II can be obtained as:

$$\text{TabII size}=2^{\lceil \log_2[18\times(q+1)]\rceil}\times(2q+3)\times 4 \text{ bit}$$

Thus, the size of the look-up table II is $2^8\times$ 68-bit, $2^9\times$140-bit and $2^{10}\times$284-bit for Decimal32, Decimal64 and Decimal128 respectively. To reduce the size and delay of look-up table II, the values of $2q+3$-digit $-\log_{10}(1+e_j10^{-j})$ can be efficiently reallocated in multiple tables. For Decimal64 (the example shown in Figure 5.6), the single look-up table II is relocated into there parts as follows: 1) the first part (TabII 1) stores all the values of $-\log_{10}(1+e_j10^{-j})$, when $j=1$ and $0\leq e_1\leq 9$; 2) the second part (TabII 2) stores the values when $2\leq j\leq 17$ and $e_j=\pm 1$; and 3) the third part (TabII 3) stores the values when $2\leq j\leq 17$, and $2\leq e_j\leq 9$ and $-9\leq e_j\leq -2$. The sizes of the TabII 1, TabII 2 and TabII 3 are $2^4\times 140$, $2^5\times 140$ and $2^8\times$140 respectively. Thus, the optimized size of look-up table II is reduced from 8.75 KByte (single table) to only 5.20 KByte (multiple tables). The look-up table III stores the number of 19 values of $2q+3$-digit $-e_j/\ln(10)$. Thus, the total optimized size of look-up table is about 5.52 Kbyte for Decimal64. The implementations of address generators to address the look-up table II and the look-up table III based on the values of j and e_j are straightforward.

A slice of TabII 2 is shown in Figure 5.6, in which the corresponding values of $2q+3$-digit $-\log_{10}(1+e_j10^{-j})$ obtained from the look-up table II is shifted $j_{init}-1$-digit to the left, and then the $q+4$-digit MSDs of the shifted value is obtained in a shifted window. The control signals (*con1*, *con2*, *con3* and *con4*) of multiplexors are generated by the control signal generator which is implemented straightforwardly. To decrease the delay of this stage, the decimal significand ($L_s[j]$, $L_c[j]$) is obtained in a $q+4$-digit decimal CSA adder.

Final Processing Stage

Figure 5.7 shows the details of the hardware implementation for the final processing stage (stage 4).

- In the stage 4: The pre-normalization logic consists of a shift amount detector and a barrel shifter. The shift amount detector is implemented by a series of comparators and an encoder to determine the signal of the shift amount for the barrel shifter. The sign and the digit width of log_{int} (*sign*(log_{int}), *DW*(log_{int})) are also obtained from the shift amount

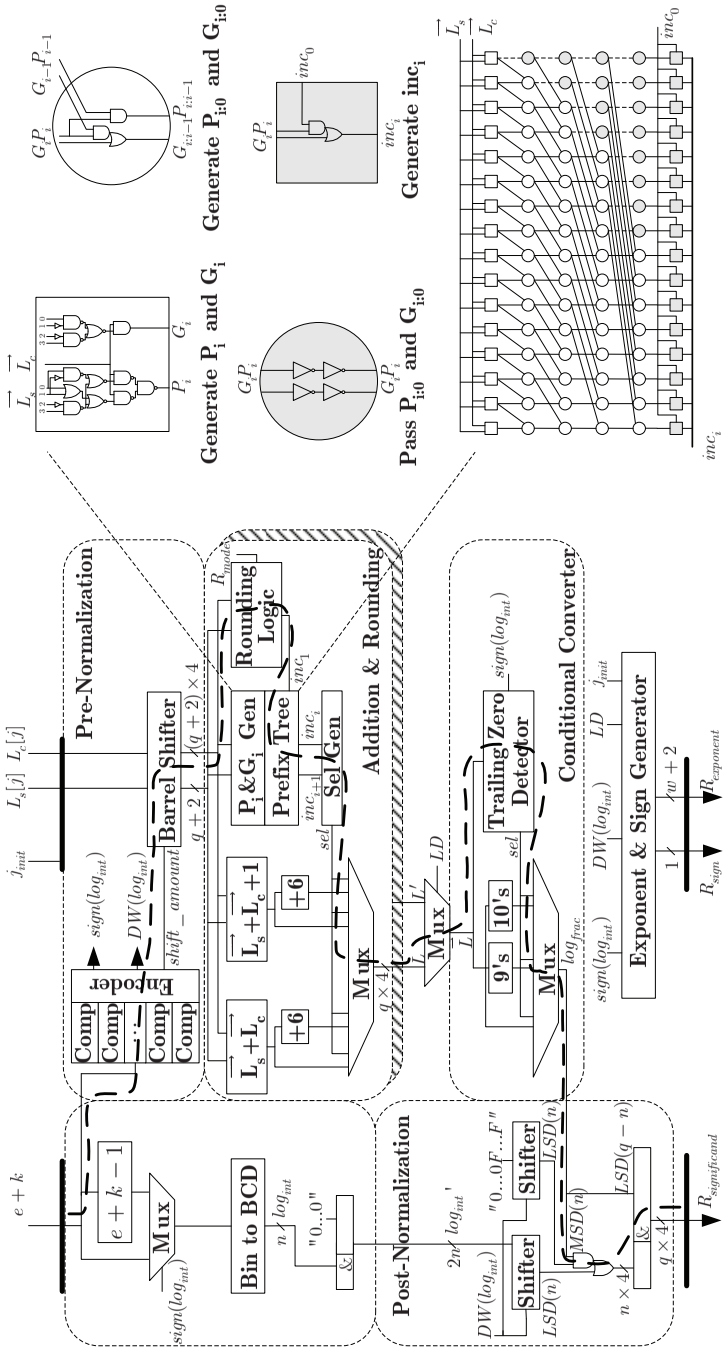

Figure 5.7: Hardware implementation of Stage 4 in DFP logarithmic converter.

detector.

The decimal compound adder is implemented based on the conditional speculative method [105]. A prefix tree based on the binary Kogge-Stone network [126] is used to generate carries into each digit of the decimal addition. The portion of the prefix tree for the Decimal64 for-

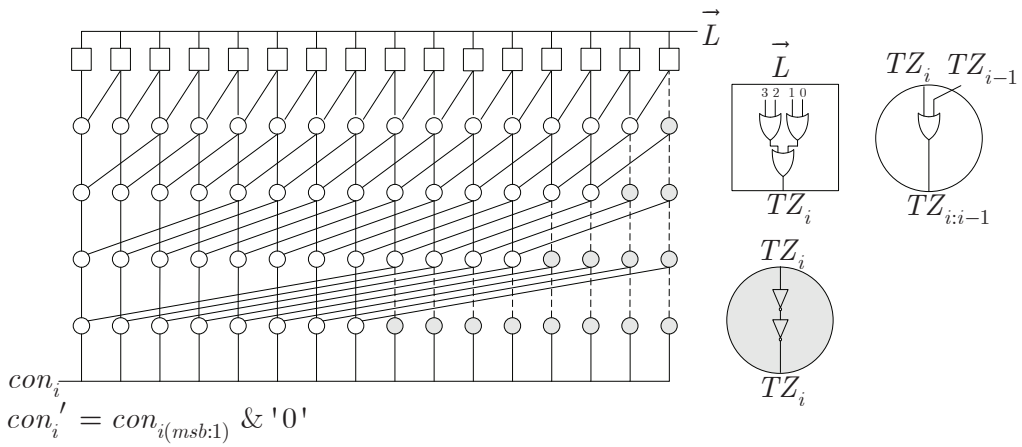

Figure 5.8: Portion of a trailing zeros detector for Decimal64.

mat, and notes of the tree (P_i and G_i obtained from the $P_i\&G_i$ generator) are shown in Figure 5.7. The carries generated by the prefix tree (inc_i and inc_{i+1}) are used to select the correct addition result of the each digit under the following conditions:

$$\overrightarrow{L_i} = \begin{cases} \overrightarrow{L_{s_i}} + \overrightarrow{L_{c_i}} & \text{if } inc_i = 0 \wedge inc_{i+1} = 0 \\ \overrightarrow{L_{s_i}} + \overrightarrow{L_{c_i}} + 1 & \text{if } inc_i = 1 \wedge inc_{i+1} = 0 \\ \overrightarrow{L_{s_i}} + \overrightarrow{L_{c_i}} + 6 & \text{if } inc_i = 0 \wedge inc_{i+1} = 1 \\ \overrightarrow{L_{s_i}} + \overrightarrow{L_{c_i}} + 1 + 6 & \text{if } inc_i = 1 \wedge inc_{i+1} = 1 \end{cases}$$

The additions of $\overrightarrow{L_{s_i}}+\overrightarrow{L_{c_i}}$ and $\overrightarrow{L_{s_i}}+\overrightarrow{L_{c_i}}+1$, can be implemented using three binary half adders and a binary full adder connected as a ripple carry chain. The logic for adding the value of 6 is used to compensate the $\overrightarrow{L_i}$ to the correct representation of the BCD encoding, which can be implemented using two binary half adders and two binary full adders.

The conditional converter consists of a 10's complement converter, a 9's complement converter, a trailing zeros detector and a 3-to-1 multiplexer. To decrease the carry propagation in the 10's complement conversion, the trailing zeros detector is applied to determine the position of the MSD of the trailing continuous zeros. The trailing zeros detector is implemented based on a prefix tree to generate control signals of con_i and con_i'. The portion of a trailing zeros detector for Decimal64 format is shown in Figure 5.8. Thus, the computation

of the each digit of the conditional converter is implemented based on:

$$log_{frac_i} = \begin{cases} 9\text{'s } com(\overrightarrow{L_i}) & \text{if } e+k>0 \wedge con_i=1 \wedge con'_i=1 \\ 10\text{'s } com(\overrightarrow{L_i}) & \text{if } e+k>0 \wedge con_i=1 \wedge con'_i=0 \\ \overrightarrow{L_i} & \text{otherwise} \end{cases}$$

A binary to BCD converter, which is used to convert the binary value of $e+k$ or $e+k-1$ to the n-digit integral part of the decimal significand (log_{int}), is implemented based on [103]. To concatenate log_{int} with log_{frac} properly in the post-normalization logic, first, n-digit zeros is regarded as a suffix to the signal log_{int} to form the signal log'_{int}; second $2n$-digit log'_{int} is shifted to $DW(log_{int})$-digit to the right, and a $2n$-digit mask (including n-digit MSDs of 0, and n-digit LSDs of F) is shifted $DW(log_{int})$-digit to the right at the same time; third, the n-digit LSDs of the shifted mask is operated (with the logic-AND operation) with the n-digit MSDs of log_{frac}, the result then is operated (with the logic-OR operation) with the n-digit LSDs of the shifted log'_{int}; fourth, the $q-n$-digit LSDs of the fractional part log_{frac} is concatenated with the n-digit result (obtained from the logic-OR operation) to achieve the q-digit final decimal significand ($R_{significand}$).

5.5 Implementation and Comparisons

The proposed improved DFP logarithmic converter that can compute operands in Decimal32, Decimal64 and Decimal128 formats, are modeled with VHDL and then simulated by using ModelSim respectively. A comprehensive testbench, which includes special test cases (NaN, Infinite, Subnormal or zero operands), corner test cases (close to one operands), and valid random DFP operands is performed to verify the correctness of the design. The proposed architectures are synthesized using Synopsys Design Compiler with the STM 90-nm CMOS standard cells library [127] under the typical condition (1.2 V_{DD} core voltage and 25 ^oc operating temperature). The clock, input signals, and output signals are assumed to be ideal. Inputs and outputs of the proposed design are registered and the design is optimized for delay.

The delay model is based on logical effort method [128], which estimates the proposed architecture delay values in a technology independent parameter, FO4 unit (the delay of an

Table 5.5: Delay and area of Decimal64 logarithmic converter.

Stage	**Worst Delay** (FO4)	**Areas** (NAND2)
Initial processing stage (Figure 5.4)	34.0	13422
Digit recurrence stage (Figure 5.5)	34.4	13454
Log computation stage (Figure 5.6)	29.6	10870
Final processing stage (Figure 5.7)	30.0	5075
Top-level control logic (FSM*)	3.5	751
Total	34.4**	43572

* FSM: finite-state machine; ** critical path delay.

Table 5.6: Details of critical path of Decimal64 logarithmic converter.

Blocks in the critical path								**Total**
Reg	buffer	Mux	CSA	Round	Mult2	Shift	setup	(*ns*)
0.08	0.09	0.06	0.29	0.21	0.51	0.23	0.08	1.55

inverter of the minimum drive strength (1x) with a fanout of four 1x inverters). To measure the total hardware cost in terms of number of gates, the area of the proposed architectures are estimated as the number of equivalent 1x two input NAND gates (NAND2). Note that 1 FO4$\approx$45ps, and 1 NAND2$\approx$4.4um^2 in the STM 90-nm CMOS standard cells library under the typical condition. Table 5.5 summarizes the delay and the area estimated using the area and delay evaluation model for the Decimal64 logarithmic converter. The worst path delay of each stage is highlighted in the corresponding figure by dashed thick line. The evaluation results show that the critical path of the proposed architecture is located in the stage 2 (highlighted in Figure 5.5), and the details of critical path in the Decimal64 implementation are reported in Table 5.6.

Since there is no comparable Decimal64 logarithmic converter, we reconstruct a 16-digit DXP logarithmic converter based on the proposed Decimal64 architecture, and then compare its results with those of a 53-bit radix-16 binary logarithmic converter [94] in terms of the critical path delay. We bring the 16-digit DXP logarithmic converter and the 53-bit

into comparison due to the following reasons: 1) they have similar dynamic ranges for the normalized coefficients ($2^{52} < 10^{16} < 2^{53}$); 2) they are implemented by the same digit-recurrence algorithms with selection by rounding; and 3) the radix-10 is close to radix-16 (processing 4-bit binary number in each iteration). To conduct a fair comparison, the timing evaluation unit ($1\tau =$ the delay of 1-bit full adder) applied in [94], is transformed to the unit of FO4 based on [57]. Since $1\tau \approx 0.8ns$ in the AMS 0.35-um CMOS standard cells library (used in [57]), we obtain $1\tau \approx 5.3$ FO4 so that the delay of the design [94] is evaluated by FO4 unit. Note that 1 FO4 $\approx 150ps$ in the AMS 0.35-um CMOS standard cells library under the typical condition. The comparison results in Table 5.7 show that the proposed 16-digit logarithmic converter has a latency close to that of the 53-bit radix-16 binary logarithmic converter in [94].

We also compare the results of the proposed design with our previous design [122] implemented based on the non-redundant data-path. The critical path delay of the original 16-digit DXP logarithmic converter is 9.28 ns (synthesized with the TSMC 0.18-um standard cell library in the typical condition, thus, 1 FO4 $\approx 75ps$). The comparison results reported in Table 5.7 show that the improved decimal logarithmic converter is 3.62 times faster than the original design in terms of the latency.

With respect to the existing works implemented based on the CORDIC algorithm in [49], it is quite difficult to compare the hardware performance between the two different algorithms. To compute the Decimal128 logarithm operation, the expression given in [49] for the number of clock cycles for a generic implementation of the fast termination CORDIC algorithm is no. cycles $= 2+16\times m+D+P$, where $m = 19$ (the number of iterations), $D = 67$ (the estimated number of cycles of a 16-digit DXP divider), and $P = 5$ (the estimated number of cycles specific to the DFP process, such as the exception handling, the packing and the unpacking from IEEE 754-2008 DFP format). For Decimal64 comparison, we have $m = 9$, $D = 49$ (the estimated number of cycles of a 7-digit DXP divider), thus the total estimated number of clock cycles to compute the Decimal64 logarithm operation is 195 (no. cycles $= 2+16\times 9+49$, without the consideration of the cycles specific to the DFP process). The critical path delay of the implementation in [49] has the cycle time of 13 FO4 (taking the Power 6 processor as a reference). The comparison results reported in Table 5.7 show that

Table 5.7: Comparison results for the delay of Decimal64 logarithmic converter.

Works	**Cycle time** (FO4)	**Cycles** (No.)	**Latency** (FO4)	**Ratio**
Proposed	34.4	19	653.6	1.00
Proposed*	34.4	18	619.2	0.94
Original [122]*	123.7	18	2226.6	3.41
Radix-16 [94]	42.4	14	593.6	0.91
CORDIC [49]	13.0	195	2535.0	3.88
Software [41]	≈ 23.0	1000	23000.0	35.19
Software library Running at Intel Core(TM) 2 Quad @ 2.66 GHz.				

* 16-digit DXP logarithmic converter.

the digit-recurrence approach proposed in this work is 3.88 times faster than the unit based on the CORDIC approach in terms of latency.

For further analysis, we compare the performance of the proposed architecture with the software approach reported in [41]. The software DFP transcendental function computation library is complied by the Intel C++ Compiler (IA32 version 11.1) [129]. It takes about 1000 clock cycles to compute a Decimal64 logarithm result, running with Intel Core(TM) 2 Quad @ 2.66 GHz microprocessor. The comparison results reported in Table 5.7 show that the proposed hardware implementation in this work is about 35.19 times faster than the software implementation.

To analyze various characters of the proposed architecture for three different formats, we construct the hardware implementations for Decimal32, Decimal64 and Decimal128 formats respectively. Since the digit-width of the input decimal significand are 7-digit, 16-digit and 34-digit (q-digit) in three DFP formats respectively, there is a need to keep at least 11-digit, 20-digit and 38-digit ($q+g+2$-digit) precision for the data-path in the stage 1 and stage 2 in order to guarantee the correct selection of digits e_j during 10, 19 and 37 number of clock cycles. To analyze the optimized size of look-up tables in the stage 3 of the proposed architecture, we efficiently reallocate values stored in the look-up tables using multiple tables approach. Since $2q+3$-digit values of $-\log_{10}(1+e_j 10^{-j})$ need to be stored in look-up table

II for accumulating the logarithm results during the iterations $j \leq h$, where $h = 8$, 17 and 35 for three DFP formats; also the number of 19 different $2q+3$-digit values of $-e_j/\ln(10)$ need to be stored in the look-up table III, the optimized size of look-up tables is about 1.36 KByte, 5.58 KByte and 22.5 KByte for three DFP formats. To achieve the q-digit decimal significand ($R_{significand}$) by the decimal compound adder in the stage 4, there is a need to keep at least 11-digit, 20-digit and 38-digit ($q+4$-digit) precision for the data-path in the stage 3 for three DFP formats.

5.6 Summary

Most previous decimal logarithmic converters are constructed by using either a binary logarithmic converter or decimal software method. As such, they can only process the binary operand or compute the decimal logarithm operation in a slow software speed. In this work, we present a DFP logarithmic converter that is based on the digit-recurrence algorithm with selection by rounding. We develop the radix-10 algorithm, improve the architecture, and implement it with the STM 90-nm CMOS standard cells library. The implementation results show that the improved architecture is 3.62 times faster than our previous design [122] in terms of the latency.

To provide more reference for floating-point-unit designers, we compare the proposed architecture with a binary radix-16 converter [94] and the results show that the latency of the proposed radix-10 decimal logarithmic converter is close to that of the binary radix-16 converter. Moreover, we compare the proposed architecture with a recent high performance implementation based on the decimal CORDIC algorithm [49]. Although a comparison between two different algorithms may depend on too many parameters, considering the timing reported in the two implementations, we are quite confident to conclude that the design presented in this chapter shows a latency 3.88 times shorter than that of the unit based on the CORDIC algorithm. In addition, compared with the software DFP transcendental function computation library [41], the proposed hardware implementation in this work is about 35.19 times faster than the software implementation.

Chapter 6

Decimal Floating-Point Antilogarithmic Converter

In this chapter, we present the algorithm and architecture of the decimal floating-point (DFP) antilogarithmic converter, based on digit-recurrence algorithm with selection by rounding. The proposed converter can compute faithful DFP antilogarithm results for any one of the three DFP formats specified in the IEEE 754-2008 standard. The proposed architecture is synthesized with STM 90-nm standard cell library and synthesis results show that the critical path delay and latency of the proposed Decimal64 antilogarithmic converter are 1.26 *ns* (28.0 FO4) and 19 clock cycles respectively; and the total hardware complexity is 30760 NAND2 gates. The delay estimation results of the proposed architecture show that its latency is 1.44 times faster that of the binary radix-16 exponential converter, and it has a significant decrease on latency in contrast to a recently published high performance CORDIC implementation.

6.1 Introduction

In this chapter, a radix-10 digit-recurrence algorithm based on the selection by rounding approach is presented to implement the DFP antilogarithmic converter in order to achieve faithful antilogarithm results of DFP operands, specified in IEEE 754-2008 standard. This work is an improved design of the research presented in [130]. The proposed architecture presented in this chapter can be easily modified to implement a DFP exponential converter. This design makes the first attempt to analyze and implement a DFP antilogarithmic converter that can compute the DFP based-10 antilogarithm operation specified in the IEEE

754-2008 standard.

This chapter is organized as follows: In Section 6.2, we give an overview of the DFP antilogarithm operation. Section 6.3 presents the proposed algorithm and error analysis for a DFP antilogarithm computation. Section 6.4 describes the improved architecture based on the redundant data-path with details of its hardware implementation. In Section 6.5, first, we analyze the area-delay evaluation results of the proposed architecture; second, we compare the performance of the proposed design with the binary radix-16 exponential converter [57], our original design [130], the recent decimal CORDIC design [49], and the software implementation [41] in terms of the latency; third, we discuss the various characters of the DFP antilogarithmic converter for the three different DFP formats. Section 6.6 gives conclusions.

6.2 DFP Antilogarithm Operation

6.2.1 Exception Handling

A valid DFP antilogarithm calculation is defined as:

$$R = Anti\log_{10}(v) = 10^{v} \tag{6.1}$$

v is a DFP number belongs to any of the three DFP interchange formats. There are some exceptional cases need to be dealt with during the DFP antilogarithm computation.

- if v is a NaN, the DFP antilogarithm operation returns NaN and signals the invalid operation exception;
- if v is a positive infinite operand, the antilogarithm result simply returns $+\infty$, and if v is a negative infinite operand, the antilogarithm result simply returns $+0$ with no exception.
- when the input DFP operand is in the range of $(\log_{10}(|v_{\max}|), +\infty]$, the antilogarithm result satisfies the condition of overflow and returns the maximum representable DFP operand or $+\infty$ based on different rounding modes.
- when the input DFP operand is in the range $[-\infty, \log_{10}(|v_{\min}|))$, the antilogarithm result satisfies the condition of underflow that rounds the intermediate result down to zero or to the minimum representable DFP number based on different rounding modes.

- if the DFP antilogarithm result is inexact, the DFP antilogarithm operation signals the inexact exception;
- Only if the DFP operands are in the range of $[\log_{10}(|v_{\min}|), \log_{10}(|v_{\max}|)]$, a normal DFP antilogarithm computation takes place. The rest of this chapter details the computation on this interval in particular.

6.2.2 Range Reduction

Since v is in the range of $[\log_{10}(|v_{\min}|), \log_{10}(|v_{\max}|)]$, the DFP antilogarithm operation can be transformed to a DXP antilogarithm computation as:

$$R = Anti\log_{10}(v) = 10^{v_{\text{int}} \cdot v_{frac}} = 10^{v_{\text{int}}} \times 10^{v_{frac}} \tag{6.2}$$

In (6.2), v_{int} is a n-digit decimal integer number in the range of $[emin-q+1, emax]$, where n equals to 3, 3 and 4 for Decimal32, Decimal64 and Decimal128 formats respectively. The v_{int} plus the value of k represents the *real exponent* of the DFP antilogarithm result, where k is achieved from normalizing the DXP result of $10^{v_{frac}}$ to the decimal integer *significand* of the DFP antilogarithm result. Since the valid value of v could be very close to zero, the fraction number v_{frac} can be represented as several leading zeros plus the q-digit decimal significand, $v_{frac} = \pm 0.00...00 z_1 z_2 ... z_{q-1} z_q$. Therefore, v_{frac} is a decimal fraction number in the range of $(-1, 1)$, which can be completely represented by at most $emin-q+1$-digit or at least $q-n$-digit. The result of $10^{v_{frac}}$ can be normalized to a decimal integer *significand* of the DFP antilogarithm result.

Since the target is a DFP computation, the DXP antilogarithm computation, $10^{v_{frac}}$, should be able to achieve enough accuracy to guarantee a faithful rounding for the DFP antilogarithm result. First, the results of $10^{v_{frac}}$ are in the range of $(0.1, 10)$, so the q-digit accurate DXP antilogarithm results are enough to represent the exact q-digit decimal *significand* of the DFP antilogarithm results. Second, since it is impossible to keep whole digit-width of v_{frac} in the hardware implementation when the value of v is very close to zero, v_{frac} is rounded to at least $q+1$-digit v'_{frac} so that we can still guarantee a faithful rounding for the DFP antilogarithm result. The proof of determining $q+1$-digit width of v'_{frac} is given in the Section 6.3.4. In the following, we focus on the algorithm and the architecture of

the q+1-digit DXP decimal antilogarithmic converter which can produce the q-digit faithful decimal *significand* of the DFP antilogarithm result.

6.3 Digit-Recurrence Algorithm for Antilogarithm

6.3.1 Overview of Algorithm

A digit-recurrence algorithm to calculate $10^{v'_{frac}}$ is summarized as follows:

$$\lim_{j\to\infty}\{v'_{frac} - \sum_{j=1}^{j}\log_{10}(f_j)\} \to 0 \tag{6.3}$$

If (6.3) is satisfied, then,

$$\lim_{j\to\infty}\{\sum_{j=1}^{j}\log_{10}(f_j)\} \to v'_{frac} \tag{6.4}$$

Thus,

$$10^{v'_{frac}} = \prod_{j=1}^{\infty} f_j \tag{6.5}$$

f_j is defined as $f_j = 1+e_j10^{-j}$ by which v'_{frac} is transformed to 0 by successive subtraction of $\log_{10}(f_j)$. As such, f_j allows the use of a decimal *shift-and-add* implementation.

According to (6.4) and (6.5), the corresponding recurrences for transforming v'_{frac} and computing the antilogarithm are presented as follows:

$$L[j+1] = L[j] - \log_{10}(1 + e_j10^{-j}) \tag{6.6}$$

$$E[j+1] = E[j] \times (1 + e_j10^{-j}) \tag{6.7}$$

In (6.6) and (6.7), $j \geq 1$, $L[1] = v'_{frac}$ and $E[1] = 1$. The digits e_j are selected so that $L(j+1)$ converges to 0. After performing the last iteration of recurrence, we obtain:

$$L[j+1] \approx 0 \tag{6.8}$$

$$E[j+1] \approx 10^{v'_{frac}} \tag{6.9}$$

Although the proposed algorithm is designed for the computation of antilogarithm results in base 10, the antilogarithm results in any base β can also be achieved by simply changing

the base '10' to 'β' as shown in (6.11):

$$L[j+1] = L[j] - \log_{\beta}(1 + e_j 10^{-j}) \tag{6.10}$$

Thus, after performing the last iteration of recurrence, the final results are transformed to:

$$E[N+1] \approx \beta^{v'_{frac}} \tag{6.11}$$

To obtain the selection function for e_j, a scaled remainder is defined as:

$$W[j] = 10^j \times L[j] \times \gamma \tag{6.12}$$

Where, γ is defined as a scaled constant. Thus,

$$L[j] = W[j] \times 10^{-j} \times \gamma^{-1} \tag{6.13}$$

To substitute (6.13) into (6.6):

$$W[j+1] = 10W[j] - 10^{j+1} \times \gamma \times \log_{10}(1 + e_j 10^{-j}) \tag{6.14}$$

According to (6.14), the results in numerical analysis show that when the value of γ is equal to a 2-digit constant 2.3, the digits e_j are selected as a function of leading digits of scaled remainder in a way that the residual $W[j]$ remains bounded.

6.3.2 Selection by Rounding

The selection of the digits e_j are achieved through rounding to the integer part of the scaled residual. To reduce the delay of selection function, the rounding is preformed on an estimate $\widehat{W}[j]$, which is obtained by truncating $W[j]$ to t fractional digits. The selection function is indicated as:

$$e_j = round(\widehat{W}[j]) \tag{6.15}$$

To allow for the use of estimates in the selection function, a redundant digit set is used for the digit e_j. In this work, the selection by rounding is performed with the maximum redundant set $e_j \in \{-9, -8, ..., 0, ...8, 9\}$.

Since $|e_{j+1}| \leq 9$,

$$-9.5 < \widehat{W}[j+1] < 9.5 \tag{6.16}$$

thus,

$$-9.5-10^{-t}<W[j+1]<9.5+10^{-t} \tag{6.17}$$

and,

$$-0.5-10^{-t}<W[j]-e_j<0.5+10^{-t} \tag{6.18}$$

(6.14) can be represented as:

$$W[j+1]=10(W[j]-e_j)-10^{j+1}\times 2.3\times\log_{10}(1+e_j10^{-j})+10e_j \tag{6.19}$$

According to (6.17), (6.18) and (6.19), the numerical analysis is processed as follows:

$$10^{j+1}\times 2.3\times\log_{10}(1+e_j10^{-j})-10e_j>-4.5+9\times10^{-t} \tag{6.20}$$

$$10^{j+1}\times 2.3\times\log_{10}(1+e_j10^{-j})-10e_j<4.5-9\times10^{-t} \tag{6.21}$$

The results in numerical analysis show that if and only if $j\geq 3$, $t\geq 1$ the conditions (6.20) and (6.21) are satisfied. In doing so, the selection by rounding is only valid for iterations $j\geq 3$, and e_1 and e_2 can be only achieved by look-up tables. However, using two look-up tables for $j=1,2$ will significantly increase the overall hardware implementations. Therefore, the restriction for e_1 is defined so that e_2 can be achieved by selection by rounding and one look-up table will be saved. Because $W[1]=10\times 2.3\times v'_{frac}$, $W[2]$ can be achieved as:

$$W[2]=230\times v'_{frac}-10^2\times 2.3\times\log_{10}(1+e_1 10^{-1}) \tag{6.22}$$

When the value of j equates to 2, the value of e_2 is in the range of $-9<e_2<9$ so that (6.20) and (6.21) are satisfied. Substituting $-9<e_2<9$, and $t=1$ in (6.18) yields:

$$-8.6<W[2]<8.6 \tag{6.23}$$

According to (6.22) and (6.23), we obtain:

$$230\times v'_{frac}-10^2\times 2.3\times\log_{10}(1+e_1 10^{-1})<8.6 \tag{6.24}$$

$$230\times v'_{frac}-10^2\times 2.3\times\log_{10}(1+e_1 10^{-1})>-8.6 \tag{6.25}$$

To satisfy the conditions (6.24) and (6.25), we obtain that when e_1 is selected from -9 to 9, the input number v'_{frac} is in the range of $[-1.03, 0.31]$. Therefore, 1) the look-up table I is

Table 6.1: Digit e_1 selection of DFP antilogarithm.

The range of v'_{farc}	e_1 (BCD)	The range of v'_{farc}	e_1 (BCD)
$[-0.02, -0.00]$	-0.0(00000000)	$[-0.49, -0.55]$	-7.0(00110000)
$[-0.03, -0.07]$	-1.0(10010000)	$[-0.56, -0.61]$	-7.4(00100110)
$[-0.08, -0.12]$	-2.0(10000000)	$[-0.62, -0.66]$	-7.7(00100011)
$[-0.13, -0.18]$	-3.0(01110000)	$[-0.67, -0.72]$	-8.0(00100000)
$[-0.19, -0.24]$	-4.0(01100000)	$[-0.73, -0.77]$	-8.2(00011000)
$[-0.25, -0.28]$	-4.5(01010101)	$[-0.78, -0.82]$	-8.4(00010110)
$[-0.29, -0.32]$	-5.0(01010000)	$[-0.83, -0.88]$	-8.6(00010100)
$[-0.33, -0.37]$	-5.5(01000101)	$[-0.89, -0.94]$	-8.8(00010010)
$[-0.38, -0.42]$	-6.0(01000000)	$[-0.95, -0.98]$	-8.9(00010001)
$[-0.43, -0.48]$	-6.5(00110101)	$[-0.99, -1.00)$	-9.0(00010000)

constructed to cover all the negative input numbers v'_{frac} in the range $(-1, 0]$; 2) in order to tune the positive input numbers v'_{frac}, $(0, 1)$, to negative, the fraction part of positive DFP input operand v'_{frac} should be firstly adjusted to negative by $v'_{frac}-1$ and its corresponding integer part v_{int} is adjusted by $v_{\text{int}}+1$. The results of the numerical analysis show that 1) 1-digit e_1 fails to create a look-up table I for achieving continuous ranges to cover all negative input numbers m. Therefore, e_1 is extended to 2-digit so that all negative input numbers m can be achieved. 2) 2-digit MSDs of the input operand m are necessary to address the initial look-up table for selection of the digit e_1. The look-up table I is constructed by a size of $2^5 \times 8$ in which the values of e_1 are stored as shown in Table 6.1.

6.3.3 Approximation of Logarithm

The values of logarithm $\gamma \times \log_{10}(1+e_j 10^{-j})$ in (6.14) can be achieved by look-up table II, for decimal antilogarithm operation, $\gamma = 2.3$. With the increasing number of iteration, however, the size of the table will become prohibitively larger. Thus, a method for reducing the table size, to achieve a significant reduction in the overall hardware requirement is necessary. A

Taylor series expansion of the logarithm function $\log_{10}(1+x)$ is demonstrated in (6.26):

$$\log_{10}(1+x) = (x - \frac{x^2}{2} + ...)/\ln(10) \tag{6.26}$$

After $j=h$ iterations, the values of $\log_{10}(1+e_j10^{-j})$ can be approximated by $e_j10^{-j}/\ln(10)$. Since we aim to guarantee the correct selection of digits e_j, the series approximation can be used in the iterations when the condition (6.52) is satisfied (refer to Section 6.3.5). Therefore, after the h^{th} iteration, the values of $2.3\times\log_{10}(1+e_j10^{-j})$ do not need to be stored in look-up table II. The values of $2.3\times e_j10^{-j}/\ln(10)$, instead, will be used for approximation. Note that the values of h are equal to 4, 9 and 18 for Decimal32, Decimal64 and Decimal128 formats respectively in the proposed antilogarithm digit-recurrence algorithm.

6.3.4 Error Analysis and Evaluation

The errors in the proposed antilogarithm digit-recurrence algorithm can be produced in three ways. The first type of error is the inherent error of algorithm, ε_i, resulted from the difference between the antilogarithm results obtained from finite iterations and the exact results obtained from infinite iterations. The second one is the inexact input error, ε_v, produced by the difference between antilogarithm results of the inexact input v'_{frac} and the exact input v_{frac}. The third one is the quantization error, ε_q, resulted from the finite precision of the intermediate values in the hardware implementation. In order to obtain a q-digit accuracy faithful antilogarithm result, the following condition must be satisfied:

$$\varepsilon_t = \varepsilon_i + \varepsilon_v + \varepsilon_q \leq 10^{-q} \tag{6.27}$$

Inherent Error of Algorithm

Since each DXP antilogarithm result is achieved after $(q+1)^{th}$ iterations, ε_i can be defined as:

$$\varepsilon_i = \prod_{j=1}^{\infty}(1+e_j10^{-j}) - \prod_{j=1}^{q+1}(1+e_j10^{-j}) \tag{6.28}$$

Thus, equation (6.28) can be written as:

$$\varepsilon_i = \prod_{j=1}^{\infty}(1+e_j10^{-j}) \times (1 - \frac{1}{\prod_{j=q+2}^{\infty}(1+e_j10^{-j})}) \tag{6.29}$$

In (6.29), since the proposed DXP antilogarithm algorithm can compute the input values in the range of $(-1, 0]$, the exact antilogarithm results, obtained from the infinite iterations are in the range of $(0.1, 1]$. In order to use the static error analysis method, we substitute the worst case ($e_j = 9$) and the maximum value of the exact antilogarithm results to (6.29), then the maximum ε_i is obtained:

$$\varepsilon_i \leq 1 - \frac{1}{\prod_{j=q+2}^{\infty}(1 + 9 \times 10^{-j})} \tag{6.30}$$

In (6.30), it is obvious that:

$$\prod_{j=q+2}^{\infty}(1 + 9 \times 10^{-j}) = e^{\sum_{j=q+2}^{\infty} \ln(1+9\times10^{-j})} \tag{6.31}$$

Since the inequation (6.31) is satisfied:

$$\sum_{j=q+2}^{\infty} \ln(1 + 9 \times 10^{-j}) < 9 \times (10^{-q-2} + 10^{-q-3} + ...) \tag{6.32}$$

We obtain:

$$\prod_{j=q+2}^{\infty}(1 + 9 \times 10^{-j}) < e^{9\times(10^{-q-2}+10^{-q-3}+...)} \tag{6.33}$$

Thus, the maximum absolute ε_i is:

$$|\varepsilon_i| < 1 - \frac{1}{e^{9\times(10^{-q-2}+10^{-q-3}+...)}} \approx 1 \times 10^{-q-1} \tag{6.34}$$

Inexact Input Error

As we described in Section 6.2.2, if a DFP operand, v, is very close to zero, the whole digit-width of $v_{frac} = \pm 0.00...00 z_1 z_2 ... z_{q-1} z_q$ could be too long to be kept in the hardware implementation. v_{frac} has to be rounded to at least q+1-digit v'_{frac} in the DXP antilogarithm algorithm. Therefore, the inexact input error can be defined as:

$$\varepsilon_v = 10^{v_{frac}} - 10^{v'_{frac}} \tag{6.35}$$

It is evident that the maximum ε_v is obtained when 1) the v_{frac} consists of q+1-digit leading zeros and q-digit decimal significand; 2) each of decimal significand digit, $z_1, z_2, ..., z_{q-1}, z_q = 9$:

$$\varepsilon_v \leq 10^{\pm 0.\underbrace{00...00}_{q+1}\underbrace{99...99}_{q}} - 10^{\pm 0.\underbrace{00...00}_{q+1}} \tag{6.36}$$

(6.36) can be written as:

$$\varepsilon_v \leq (10^{\pm 0.\underbrace{99...99}_{q}})^{10^{-q-1}} - 1 \tag{6.37}$$

Thus,

$$\log_{10}(1+\varepsilon_v) \leq \pm 0.\underbrace{99...99}_{q} \times 10^{-q-1} \tag{6.38}$$

According to Taylor series expansion of the logarithm function $\log_{10}(1+x)$, we obtain:

$$(\varepsilon_v - \frac{\varepsilon_v^2}{2} + ...)/\ln(10) < \varepsilon_v/\ln(10) \leq \pm 0.\underbrace{99...99}_{q} \times 10^{-q-1} \tag{6.39}$$

Therefore, the maximum absolute ε_v is:

$$|\varepsilon_v| \leq 2.303 \times 10^{-q-1} \tag{6.40}$$

Quantization Error

The DXP antilogarithm results can be obtained by $q+1$ times successive multiplication as shown in (6.41):

$$10^{v'_{frac}} = \prod_{j=1}^{q+1} (1 + e_j 10^{-j}) \tag{6.41}$$

Since only the finite precision of intermediate multiplication results are operated in the hardware implementation, the quantization errors, ε_q, are produced. In this work, we choose the $q+2$-digit as the minimal number of fractional digits (FDs) for each of the intermediate multiplication result.

In (6.41), $e_j 10^{-j}$ are the j-digit decimal fraction numbers, and $e_1 10^{-1}$ is 2-digit. Therefore, the digit-width of the decimal fraction number is accumulated in each iteration, in order to represent the full precision of the intermediate multiplication result. In the initial $j \leq w$ iterations, there is no truncated errors, $\tau = 0$, produced because the fractional digit-width of intermediate multiplication results are smaller than the minimal $q+2$-digit. However, from the $j > w+1$ iterations, the truncated error, $\tau = 10^{-q-2}$, is produced in each iteration. The value of w determined by (6.42), are equal to 3, 5 and 7 for Decimal32, Decimal64 and Decimal128 DFP formats respectively.

$$2 + \sum_{j=2}^{w} j \leq q + 2 \tag{6.42}$$

After the last $q+1$ iteration, the total quantization error, ε_q, can be represented as:

$$\varepsilon_q = \tau \times [\prod_{j=w+1}^{q+1}(1+e_j 10^{-j}) + ... + \prod_{j=q+1}^{q+1}(1+e_j 10^{-j}) + 1] \tag{6.43}$$

According to the same mathematic method as (6.31), (6.32) and (6.33), each successive multiplication in (6.43) satisfies:

$$\tau \times \prod_{j=l}^{q+1}(1+e_j 10^{-j}) < \tau \times e^{\sum_{j=l}^{q+1} e_j 10^{-j}} \tag{6.44}$$

In (6.44), $l = w+1, w+2, ...,$ or $q+1$ for each successive multiplication respectively. Then, the total quantization error, ε_q, satisfies:

$$\varepsilon_q < \tau \times (e^{\sum_{j=w+1}^{q+1} e_j 10^{-j}} + ... + e^{\sum_{j=q+1}^{q+1} e_j 10^{-j}} + 1) \tag{6.45}$$

Considering the worst case ($e_j = 9$) in (6.45), we obtain the maximum absolute ε_q:

$$|\varepsilon_q| < (q-w+2) \times 10^{-q-2} \tag{6.46}$$

If the proposed architecture is created based on the carry-save redundant data-path, there are truncated errors, $\tau = 10^{-q-2}$ from the first $j \leq 1$ iterations, because the fractional digit-width of intermediate multiplication results are represented in the carry-save representation in which the carry may occur in the $(q+2)^{th}$ digit and shifted out of data-path in the first interaction. Thus, for the carry-save redundant data-path, the value of w in (6.46) is zero for any one of the DFP formats.

Error Evaluation

Since the final antilogarithm result has q-digit accuracy, the maximum final rounding error is 0.5 ulp, $\varepsilon_r = \pm 0.5 \times 10^{-q}$. Having ε_i, ε_v, ε_q obtained in (6.34), (6.40) and (6.46) respectively, we achieve the maximum total error ε_t as:

$$|\varepsilon_t| = |\varepsilon_i| + |\varepsilon_v| + |\varepsilon_q| \leq 0.331 \times 10^{-q} + (q-w+2) \times 10^{-q-2} \tag{6.47}$$

We substitute the digit-width of the decimal significand for three different DFP interchange formats, q = 7-digit, 16-digit and 34-digit, into (6.47) respectively. The results indicate

Table 6.2: Error analysis of DFP antilogarithm for DFP interchange formats.

Format Names	**Decimal32**	**Decimal64**	**Decimal128**
Significand (q-digit)	7	16	34
Num. of Iteration (q+1)	8	17	35
Accuracy (q-digit)	7	16	34
FDs-digit (q+2-digit)	9	18	36
Max. Error ($\|\varepsilon_t\| \times 10^{-q}$)	0.421	0.511	0.691

that the maximum total error, ε_t, obtained in three different DFP interchange formats are smaller than the accuracy requirement, 10^{-q} that can satisfies the condition (6.27). Table 6.2 shows the error analysis for three different DFP interchange formats. Since the proposed architecture is implemented based on the redundant carry-save data-path, the value of w is zero for three DFP formats to evaluate ε_q by (6.46). The error analysis in Table 6.2 proves that the proposed decimal antilogarithm algorithm can guarantee q-digit accuracy for the DXP antilogarithm operation, therefore, after the faithful rounding, an exact q-digit decimal significand of DFP antilogarithm results can be achieved.

6.3.5 Guard Digit of Scaled Residual

Since only the finite precision scaled residual $W[j]$ is operated in the hardware implementation, we need to analyze how many guard digits, g, needed to prevent the rounding error of $W[j]$, ε_w, from affecting the correct selection of digits e_j. Since $W[j]$ is converged in the range of $(-9.6, 9.6)$, we define the digit-width of $W[j]$ as q+g+2-digit, consisting of 2-digit integer part and q+g-digit fraction part.

In iterations (j=1) to (j=q+1), because q+g+2-digit rounded values of $-2.3\times\log_{10}(1+e_j10^{-j})$ and $-2.3\times e_j10^{-j}/\ln(10)$ are obtained from look-up table II and look-up table III respectively, the rounding error, $\pm 0.5\times 10^{-q-g}$, is produced in each iteration. The maximum quantization error, ε^1_{wq}, is:

$$|\varepsilon_{wq}| \leq \sum_{j=1}^{q+1} 0.5\times 10^{-q-g} \tag{6.48}$$

The value of $-2.3\times\log_{10}(1+e_j10^{-j})$ is approximated by the value of $-2.3\times e_j10^{-j}/\ln(10)$ in

iterations ($j=h+1$) to ($j=q+1$). However, according to the series expansion of logarithm function in (6.26), an approximation error, ε_{wa}, is produced in each iteration:

$$\varepsilon_{wa} = -2.3\times \sum_{j=h+1}^{q+1} (-\frac{(e_j 10^{-j})^2}{2} + \frac{(e_j 10^{-j})^3}{3} - ...)/\ln(10) \tag{6.49}$$

we keep $(e_j 10^{-j})^2/2\ln(10)$ to analyze ε_{wa}:

$$\varepsilon_{wa} \leq 2.3\times \sum_{j=h+1}^{q+1} (\frac{(e_j 10^{-j})^2}{2})/\ln(10) \tag{6.50}$$

Considering the worst case ($e_j=9$ or -9), we obtain the maximum ε_{wa}:

$$|\varepsilon_{wa}| \leq 4.01 \times 10^{-2h-1} \tag{6.51}$$

Therefore, according to (6.14), after the $(q+1)^{th}$ iteration, the truncation error of $W[j]$, ε_w, is obtained as:

$$|\varepsilon_w| \leq 10^{q+1}\times(|\varepsilon_{wq}|+|\varepsilon_{wa}|) = (0.5q+0.5)\times 10^{1-g} + 4.01\times 10^{q-2h} \tag{6.52}$$

Since, the digit e_j is selected by rounding the scaled residual $\widehat{W}[j]$ to its integer part in each iteration, ε_w needs to satisfy the conditions, $\varepsilon_w < 1$ in order to guarantee the correct selection of digits e_j. To satisfy this condition for three different DFP interchange formats in which q is equal to 7-digit, 16-digit and 34-digit respectively, we obtain that when the values of h are equal to 4, 9 and 18, the guard digit, g, are equal to 2, 2 and 3 for three different DFP interchange formats.

6.4 Architecture of DFP Antilogarithmic Converter

Figure 6.1 shows the architecture of the proposed DFP antilogarithmic converter in the top level. We only detail the architecture for the computation of the sign bit (R_{sign}), the real exponent (R_{exp}) and the decimal significand ($R_{significand}$) of DFP antilogarithm results, since other issues of DFP antilogarithmic converter such as the exception handling, the packing and the unpacking from IEEE 754-2008 DFP format are straightforward. In the data-path of the proposed architecture, the residual $W[j]$ is represented by the $q+g+3$-digit intermediate value (including 1-digit sign, 2-digit integer, q-digit fraction, and g-digit

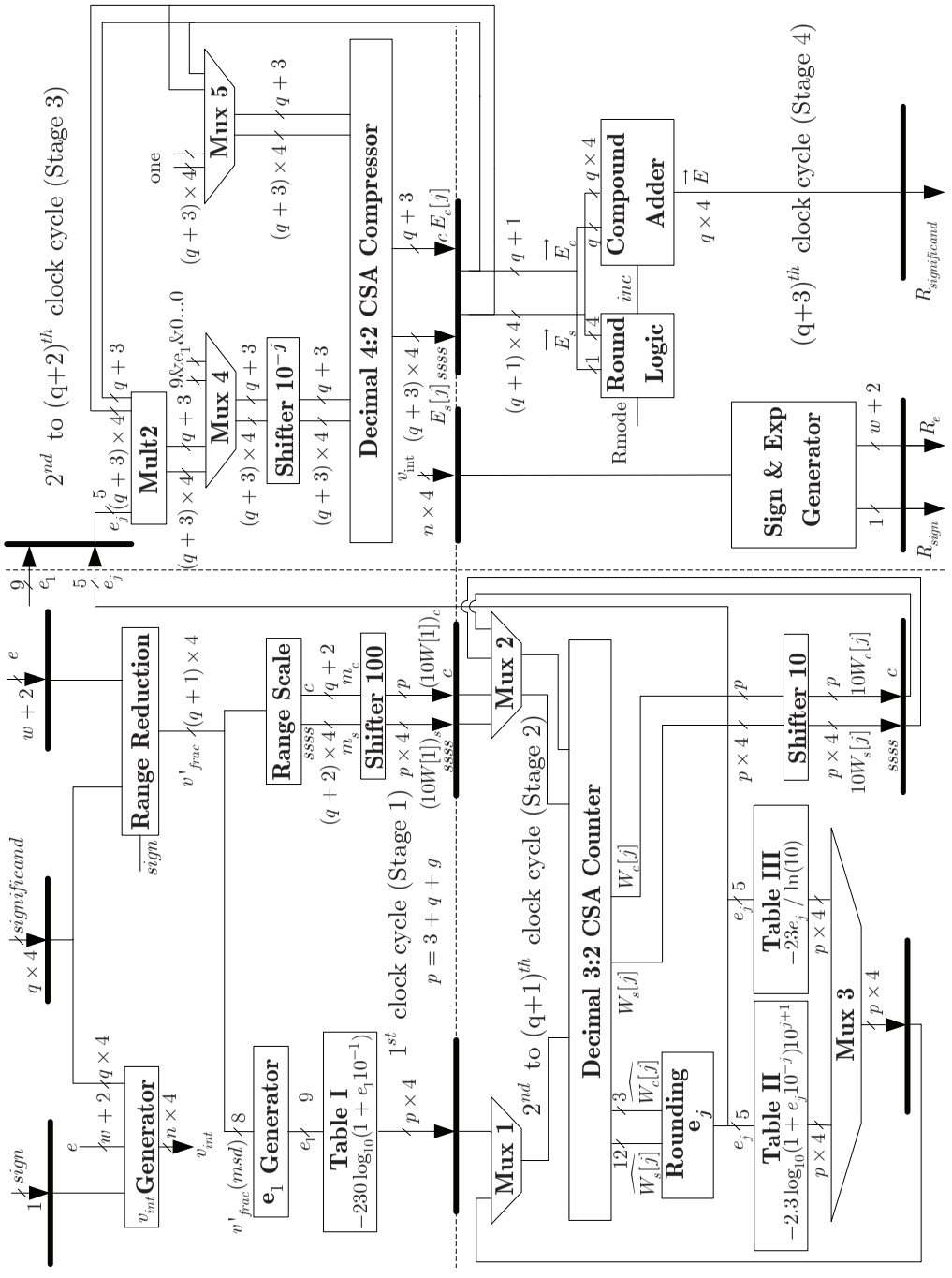

Figure 6.1: Improved architecture of DFP antilogarithmic converter.

guard digit); the decimal significand $E[j]$ is represented by the q+3-digit intermediate value (including 1-digit integer, and q+2-digit fraction); and the digit e_1 is represented by a 9-bit intermediate value (including 1-bit sign and 2-digit 10's complement value of e_1 in the BCD encoding), and the other digits e_j are represented by a 5-bit intermediate value (including

1-bit sign and 1-digit absolute value of e_j). To speed-up the execution of recurrences, all intermediate values in the data-path are represented using the redundant decimal carry-save representation. For example, the residual $W[j]$ with the decimal carry-save representation is shown in Figure 5.2, where $ssss$ represents a 1-digit $W_s[j]$, c represents a 1-bit $W_c[j]$, and $\widehat{W[j]}$ represents an estimated value of the residual $W[j]$ (including 3-digit most significand digits (MSDs) of $W[j]$). As a consequence of this representation, the delay of the addition and the multiply operation in the recurrence are independent of the computational precision.

6.4.1 Datapath

The data-path of the proposed architecture is pipelined and re-timed into four stages in order to minimize and balance the critical path delay. The initial processing stage (stage 1) is to obtain the initial digit e_1, and then the digit recurrence stage (stage 2) is to obtain remaining digits e_j. The antilogarithm computation stage (stage 3) is to achieve the $q+2$-digit intermediate decimal significand of the DFP antilogarithm result. Finally, 1-bit R_{sign}, $w+2$-bit R_{exp} and q-digit $R_{significand}$ of the DFP antilogarithm result are achieved in the final processing stage (stage 4). The cycle-based sequence of operations is summarized as follows:

- Stage 1, In 1^{st} clock cycle (In iteration ($j=1$)):

- The 1-bit sign, $w+2$-bit real exponent, and the q-digit non-normalized decimal significand, as input operands, are obtained from input registers.
- The q-digit decimal significand and $w+2$-bit real exponent are processed in the range reduction logic to achieve the $q+1$-digit DXP operand v'_{frac}. Meanwhile, the value of v_{int} is obtained in the v_{int} generator and sent to the stage 4 by a register.
- If the DFP operand is positive, the fraction part of the DFP operand, v'_{frac}, should be firstly adjusted to a negative fraction number by $v'_{frac}-1$ in a 10's complement converter, then the negative fraction number, $v'_{frac}-1$, is the input of the DXP decimal antilogarithmic converter. Meanwhile, its corresponding integer part, v_{int}, is adjusted by $v_{int}+1$ and sent to the stage 4.
- The digit e_1 is obtained from a look-up table I (refer to Table 6.1) based on the value

of 2-digit MSDs of v'_{frac}.

- The v'_{farc} is multiplied by a 2-digit constant 2.3 in a multiply logic (Mult1) to achieve the q+3-digit value of $m = 2.3 \times v'_{frac}$ with the carry-save representation (m_s, m_c).
- The value of m out from Mult1 is shifted 2-digit to the left to achieve $10W[1]$ ($W[1] = 10 \times 2.3 \times v'_{frac}$); and the value of $-230\log_{10}(1+e_1 10^{-1})$is directly obtained from the look-up table II. Then, values of $10W[1]$ and $-230\log_{10}(1+e_1 10^{-1})$ are sent to the stage 2 by registers.

- Stage 2, From 2^{nd} to $(q+1)^{th}$ clock cycle (In iterations $j=2$ to $j=q+1$):

- In the 2^{nd} clock cycle, the residual $W[j]$ is obtained by adding $10W[1]$ (selected from Mux2) and $-230\log_{10}(1+e_1 10^{-1})$ (selected from Mux3) together in a decimal 3:2 CSA compressor. Then, the digit e_j can be obtained by rounding 3-digit $\widehat{W[j]}$ in a rounding e_j logic.

$$(W_s[j], W_c[j]) = 10W[1] - 230\log_{10}(1+e_1 10^{-1})$$
$$e_j = round(\widehat{W[j]})$$

- The value of $W[j]$ in the carry-save representation is shifted 1-digit to the left to achieve the value of $10 \times W[j]$, which is sent back to Mux2 for the next iteration.
- From the number of $j = 2$ to $j = h^{th}$ iteration, the value of $-2.3 \times 10^{j+1} \times \log_{10}(1 + e_j 10^{-j})10^{j+1}$ is directly obtained from a look-up table III and sent back to Mux1 for the next iteration.

$$(W_s[j+1], W_c[j+1]) = 10W[j] - 2.3 \times 10^{j+1}\log_{10}(1+e_j 10^{-j})$$
$$e_j = round(\widehat{W[j+1]})$$

- From the number of $j = (h+1)^{th}$ to $j = (q+1)^{th}$ iteration, the value of $-2.3 \times 10 \times e_j / \ln(10)$ is obtained from a look-up table IV and sent back to Mux1. Thus,

$$(W_s[j+1], W_c[j+1]) = 10W[j] - 2.3 \times 10 \times e_j / \ln(10)$$
$$e_{j+1} = round(\widehat{W[j+1]})$$

- After the $(q+1)^{th}$ clock cycle, all the digits e_j are achieved with the selection by rounding.

- Stage 3, From 2^{nd} to $q+2^{nd}$ clock cycle (In iterations $j=2$ to $j=q+2$):

- In the 2^{nd} clock cycle, 2-digit e_1 is chosen by Mux4 and complemented with the value of 9 (integer part) and zeros (fractional part) and shifted 1-digit to the right to achieve $e_1 10^{-1}$ in a barrel shifter. Meanwhile, the value of $E[1]=1$ is chosen by Mux5. The decimal significand result $E[2]$ of the first iteration is obtained in the q+3-digit decimal 4:2 CSA compressor.

$$(E_s[2], E_c[2]) = 1 + e_1 10^{-1}$$

- From the 3^{rd} to $(q+2)^{th}$ clock cycle, the intermediate value of $e_j E[j]$ out from a multiply logic (Mult2) is shifted j-digit to the right to obtain $e_j E[j] 10^{-j}$ in a barrel shifter. The value of $E[j]$ is selected by Mux5 for the computation of $E[j+1]$ in the next iteration.

$$((e_j E[j] 10^{-j})_s, (e_j E[j] 10^{-j})_c) = e_j \times E[j] 10^{-j}$$
$$(E_s[j+1], E_c[j+1])) = e_j E[j] 10^{-j} + E[j]$$

After the $(q+2)^{th}$ clock cycle, the q+3-digit decimal significand of DFP antilogarithm results is obtained.

-Stage 4, In $(q+3)^{th}$ clock cycle:

- In the $(q+3)^{th}$ clock cycle, the sum and carry of q+1-digit MSDs of the fractional part of $E[j]_s$ and $E[j]_c$ ($\overrightarrow{E_s}$ and $\overrightarrow{E_c}$) are added together to obtain the value of $\overrightarrow{E}$ in a q-digit decimal compound adder. At the same time, the L is rounded to the faithful decimal significand $R_{significand}$ based on the value inc of the rounding position in a rounding logic. Since we only consider the *roundTiesToEven* mode in this design, the rounding logic generates an increment inc based on:

$$inc = \begin{cases} 1 & \text{if } (r_d > 5 \ or \ (r_d = 5 \ and \ LSB(L) = 1)) \\ 0 & \text{if } (r_d < 5 \ or \ (r_d = 5 \ and \ LSB(L) = 0)) \end{cases}$$

Where r_d represent the LSD of $\overrightarrow{E[j]}$. The other rounding modes can be implemented in a similar manner.

- The $w+2$-bit exponent R_{exp}, and 1-bit sign R_{sign} are obtained in a sign&exponent generator.

Table 6.3: Example of a Decimal64 antilogarithm operation.

$v = (-1)^1 \times 8576308882936892 \times 10^{-16}$, $R = 10^{v_{\text{int}}.v_{frac}} = 10^0 \times 10^{-0.8576308882936892}$, $v_{\text{int}} = 0, v_{frac}{}' = -0.85763088829868920$
$\rightarrow e_1 = -8.6$, $m = 2.3 \times v_{frac}{}'$ (m_s=7926448956923414740, m_c=0101000000001100100), $h = 9$, $g = 2$

	$\widehat{W[j]}$		$W[j]$		$E[j]$
$W_s[1]$			979.264489569234147400		**In 1^{st} clock cycle (1^{st} iteration)**
$W_c[1]$			001.010000000011001000		
$10W_s[1]$			792.644895692341474000		
$10W_c[1]$			010.100000000110010000	$E_s[1] =$	1.000000000000000000
$-2.3 \times \log_{10}(1 + e_1 10^{-1}) \times 10^2$			+ 196.390551794005254100	$E_c[1] =$	0.000000000000000000
$W_s[2]$	980		898.034346386456638100	$(E[1]e_1 10^{-1})_s$	9.140000000000000000
$W_c[2]$	011		101.101101100000100000	$(E[1]e_1 10^{-1})_c$	+ 0.000000000000000000
$\widehat{W[2]}$	−08	$\rightarrow$	$e_2 = -1$	$E_s[2] =$	0.140000000000000000
$10W_s[2]$			980.343463864566381000	$E_c[2] =$	0.000000000000000000
$10W_c[2]$			011.011011000001000000		
$-2.3 \times \log_{10}(1 + e_2 10^{-2}) \times 10^3$			+ 010.039052425635195000		**In 2^{nd} clock cycle (2^{nd} iteration)**
$W_s[3]$	013		001.383426289192476000	$(E[2]e_2 10^{-2})_s$	9.887488888888888888
$W_c[3]$	000		000.010101001010100000	$(E[2]e_2 10^{-2})_c$ +	0.111111111111111111
$\widehat{W[3]}$	+13	$\rightarrow$	$e_3 = +1$	$E_s[3] =$	9.038599999999999999
$10W_s[3]$			013.834262891924760000	$E_c[3] =$	1.100000000000000000
$10W_c[3]$			000.101010010101000000		
$-2.3 \times \log_{10}(1 + e_3 10^{-3}) \times 10^4$			+ 990.026217975671270000		**In 3^{rd} clock cycle (3^{rd} iteration)**
...	...		...	...	...
$W_s[10]$	021		002.199071104000000000	$(E[9]e_9 10^{-9})_s$	9.999999984725084930
$W_c[10]$	010		001.001011000000000000	$(E[9]e_9 10^{-9})_c$ +	0.000000011111110111
$\widehat{W[10]}$	+31	$\rightarrow$	$e_{10} = +3$	$E_s[10] =$	0.028782483451741196
$10W_s[10]$			021.990711040000000000	$E_c[10] =$	0.110011011010010000
$10W_c[10]$			010.010110000000000000		
$-2.3 \times 10 \times e_{10}$ / ln(10)			+ 970.033680748675623892		**In 10^{th} clock cycle (10^{th} iteration)**
$W_s[11]$	019		001.933401788675623892		
$W_c[11]$	001		000.101100000000000000		
$\widehat{W[11]}$	+20	$\rightarrow$	$e_{11} = +2$		
$10W_s[11]$			019.334017886756238920		
$10W_c[11]$			001.011000000000000000		
$-2.3 \times 10 \times e_{11}$ / ln(10)			+ 980.022453832450415928		**In 11^{th} clock cycle (11^{th} iteration)**
...	...		...	...	...
$W_s[17]$	036		993.644904860602385560	$E_s[17] =$	9.027693494806399868
$W_c[17]$	011		111.110100110111100000	$E_c[17] =$	1.111110000100001100
$\widehat{W[17]}$	+47	$\rightarrow$	$e_{17} = +5$		**In 17^{th} clock cycle (17^{th} iteration)**
In 18^{th} clock cycle				$(E[17]e_{17}10^{-17})_s$	9.999999999999999906
				$(E[17]e_{17}10^{-17})_c$+	0.000000000000000100
				$E_s[18] =$	0.138682384895290964
				$E_c[18] =$	0.000111110011110010

In 19^{th} clock cycle

$v_{int} = 0$ $\quad R_{\text{exp}} = v_{\text{int}} - 16 = -16$ $\quad \leftarrow \quad \vec{E} = .1387934949064010 \quad \leftarrow$

$R_{sign} = 0$ $\quad R_{\text{exp}} = -16$ ("1111110000") $\quad R_{significand} = 1387934949064010$

$\overrightarrow{E_s} =$ 13868238489529096 $\leftarrow r_d{}'$

$\overrightarrow{E_c} =$ 0001111100111001 $\leftarrow inc$

\+ compound addition 1

Table 6.3 shows some iterations of a 64-bit DFP antilogarithm operation executed in the proposed architecture.

6.4.2 Hardware Implementation

The details of the hardware implementation for the each stage of the proposed DFP antilogarithmic converter are presented in this section.

Initial Processing Stage

Figure 6.2 shows the details of the hardware implementation of stage 1.

-In the stage 1: The range reduction logic consists of a decimal bi-directional barrel shifter, an encoder, a 10's complement converter, a 9's complement converter, a trailing zeros detector and a 3-to-1 multiplexer. The decimal barrel shifter that can shift a decimal significand by any amount from 0 to $q-1$ digits in two directions is implemented by a $\log_2(q)$ levels of multiplexors. The shift amount is achieved by the shift amount detector based on the value of exponent $e+16$. To decrease the carry propagation in the 10's complement conversion, the trailing zeros detector is applied to determine the position of the MSD of the trailing continuous zeros. The trailing zeros detector is implemented based on a prefix tree to generate control signals of *sel* for the 3-to-1 multiplexer. The portion of a trailing zeros detector for Decimal64 format is shown in Figure 5.8. The computation of the each digit of v'_{frac} is based on:

$$v'_{frac_i} = \begin{cases} 9\text{'s complement}(\overrightarrow{v'_{frac}}) & \text{if } sign=0 \wedge con_i=1 \wedge con'_i=1 \\ 10\text{'s complement}(\overrightarrow{v'_{farc}}) & \text{if } sign=0 \wedge con_i=1 \wedge con'_i=0 \\ \overrightarrow{v'_{frac}} & \text{otherwise} \end{cases}$$

The v_{int} generator consists of a leading-zero-counter (LZC), a decimal barrel shifter, a encoder, and decimal +1 logic and 3-to-1 multiplexer. First, the leading zeros of the non-normalized decimal significand are counted by the LZC; second, the encoder generates the left shift amount and selection signals for the decimal barrel shifter and the multiplexer respectively; third, the decimal significand is shifted to $e+16-n$-digit to the left, and the n-digit MSD of shifted significand represents the v_{int}. The computation of v_{int} is based on:

$$v_{int} = \begin{cases} 0, & \text{if } e+16-k \leq 0 \\ \lfloor \text{left\_shift}(significand, e+16-n) \rfloor & \text{if } e+16-k>0 \text{ and } sign=1 \\ \lfloor \text{left\_shift}(significand, e+16-n) \rfloor + 1 & \text{if } e+16-k>0 \text{ and } sign=0 \end{cases}$$

Note that the overflow and underflow of the DFP antilogarithm operation can be detected by the v_{int} generator, and the implementation of the detection of them is straightforward.

The e_1 generator is implemented based on a look-up table according to the Table 6.1. After e_1 is obtained, the look-up table I stores the number of 20 values of $q+g+3$-digit

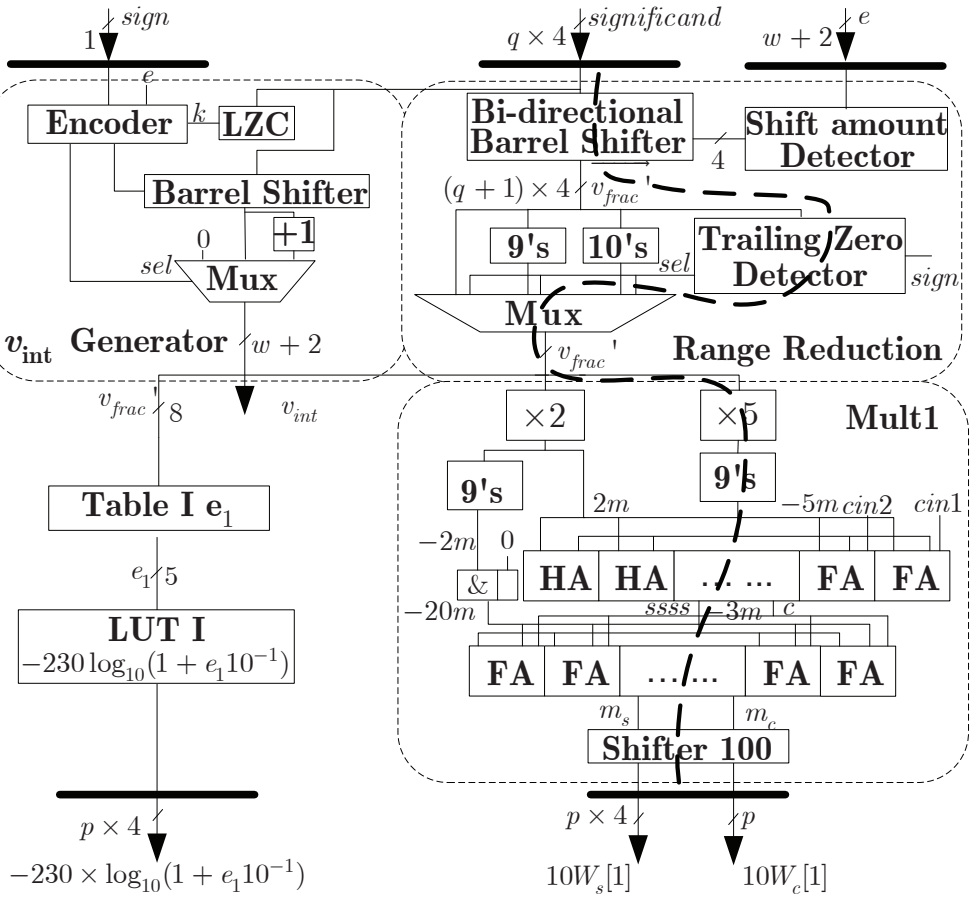

Figure 6.2: Hardware implementation of Stage 1 in DFP antilogarithmic converter.

$-230\times\log_{10}(1+e_1 10^{-1})$. Since g is equal to 2, 2 and 3 for three different interchange formats, the size of the look-up table I is $2^5\times$ 48-bit, $2^5\times 84$-bit and $2^5\times 160$-bit for Decimal32, Decimal64 and Decimal128 respectively.

The multiply logic (Mult1) is applied to compute the value of $m = 2.3\times v'_{frac}$, where v'_{frac} is a $q+1$-digit negative value in the range of $-1 < m \leq 0$. The Mult1 is implemented based on the partial product generation logic presented in [23]. The multiples are formed by adding two of a initial multiple set $-3m$ (achieved by adding -5m and 2m in a 3:2 CSA counter) and $-20m$ (achieved by shifting 1-digit to the right of $-2m$). Both $2m$ and $5m$ can be generated with only a few logic delays, since there is no carry propagation beyond the next more significant digit. The boolean equations for generating double and quintuple of the BCD number are presented in [125]. To decrease the delay of the addition, two levels of decimal CSA adders are implemented to develop multiples m_s and m_c. The boolean equation

for computing 1-digit decimal addition of the BCD number is presented in [20]. The signals *cin1* and *cin2* are generated to supplement the LSD due to the 9's complement conversion ($-2m$ and $-5m$). The signal *cin1* and *cin2* are added in the LSD and the second LSD of the first level of CSA adders respectively.

Digit Recurrence Stage

Figure 6.3 shows the details of the hardware implementation of the digit recurrence stage (stage 2).

- In the stage 2: The 3:2 decimal CSA compressor is implemented by one level of q+g+3-digit 3:2 CSA counter in order to achieve the residual ($W_s[j]$, $W_c[j]$). Then, 1-digit sign (S_s, S_c), 1-digit integer part(I_s, I_c) and 1-digit fraction part ($f_{s_{MSD}}$, $f_{c_{MSB}}$) of the residual are sent to the rounding e_j logic for selecting digits e_j by rounding the residual ($\widehat{W_s[j]}$, $\widehat{W_c[j]}$). The sign of the digit e_j is obtained by the sign detector block which is implemented based on the equation:

$$sign = (S_s^0 \oplus S_c) \oplus (I_s^3 \wedge I_s^0 \wedge I_c)$$

The 1-digit fraction ($f_{s_{MSD}}$, $f_{c_{MSB}}$) and the value of 5 are added together in the 1-digit decimal full adder to generate the signal *carry* to determine the rounding operation. The signals *carry* and *sign* are sent to selection generator to achieve a control signal *sel* of a 4-to-1 multiplexer. The value of $|e_j|$ is achieved in four parallel full adders by adding the value of 0, 1, 6 and 5 with the signals of $f_{s_{MSD}}$ and $f_{c_{MSB}}$ respectively:

$$|e_j| = \begin{cases} I_s + I_c + 0 & \text{if } sign = 0 \wedge carry = 0 \\ I_s + I_c + 1 & \text{if } sign = 0 \wedge carry = 1 \\ \overline{I_s + I_c + 6} & \text{if } sign = 1 \wedge carry = 0 \\ \overline{I_s + I_c + 5} & \text{if } sign = 1 \wedge carry = 1 \end{cases}$$

Thus, the digit e_j is obtained by concatenating 1-bit *sign* with 1-digit $|e_j|$.

The look-up table II stores all the values of the q+g+3-digit $-2.3 \times 10^{j+1} \times \log_{10}(1+e_j 10^{-j})$, where the number of iteration j is in the range of $1 \leq j \leq h$. Since the digit e_j is in the range of $-9 \leq e_j \leq 9$, there are 18 different values (except for the value when $e_j = 0$) that

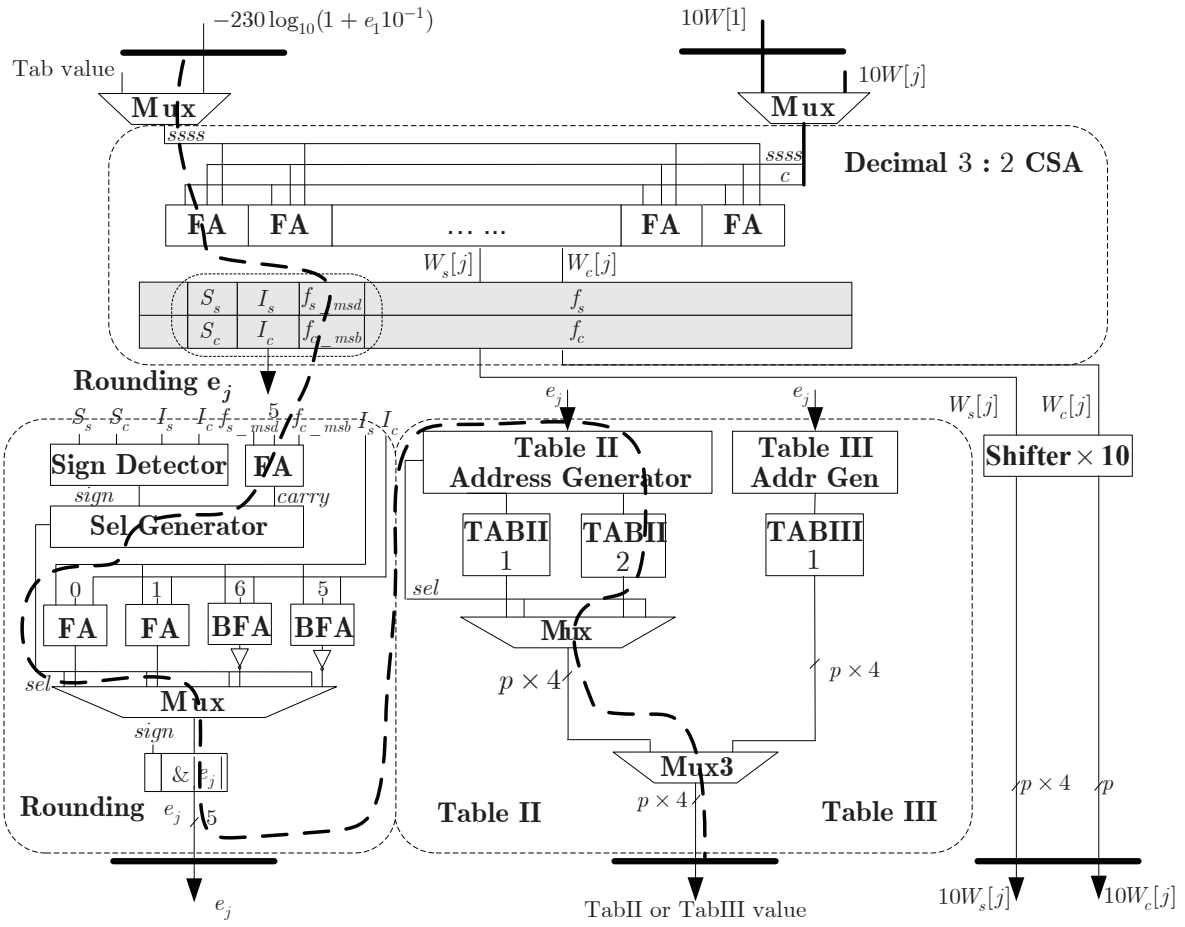

Figure 6.3: Hardware implementation of Stage 2 in DFP antilogarithmic converter.

need to be stored in the look-up table for each iteration. Since the address of the look-up table in hardware must be restricted to a power of two, the size of the look-up table II can be obtained as:

$$\text{TabII size}=2^{\lceil \log_2[18\times(h-1)]\rceil}\times(q+g+3)\times 4 \text{ bit}$$

Since the values of h are equal to 4, 9 and 18, the guard digit for three different interchange formats. Thus, the size of the look-up table II is $2^6\times$ 48-bit, $2^8\times$84-bit and $2^9\times$160-bit for Decimal32, Decimal64 and Decimal128 respectively.

To reduce the size and delay of look-up table II, the values of q+g+3-digit $-2.3\times10^{j+1}\times\log_{10}(1+e_j10^{-j})$ can be efficiently reallocated in multiple tables. For Decimal64 (the example shown in Figure 6.3), the single look-up table II is relocated into the following two parts: 1) the first part (TabII 1) stores all the values of $-2.3\times10^{j+1}\times\log_{10}(1+e_j10^{-j})$, when $2\leq j\leq 9$ and

$e_j = \pm 1$; 2) the second part (TabII 2) stores stores the values when $2 \leq j \leq 9$, and $2 \leq e_j \leq 9$ and $-9 \leq e_j \leq -2$. The sizes of the TabII 1 and TabII 2 are $2^4 \times 84$ and $2^7 \times 84$ respectively. Thus, the optimized size of look-up table II is reduced from 2.64 KByte (single table) to only 1.48 KByte (multiple tables). The look-up table III stores 19 values of $q+g+3$-digit $-2.3 \times e_j / \ln(10) \times 10$, and it is implemented by a size of $2^5 \times 84$-bit table. Thus, the total optimized size of look-up tables (including the look-up table I, II, and III) is about 2.14 KByte for Decimal64. The implementations of address generators to address the look-up table II and the look-up table III based on the values of j and e_j are straightforward.

Antilogarithm Computation Stage

Figure 6.4 shows the details of the hardware implementation of the antilogarithm computation stage (stage 3).

- In the stage 3: The multiply logic (Mult2) is applied to compute the value of $E_s[j]e_j + E_c[j]e_j$, where e_j is a value in the range of $-9 \leq e_j \leq 9$. The multiple of $E_s[j]e_j$ is formed by adding two of a initial multiple set $\{m', -m', 2m', -2m', 5m', -5m', 10m', -10m'\}$ (selecting by signals *sel1* and *sel2* generated by a recorder). The implementation of $4m$ logic can be generated via connecting two $2m$ logics in series. The signals *cin1* and *cin2* are generated by a recorder to supplement the LSD due to the 9's complement conversion. The signal *cin1* and *cin2* are added in the LSD of the two levels of CSA adders respectively.

Since each bit of $E_c[j]$ is only zero or one, the signal of $E_c[j]\,|e_j|$ can be achieved in a carry extend block which can be implemented by a series of logical-AND gates. If the digit $e_j < 0$ ($sign(e_j) = 1$), the signal of $E_c[j]e_j$ is obtained by the 9's complement conversion of $E_c[j]\,|e_j|$, and then the signal $sign(e_j$ is supplemented in the LSD of the signal $(e_j E[j])_c$, otherwise, the signal of $E_c[j]e_j$ is directly obtained from the signal of $E_c[j]\,|e_j|$.

$$(E_c[j]e_j)_i^{3:0} = \begin{cases} E_c[j]_i \wedge e_j^{3:0} & \text{if } sign(e_j) = 0 \\ 9\text{'s } com(E_c[j]_i \wedge e_j^{3:0}) & \text{if } sign(e_j) = 1 \end{cases}$$

Thus, the value of $E[j]e_j$ ($(e_j E[j])_s$, $(e_j E[j])_c$) are achieved by adding the $E_s[j]e_j$ and $E_c[j]e_j$ in a decimal CSA adder. Finally, the signals of $(E[j]e_j 10^{-j})_s$ and $(E[j]e_j 10^{-j})_c$ are obtained in a decimal barrel shifter. The 4:2 decimal CSA compressor, applied to add the values of

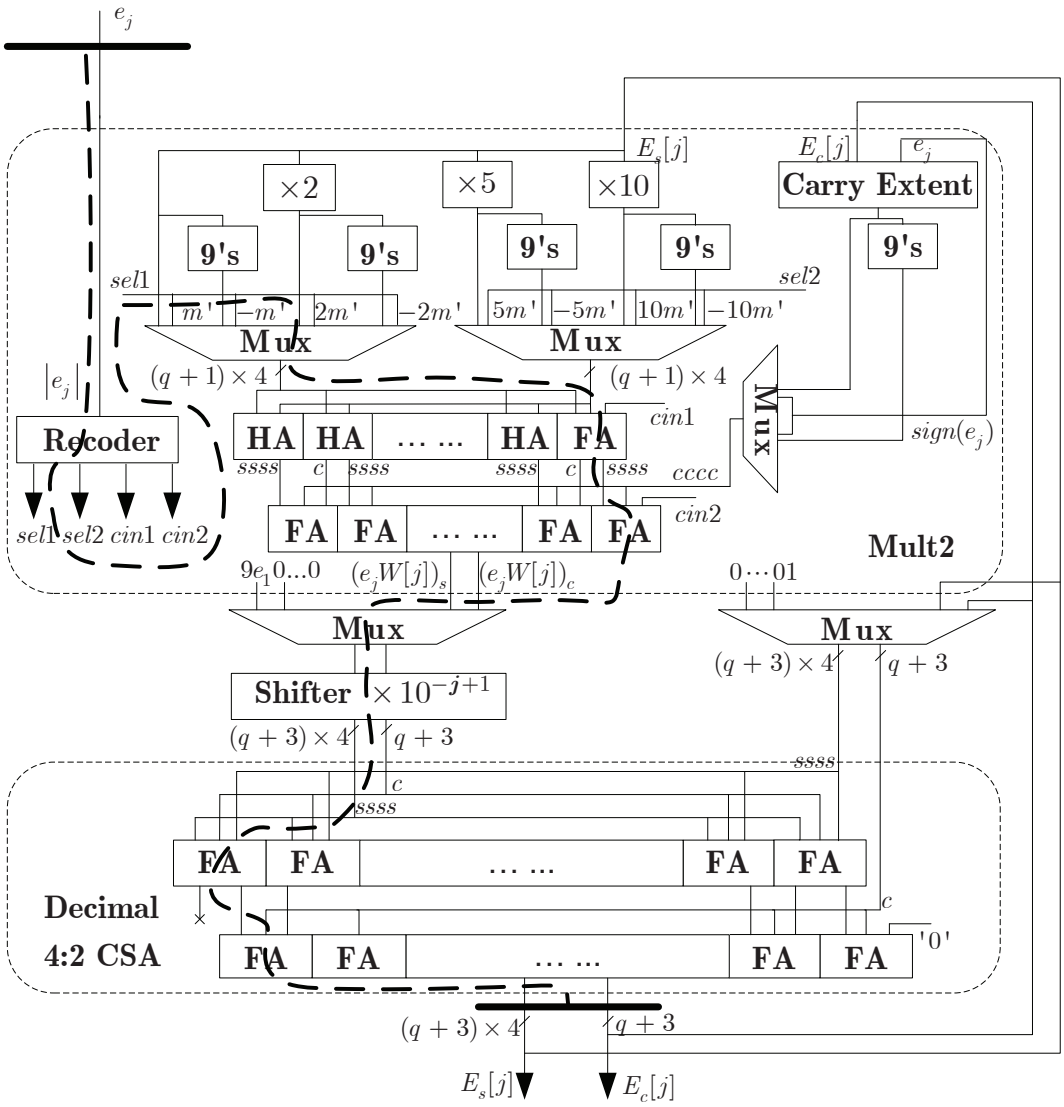

Figure 6.4: Hardware implementation of Stage 3 in DFP antilogarithmic converter.

$E[j]$ ($E_s[j]$, $E_c[j]$), and $E[j]e_j10^{-j}$ ($(E[j]e_j10^{-j})_s$, $(E[j]e_j10^{-j})_c$) together is implemented by two levels of q+3-digit 3:2 CSA counters.

Final Processing Stage

Figure 6.5 shows the details of the hardware implementation for the final processing stage (stage 4).

- In the stage 4: The decimal compound adder is implemented based on the conditional speculative method[105]. A prefix tree based on the binary Kogge-Stone network [126] is used to generate carries into each digit of the decimal addition. The portion of the prefix

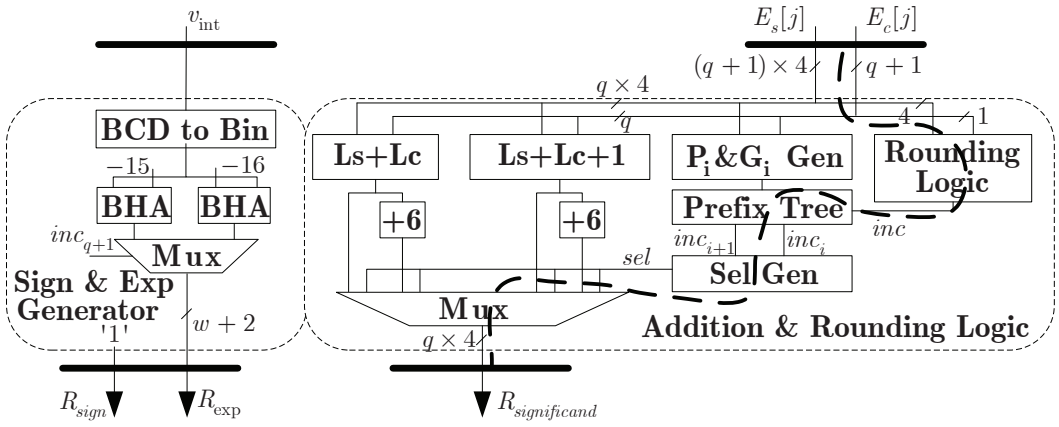

Figure 6.5: Hardware implementation of Stage 4 in DFP antilogarithmic converter.

tree for the Decimal64 format, and notes of the tree (P_i and G_i obtained from the P_i&G_i generator) are shown in Figure 5.7. The carries generated by the prefix tree (inc_i and inc_{i+1}) are used to select the correct addition result of the each digit under the following conditions:

$$\overrightarrow{E_i} = \begin{cases} \overrightarrow{E_{s_i}} + \overrightarrow{E_{c_i}} & \text{if } inc_i = 0 \wedge inc_{i+1} = 0 \\ \overrightarrow{E_{s_i}} + \overrightarrow{E_{c_i}} + 1 & \text{if } inc_i = 1 \wedge inc_{i+1} = 0 \\ \overrightarrow{E_{s_i}} + \overrightarrow{E_{c_i}} + 6 & \text{if } inc_i = 0 \wedge inc_{i+1} = 1 \\ \overrightarrow{E_{s_i}} + \overrightarrow{E_{c_i}} + 1 + 6 & \text{if } inc_i = 1 \wedge inc_{i+1} = 1 \end{cases}$$

The additions of $\overrightarrow{E_{s_i}}+\overrightarrow{E_{c_i}}$ and $\overrightarrow{E_{s_i}}+\overrightarrow{E_{c_i}}+1$ can be implemented using three binary half adders and a binary full adder connected as a ripple carry chain. The logic for adding the value of 6 is used to compensate the $\overrightarrow{E_j}$ to the correct representation of the BCD encoding, which can be implemented using two binary half adders and two binary full adders. Since the value of $R_{significand}$ may be rounded to the value of 1 ($inc_{q+1}=1$), when it happens, the $R_{significand}$ is directly set as one.

A BCD to binary converter is implemented based on [103] in order to convert the decimal value of v_{int} to the w+2-bit binary format. The w+2 exponent R_{exp} is selected based on the value of inc_{q+1} in a 2-to-1 multiplexer:

$$R_{exp} = \begin{cases} v_{int} - 16 & \text{if } inc_{q+1} = 0 \\ v_{int} - 15 & \text{if } inc_{q+1} \neq 0 \end{cases}$$

Since the DFP antilogarithm result should be positive, the sign bit (R_{sign}) is zero.

6.5 Implementation and Comparisons

The proposed improved DFP antilogarithmic converter, which can compute operands in Decimal32, Decimal64 and Decimal128 formats, are modeled with VHDL and then simulated by using ModelSim respectively. A comprehensive testbench, which includes special test cases (NaN, Infinite, Subnormal or zero operands), corner test cases (close to one operands), and valid random DFP operands is used to verify the correctness of the design. The proposed architectures are synthesized using Synopsys Design Compiler with the STM 90-nm CMOS standard cells library [127] under the typical condition (1.2 V_{DD} core voltage and 25 ^{o}c operating temperature). The clock, input signals, and output signals are assumed to be ideal. Inputs and outputs of the proposed design are registered and the design is optimized for delay.

The delay model is based on logical effort method [128], which estimates the proposed architecture delay values in a technology independent parameter, FO4 unit (the delay of an inverter of the minimum drive strength (1x) with a fanout of four 1x inverters). To measure the total hardware cost in terms of number of gates, the area of the proposed architectures are estimated as the number of equivalent 1x two input NAND gates (NAND2). Note that 1 FO4$\approx$45ps, and 1 NAND2$\approx$4.4um^2 in the STM 90-nm CMOS standard cells library under the typical condition. Table 6.4 summarizes the delay and the area estimated based on the area and delay evaluation model for the Decimal64 antilogarithmic converter. The worst path delay of each stage is highlighted in the corresponding figure by dashed thick line. The evaluation results show that the critical path of the proposed architecture is located in the stage 3 (highlighted in Figure 6.4), and that the details of critical path in the Decimal64 implementation are reported in Table 6.5.

Since there is no comparable Decimal64 antilogarithmic converter, we reconstruct a 16-digit DXP antilogarithmic converter based on the proposed Decimal64 architecture, and then compare its results with those of a 53-bit radix-16 binary exponential converter [57] in terms of the critical path delay. To conduct a fair comparison, the timing evaluation unit (1τ = the delay of 1-bit full adder) applied in [57], is transformed to the unit of FO4 based on [57]. Since $1\tau\approx0.8ns$ in the AMS 0.35-um CMOS standard cells library (used in [57]), we obtain

Table 6.4: Delay and area of Decimal64 antilogarithmic converter.

Stage	**Worst Delay** (FO4)	**Areas** (NAND2)
Initial processing stage (Figure 6.2)	25.3	7501
Digit recurrence stage (Figure 6.3)	23.6	6134
Antilog computation stage (Figure 6.4)	28.0	15485
Final processing stage (Figure 6.5)	18.6	1178
Top-level control logic (FSM*)	3.5	672
Total	28.0**	30970

* FSM: finite-state machine; ** critical path delay.

Table 6.5: Details of critical path of the Decimal64 antilogarithmic converter.

Blocks in the critical path						**Total**
Reg	Mult2	Mux	Shift	CSA 4:2	setup	(ns)
0.07	0.53	0.06	0.23	0.29	0.08	1.26

$1\tau \approx 5.3$ FO4 so that the delay of the design [57] is evaluated by FO4 unit. Note that 1 FO4 $\approx 150ps$ in the AMS 0.35-um CMOS standard cells library under the typical condition. The comparison results in Table 6.6 show that the proposed 16-digit antilogarithmic converter has a 1.44 times shorter latency than that of the 53-bit radix-16 binary exponential converter in [57].

The results of the proposed design are also compared with those of our original design implemented based on the non-redundant data-path reported in [130]. The critical path delay of the original 16-digit DXP antilogarithmic converter is 8.25 ns (synthesized with the TSMC 0.18-um standard cell library in the typical condition, thus, 1 FO4 $\approx 75ps$). The comparison results reported in Table 6.6 show that the improved DXP antilogarithmic converter is about 3.91 times faster than the original design in terms of the latency.

With respect to the existing works implemented based on the CORDIC algorithm in [49], it is quite difficult to compare the hardware performance between the two different algorithms. To compute the Decimal128 exponential operation, the expression given in [49] for

the number of clock cycles for a generic implementation of the fast termination CORDIC algorithm is no. cycles $=2+16\times m+M+P$, where $m=19$ (the number of iterations), $M=20$ (the estimated number of cycles of a 16-digit DXP multiplier), and $P=2$ (the estimated number of cycles specific to the DFP process, such as the exception handling, the packing and the unpacking from IEEE 754-2008 DFP format). For Decimal64 comparison, we have $m=9$, $M=11$ (the estimated number of cycles of a 7-digit DXP multiplier), thus the total estimated number of clock cycles to compute the Decimal64 exponential operation is 157 (no. cycles $=2+16\times 9+11$, without the consideration of the cycles specific to the DFP process). The critical path delay of the implementation in [49] has the cycle time of 13 FO4 (taking the Power 6 processor as a reference). The comparison results reported in Table 6.6 show that the digit-recurrence approach proposed in this work is 3.84 times faster than the unit based on the CORDIC approach in terms of latency.

For further analysis, we compare the performance of the proposed architecture with the software approach reported in [41]. The software DFP transcendental function computation library is complied by the Intel C++ Compiler (IA32 version 11.1) [129]. It takes about 1060 clock cycles to compute a Decimal64 exponential result, running with Intel Core(TM) 2 Quad @ 2.66 GHz microprocessor. The comparison results reported in Table 6.6 show that the proposed hardware implementation in this work is about 45.83 times faster than the software implementation.

To analyze various characters of the proposed architecture for three different formats, we construct the hardware implementations for Decimal32, Decimal64 and Decimal128 formats respectively. Since the digit-width of the input decimal significand are 7-digit, 16-digit and 34-digit (q-digit) in three DFP formats respectively, there is a need to keep at least 12-digit, 21-digit and 39-digit ($q+g+3$-digit) precision for the data-path in the stage 1 and stage 2 in order to guarantee the correct selection of digits e_j during 10, 19 and 37 number of clock cycles. To analyze the optimized size of look-up tables in the stage 3 of the proposed architecture, we reallocate values stored in the look-up tables using multiple tables approach. Since $q+g+3$-digit values of $-2.3\times\log_{10}(1+e_j 10^{-j})\times 10^{j+1}$ need to be stored in the look-up-table I and look-up table II for the computation of the residual $W[j]$ during the iterations $j\leq h$ ($h=4$, 9 and 18 for three DFP formats), and 19 different $q+g+3$-digit values of

Table 6.6: Comparison results for the delay of Decimal64 antilogarithmic converter.

Works	**Cycle time** (FO4)	**Cycles** (No.)	**Latency** (FO4)	**Ratio**
Proposed	28.0	19	532.0	1.00
Proposed*	28.0	18	504.0	0.95
Original [130]*	110.0	18	1980.0	3.72
Radix-16 [57]	45.1	17	766.7	1.44
CORDIC [49]	13.0	157	2041.0	3.84
Software [41]	$\approx$ 23.0	1060.0	24380.0	45.83
Software library Running at Intel Core(TM) 2 Quad @ 2.66 GHz.				

* 16-digit DXP exponential converter.

$-2.3\times10\times e_j/\ln(10)$ need to be stored in the look-up table III, thus, the optimized size of look-up tables are about 0.69 KByte, 2.14 KByte and 7.23 KByte for three DFP formats. To achieve the q-digit decimal significand ($R_{significand}$) by the decimal compound adder in the stage 4, there is a need to keep at least 10-digit, 19-digit and 37-digit (q+3-digit) precision for the data-path in the stage 3 for three DFP formats.

6.6 Summary

In this chapter, we present a DFP antilogarithmic converter that is based on the digit-recurrence algorithm with selection by rounding. We develop the radix-10 algorithm, improve the architecture, and implement it with the STM 90-nm CMOS standard cells library. The implementation results show that the improved architecture is 3.91 times faster than our previous design [130] in terms of the latency. To provide more reference for floating-point-unit designers when they consider a fast implementation for the radix-10 implementation, we compare the proposed architecture with a binary radix-16 converter [57] and the results show that the proposed radix-10 decimal antilogarithmic converter has a 1.44 times shorter latency than that of the binary radix-16 converter. Moreover, we compare the proposed architecture with a recent high performance implementation based on the decimal CORDIC

algorithm [49]. Although a comparison between two different algorithms may depend on too many parameters, the design presented in this chapter shows a latency 3.84 times faster than that of the unit based on the CORDIC algorithm. In addition, compared with the software DFP transcendental function computation library [41], the proposed hardware implementation is about 45.83 times faster than the software implementation.

Part V

Decimal Reciprocal and Radix-100 Division Units

Chapter 7

Design and Implementation of Decimal Reciprocal Unit

This chapter presents the efficient design and implementation of a 16-digit DXP decimal reciprocal unit based on the new IEEE 754-2008 standard for floating-point arithmetic. In this design, the decimal reciprocal result is obtained through an initial approximation of the reciprocal that consists of a look-up table, a multiplication, and followed three Newton-Raphson iterations. We analyze the computation error for the look-up tables with different sizes, in order to find the smallest size of look-up table for the efficient hardware implementation. The proposed design can utilize a $2^{10} \times 10$ bits look-up table, which is half of the look-up table as that used in the previous design to compute a faithful reciprocal result. The proposed architecture takes 119 clock cycles to achieve the faithful 16-digit accuracy approximation of the reciprocal of a 16-digit DXP number. The proposed implementation of decimal reciprocal unit is verified with a Xilinx Virtex-II Pro P70 FPGA device and synthesized by using TSMC 0.18um standard cell library.

7.1 Introduction

In recent years, more and more hardware-oriented works for the decimal division, with either digit-by-digit recurrence or functional iteration algorithms, have been proposed in [25, 29, 27, 26, 28]. A decimal division arithmetic unit is the most time consuming and the most complex arithmetic unit among the basic decimal arithmetic operations. Designing a high-speed reciprocal unit is very useful for division operation because the division can be replaced by the following method: the reciprocal of divisor is computed at first, and then it is used as

the multiplier in a subsequent multiplication with the dividend. If several divisions by the same divisor need to be performed, this method is especially efficient when several divisions are produced by one single divisor, because once the reciprocal of the divisor is found in the first division, each one of the subsequent divisions only contains one additional multiplication.

Creating an efficient look-up table is a key point in designing a decimal reciprocal unit based on Newton-Raphson algorithm. In today's world, as the required precision of the approximation increases, the size of the memory to implement the table lookups becomes prohibitive. In this work, the look-up table is created by an algorithm based on the first-order Taylor expansion. Compared with Symmetric Bipartite Tables method [131], the size of the look-up table is much smaller to achieve the same accuracy. This method requires first, a value from the table, and second, a multiplication of this value with the modified operand to obtain the initial approximation [132]. The Newton-Raphson iteration is the second stage of the implementation, which includes a decimal squarer, a decimal multiplier and a decimal adder.

This work presents the design and implementation of a decimal reciprocal unit, in which the initial approximation is achieved using an efficient look-up table and a multiplication. After the initial approximation, three Newton-Raphson iterations are operated to obtain the final 16-digit (Decimal64) accuracy approximation of the reciprocal of a DFP operand. This decimal reciprocal operation takes 119 clock cycles to achieve the 16-digit faithful reciprocal results. It just requires a subsequent multiplication to realize the division operation. To represent the signed decimal intermediate value, all variables in the architecture are represented with 10's complement number system in the BCD encoding. The remainder of this chapter is organized as follows: Section 7.2 presents the algorithm that is used to create the look-up table, and then describes how to determine the minimal size of look-up table; Section 7.3 presents the Newton-Raphson iteration, and how it is applied in our design; Section 7.4 presents an overview of the architecture of the decimal reciprocal units; Section 7.5 shows the implementation results. Section 7.6 gives the conclusion.

7.2 Initial Reciprocal Approximation

To create an efficient look-up table that has a small size while maintaining accuracy is one of our goals in our design. The algorithm applied to create the look-up-table is shown as follows:

7.2.1 Algorithm

According to IEEE 754-2008, the normalized decimal significand of Decimal64 operands can be viewed as the normalized 16-digits DXP number in the range of $0.1 \leq X < 1.0$. An implicit leading zero and the 16 fractional digits constitute the mantissa. Thus, the 16-digits mantissa is expressed as:

$$X_{mantissa} = [0.x_1x_2x_3..x_{16}] \tag{7.1}$$

Then, X can be split into two parts: one part from 1^{st} to m^{th}, and the other from $(m+1)^{th}$ digit to 16^{th} digit, shown as follows:

$$X_1 = [0.x_1x_2x_3...x_m] \tag{7.2}$$

$$X_2 = [0.x_{m+1}x_{m+2}...x_{16}] \times 10^{-m} \tag{7.3}$$

Thus,

$$X_{mantissa} = X_1 + X_2 \tag{7.4}$$

According to [28], the initial reciprocal approximation is obtained by the first-order Taylor expansion at the subinterval midpoint:

$$X^{-1} \approx \frac{1}{X_1+5\times10^{-m-1}} - \frac{1}{(X_1+5\times10^{-m-1})^2} \times (X - (X_1 + 5 \times 10^{-m-1})) \approx \frac{2\times X_1 - X + 10^{-m}}{(X_1+5\times10^{-m-1})^2} \tag{7.5}$$

Since $(2 \times X_1 - X = X_1 - X_2)$ and $(10^{-m} - X_2)$ correspond to the 10's complement of X_2, equation (7.5) can be rewritten as:

$$X^{-1} \approx \frac{X_1 + 10^{-m} - X_2}{(X_1 + 5 \times 10^{-m-1})^2} \tag{7.6}$$

Thus, the initial reciprocal approximation R_0 can be obtained as:

$$X^{-1} \approx R_0 = C' \times X' \tag{7.7}$$

Where,

$$C' = \frac{1}{\left(X_1 + 5 \times 10^{-m-1}\right)^2} \tag{7.8}$$

$$X' = X_1 + 10^m - X_2 \tag{7.9}$$

In equation (7.7), the first term $C' = \frac{1}{(X_1+5\times10^{-m-1})^2}$ is read from the look-uptable addressed by X_1 (without the leading zero) as a constant term. For the remaining term $X' = X_1 + 10^m - X_2$, it can be achieved from the operand modifier. The operation of the operand modifier is to keep the digits from 1^{st} to m^{th} unchanged and to modify the digits from $(m+1)^{th}$ digit to 16^{th} to 10's complement values.

7.2.2 An Efficient Look-up Table Creation

In [28], a look-up table with size of $2^p \times 2p$ bits is created, where $p = \lceil (k \cdot 10)/3 \rceil$ and k indicates the number of digits used to access the look-up table. The initial approximation is accurate to $(2k-3)$ fraction digits. The X_1 and the output from the look-up table, C', is encoded by using Densely Packed Decimal (DPD) format [54]. It is obvious that initial values read from the look-up table contain more bits than the theoretical initial approximation. For example, according to [28], if 3 digits initial approximation needs to be guaranteed, the look-up table size is $2^{10} \times 20$ bits and can be further reduced.

In order to keep the minimal look-up table size, we assume the look-up table size is $2^3 \times p$ corresponding to the proposed 16-digit DXP reciprocal unit. Different values of p are verified on a MATLAB simulation model, and the architecture simulated in MATLAB is matched to the hardware implementation of the proposed reciprocal unit. A group of test vectors that contain 900 16-digit decimal operand are selected. They are uniformly distributed from 0.1 to 1 and cover all the combination of 3 input digits for generating the values stored in the look-up table. Firstly the ROM size with $2^{10} \times 20$ bits mentioned in [28] is simulated, then $p = 18$, 14, 10, are selected after our previous simulation. Three look-up tables are created with the size of $2^{10} \times 18$, $2^{10} \times 14$ and $2^{10} \times 10$ bits respectively. Table 7.1 compares the three newly created look-up tables with one that provided by a related research [28]. Since the accuracy of the decimal reciprocal does not decrease because of shrinking the look-up table

Table 7.1: Evaluation of different size of look-up table.

Size of Look-up Table	$2^{10}\times 20$ [28]	$2^{10}\times 18$	$2^{10}\times 14$	$2^{10}\times 10$
Table Size Decrease Ratio	1.00	0.90	0.70	0.50

size from $2^{10}\times 20$ to $2^{10}\times 10$, $2^{10}\times 10$ bits is chosen in our implementation and it is only half size of the one used in [28]. The simulation results show that the proposed algorithm can maintain 16-digits accuracy after 3 Newton-Raphson iterations, while shrinking the look-up table size from $2^{10}\times 20$ to $2^{10}\times 10$ bits.

According to the modified algorithm (7.6) and our simulation results, the equation (7.3) is revised to be:

$$X_2 = [0.x_{m+1}] \times 10^{-m} \tag{7.10}$$

Where $m=3$, so correspondingly, X' becomes

$$X' = [0.x_1 x_2 ... x_m \tilde{x}_{m+1}] \tag{7.11}$$

7.3 Newton-Raphson Iteration

Newton-Raphson iteration is a well-known iterative method to approximate the root of a non-linear function. Let $f(x)$ be a well-behaved function, and let r be a root value of the equation $f(x)=0$. We start with x_0 which is a good estimate of r and let $r=x_0+h$. The number h measures how far the estimated x_0 is from the truth. Since h is very 'small', the linear approximation can be used to conclude that:

$$0 = f(r) = f(x_0 + h) \approx f(x_0) + hf'(x_0) \tag{7.12}$$

And therefore, unless $f'(x_0)$ is close to 0,

$$h \approx -\frac{f(x_0)}{f'(x_0)} \tag{7.13}$$

It follows that

$$r = x_0 + h \approx x_0 - \frac{f(x_0)}{f'(x_0)} \tag{7.14}$$

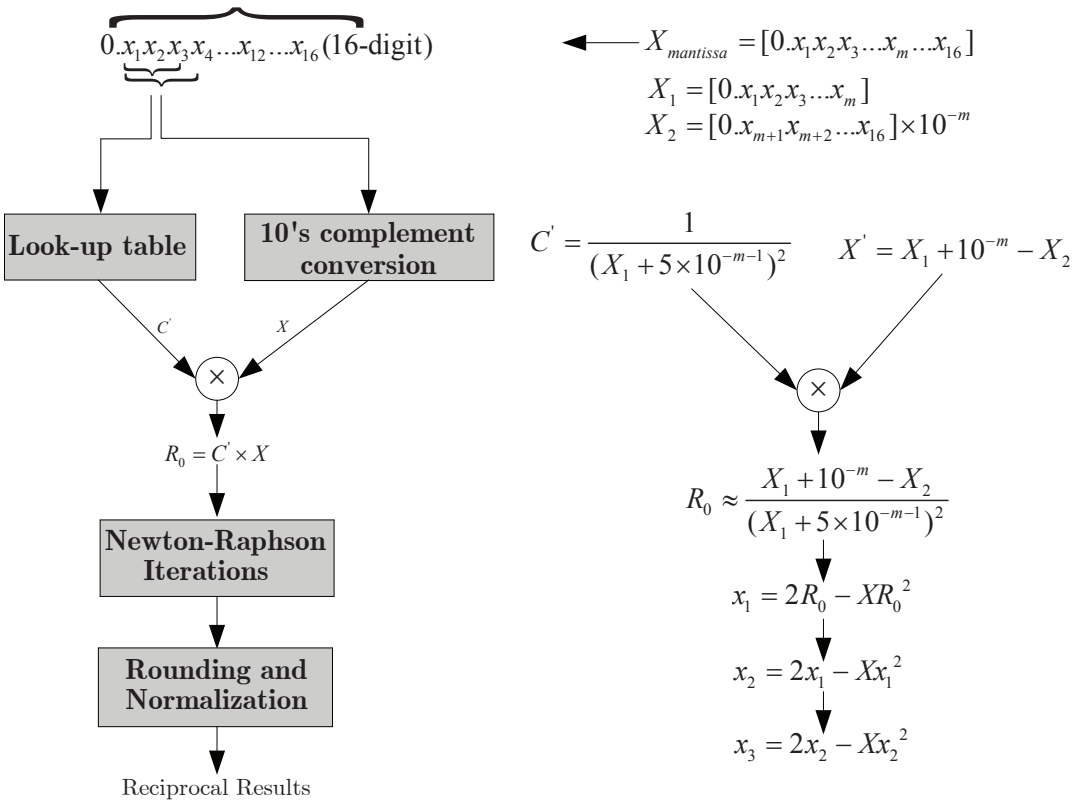

Figure 7.1: Data-path of the reciprocal computation.

Our new improved estimated x_1 of r is therefore given by:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{7.15}$$

Consequently, if x_i is the current estimate, then the next estimate x_{i+1} is given by:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{7.16}$$

The obtained equation (7.16) is called the Newton-Raphson formula. In order to compute the reciprocal, the following function and its derivative are used:

$$f(x) = 1/x - X \tag{7.17}$$

Thus,

$$f'(x) = -1/x^2 \tag{7.18}$$

Substituting the equations (7.16) and (7.17) for the equation (7.18) yields:

$$x_{i+1} = x_i(2 - Xx_i) \tag{7.19}$$

The equation (7.19) is rewritten as:

$$x_{i+1} = 2x_i - X{x_i}^2 \tag{7.20}$$

which can be implemented in hardware in order to double the accuracy in each iteration. By using the equation (7.20), one decimal square, one decimal multiplication, one double generator and one decimal subtraction are required for computing x_{i+1}. Let $\delta_{i+1} = 1/X - x_i$ be the error in each iteration, then it can also be expressed as:

$$\delta_{i+1} = 1/X - x_{i+1} = 1/X - x_i(2 - x_i X) \tag{7.21}$$

Which can also be expressed as:

$$\delta_{i+1} = X(1/X - x_i)^2 = X{\delta_i}^2 \tag{7.22}$$

The equation (7.22) clearly proves that the absolute error degrades quadratically in each Newton-Raphson iteration because it is proportional to the square of the previous error. Figure 7.1 summarizes the data-path of the proposed decimal reciprocal unit based on the proposed approach described above.

7.4 Hardware Implementation

Figure 7.2 shows the sequential architecture of the proposed decimal reciprocal unit. It takes 119 clock cycles to achieve the 16-digits accuracy approximation of the reciprocal of a decimal-64 floating-point number. The architecture involves two stages. In the first stage, the initial reciprocal approximation of R_0 is obtained through look-up table and operand modifier. The size of the look-up table is $2^{10} \times 10$ bits, which is a tradeoff of hardware cost and accuracy. The second stage is the implementation of Newton-Raphson iteration. In this work, the decimal multiplier and squarer are implemented based on the DXP sequential multiplier in [125].

At the beginning, the first 3-digits are obtained from the mantissa of X. They are represented by 12 bits BCD code and converted to 10 bits DPD code as the address of the look-up table to achieve a 10 bits value. They represent the 3 digits values from the

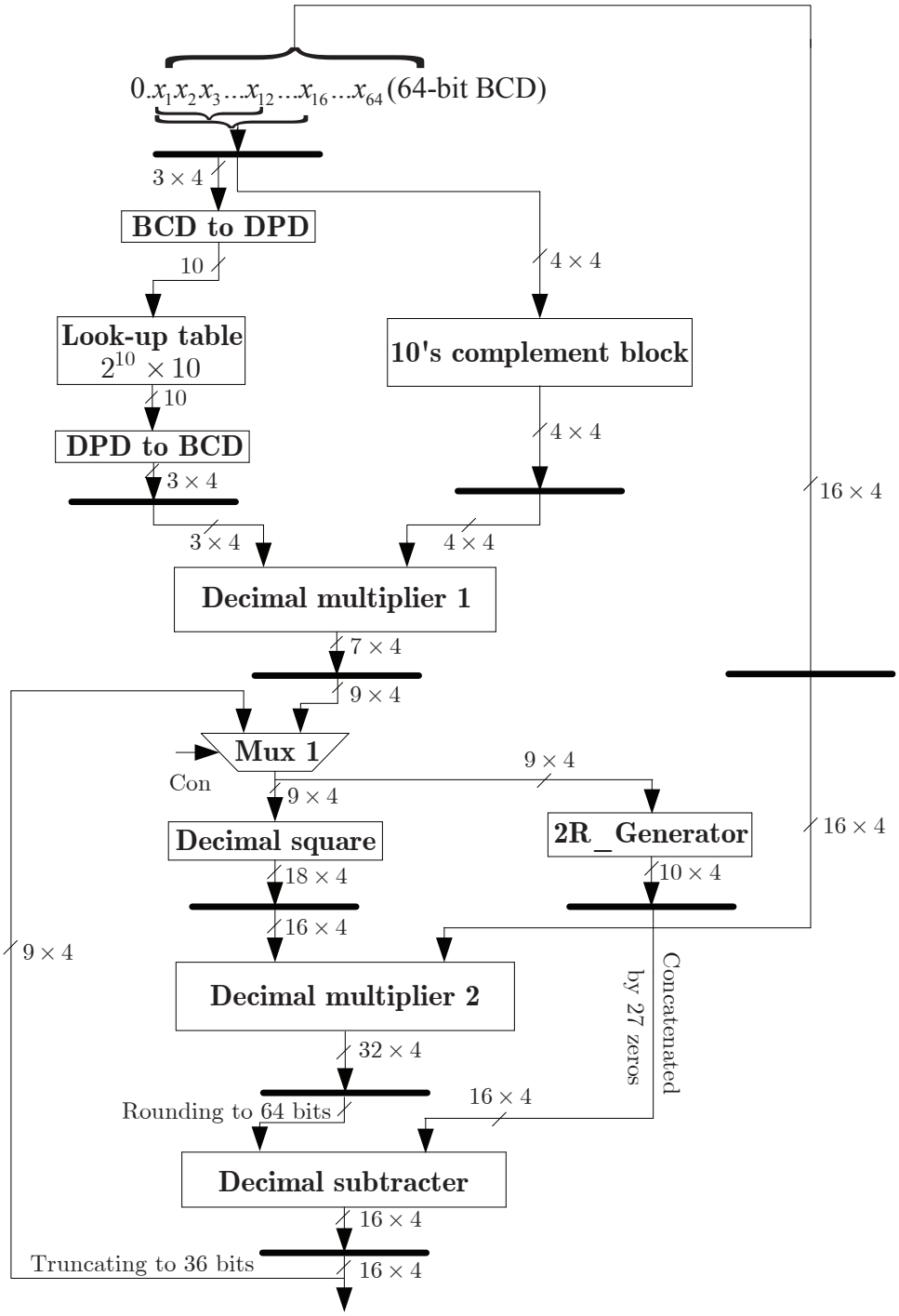

Figure 7.2: Architeture of the proposed reciprocal unit.

look-up table, and then those 10 bits DPD code is converted back to 12 bits BCD code for subsequent calculation. At the same time, the first 4-digits (16 bits BCD) of X are carried into the operand modifier. The operand modifier unit is constructed with 10's complement

converter. The most significant 12 bits of X stay the same, and the least significant 4 bits are converted to its 10's complement values. So the first 2 clock cycle's work is to obtain the values from the look-up table and the operand modifier.

From the 3^{rd} to 11^{th} clock cycle, the output of the look-up table is multiplied by the output of the operand modifier in the decimal multiplier 1 to achieve the initial approximation for the reciprocal of X. Then, the output of multiplier 1 is complemented by 8-bit zeros and selected by Mux1 and sent to the decimal squarer, and the result of the squarer is obtained at the 25^{th} clock cycle. Because the size of a decimal squarer is smaller than a decimal multiplier and the decimal square computation is faster than the decimal multiplication, we choose to use a decimal squarer in our design. The Newton-Raphson iteration result is achieved just using one decimal square computation, one decimal multiplication and one decimal subtraction. From the following 26^{th} and 46^{th} clock cycles, the 16-digits mantissa is multiplied by the result of the squarer in multiplier 2, followed by a rounding operation. At the 47^{th} clock cycle, the rounding result of multiplier 2 is subtracted from the $\times 2$ value of the output of the multiplier 1 that is concatenated by 27 zeros. Now the first result of Newton-Raphson iteration is achieved. Then it is truncated and selected by Mux1 to be computed in the decimal squarer. From the 48^{th} to 118^{th} clock cycles, the second and third Newton-Raphson iterations are operated. At the 119^{th} clock cycle, the result of the third Newton-Raphson iteration is obtained, which is the 16-digits accuracy approximation of decimal reciprocal. The decimal reciprocal unit takes 11 clock cycles to get the initial approximation of the reciprocal, and each Newton-Raphson iteration takes 36 clock cycles. Consequently, it takes 119 clock cycles to obtain the reciprocal approximation of a 16-digit DXP operand in our decimal reciprocal unit.

7.5 Implementation Results

The proposed architecture for this decimal reciprocal unit is modeled in VHDL and synthesized with TSMC 0.18 um standard cell library using Synopsys. The RTL and gate level netlists are all verified with the same test vectors generated from the MATLAB fixed-point model. Table 7.2 gives the proposed implementation results of design in several aspects. The

Table 7.2: Hardware implementation results.

	16-digit DXP Reciprocal Unit
Look-up table size (bits)	$2^{10} \times 10$
Cycle time (ns)	2.79
No. of cycles	119

worst case delay path is produced in the decimal subtracter, in which the maximum clock frequency is 358.4 MHz. The decimal reciprocal unit is also implemented using a Xilinx Virtex2p XC2VP70 FPGA board with package ff1517 and speed -7. The implementation of this decimal reciprocal unit is synthesized with Synplify pro 8.5, which occupies 1 BRAM out of 328, 1 GCLK I/O block out of 16, 132 I/O blocks out of 964, and 2009 slices out of 33088. The maximum clock frequency is 169 MHz.

7.6 Summary

This chapter presents the design of a decimal reciprocal unit for the computation of the 16-digit DXP operand. The unit uses the minimum size of the look-up table ($2^{10} \times 10$ bits) for the initial approximation algorithm, and it takes 119 clock cycles to compute the decimal reciprocal results by using look-up table and three Newton-Raphson iterations. And most importantly, a 16-digits accuracy can be guaranteed in all the cases. The main contribution of our work is that the presented design utilizes a $2^{10} \times 10$ bits look-up table, which is half size of the look-up table used in the design [28]. In the future, a combined DFP reciprocal, division and square root unit can be designed and implemented based on the proposed Newton-Raphson iteration method in order to achieve exact rounded decimal reciprocal, division and square root results of the DFP operands in a single unit. Also, more mathematical error analysis should be investigated to create more efficient look-up table with a smaller size.

Chapter 8

Design and Implementation of A Radix-100 Decimal Division

In this chapter, a 16-digit fixed-point (DXP) radix-100 decimal divider is designed and implemented. A decimal non-restoring algorithm with pre-scaling method is adopted in the proposed 16-digit radix-100 decimal divider. The quotient digits, q_i, in a signed-digit (SD) format, is determined by the 2-digit integer of the partial remainder and can be produced in each iteration. The size of the look-up table for pre-scaling is reduced to half by an additional scaling step. The generation of multiple divisor is accomplished by a decimal carry-save adder (DCSA) tree and a decimal carry-look-ahead (DCLA) adder. The proposed 16-digit radix-100 decimal divider is implemented in Virtex-II PRO P30 FPGA configuration and synthesized with TSMC 0.18-um standard cell library respectively. The implementation result indicates that the maximum frequencies in FPGA and 0.18-um technology are 50.1MHz and 111.1 MHz respectively, and the quotient result with 16-digit accuracy can be obtained in $3 + (2 \times 9)$ cycles.

8.1 Introduction

A decimal division arithmetic is the most time-consuming and complex arithmetic unit among the fundamental decimal arithmetic operations. In [28], Wang and Schulte present a decimal floating-point (DFP) divider in which the reciprocal of divisor is obtained by initial approximation and Newton-Raphson iterations, and then multiplied with dividend to obtain the quotient. In [27], a radix-10 algorithm, based on the SRT digit-recurrence methods with a minimally redundant signed-digit set ($\rho = 5/9$), is proposed to implement a 16-digit

DXP divider. In [25, 29], the radix-10 dividers are constructed based on non-restoring digit-by-digit algorithm and one decimal quotient digit is achieved in each iteration. In [26], a decimal division unit is designed based on the radix-10 non-restoring algorithm with pre-scaling method. In this work, we analyze and implement a radix-100 decimal division with non-restoring algorithm, which is the first attempt in the field of the radix-100 decimal division (two decimal quotient digits obtained in each iteration). With the increase of radix from 10 to 100, some problems in radix-100 division occur: 1) the size of the look-up table which stores pre-scale parameters is significantly increased; 2) since the multiple range has been increased from $[-9, 9]$ to $[-99, 99]$, the arithmetic of multiple of divisor becomes more complex. To reduce the size of the look-up table, a new pre-scaling method is proposed in this radix-100 decimal divider, which can shrink the look-up table to half size. The generation of multiple divisor is obtained by a carry-save-tree and a decimal carry-look-ahead (DCLA) adder which can reduce carry delay.

This chapter is organized as follows: Section 8.2 describes the radix-100 non-restoring decimal division with pre-scaling algorithm and a method for reducing size of the look-up table. Section 8.3 depicts an overview of the architecture of the radix-100 decimal divider; Section 8.4 analyzes the implementation results. Section 8.5 gives the conclusions.

8.2 Algorithm

8.2.1 Radix-100 Non-Restoring Decimal Division

In this work, we mainly focus on a 16-digit DXP radix-100 division which can achieve a 16-digit accurate quotient. The overview of this radix-100 non-restoring division algorithm is represented as follows:

The dividend and divisor are 16-digit DXP numbers that are compliant with *significand* region of the 64-bit DFP format in IEEE 754-2008. At the beginning, the dividend and divisor are normalized to the range of [10, 100). The partial remainder, P, is obtained by the equation (8.1):

$$P[i+1] = 100 \times (P[i] - q_i \times D) \tag{8.1}$$

In (8.1), $P[1]$ is dividend; D is divisor; and q_1 is the 2-digit most significant digit (MSD) of dividend. The quotient digits q_i are produced by a selection function that is determined by both partial remainder $P[i]$ and divisor D. In this work, the quotient digits q_i are achieved only by truncating the partial remainder, P, to its 2-digit MSD integer as shown in equation (8.2),

$$q_i = truncated(P[i]) \tag{8.2}$$

Finally, the final 16-digit accurate quotient result Q is calculated by the accumulation of the quotient digits as shown in equation (8.3):

$$Q = \sum_{i=1}^{9} q[i] \times 10^{-(i-1)} \tag{8.3}$$

8.2.2 Pre-scaling method

In order to simplify the quotient digits selection function, the divisor is pre-scaled to the value close to 1, so that in the selection function each 2-digit quotient can only be obtained by truncating partial remainder to its 2-digit integer part. Since less redundancy of the range of 2-digit quotient requires more selection function table size to distinguish different multiples, we adopt the maximum 2-digit quotient selection range $q_{i+1} \in \{-99, -98, -97, ..., 0, ...97, 98, 99\}$. If the pre-scaled divisor, $(1 + \alpha)$ is substituted in (8.1), then,

$$P[i+1] = 100 \times (P[i] - q_i - q_i \times \alpha) \tag{8.4}$$

$P[i+1]$ is converged in the range

$$-100 < P[i+1] < 100 \tag{8.5}$$

According to (8.4) and (8.5), (8.6) is obtained:

$$-1 < P[i] - q_i - q_i \times \alpha < 1 \tag{8.6}$$

The range of α is analyzed by cases in order to keep the convergence of $P[i+1]$ as follows:
Caes 1: If $P[i]$ equals to zero, q_i obtained by truncating $P[i]$ is zero, then (8.6) is always satisfied. Thus,

$$\alpha \in R \tag{8.7}$$

Table 8.1: Adjustment of divisor.

Divisor Range	$[10, 20)$	$[20, 30)$	$[30, 50)$	$[50, 100)$
Adjustment Divisor	$5D$	$3D$	$2D$	D

Case 2: If $P[i] > 0$, the $0 \leq P[i] - q_i \leq 1$ is satisfied, where $1 \leq q_i \leq 99$. Thus,

$$0 < \alpha < 1/99 \tag{8.8}$$

Case 3: If $P[i] < 0$, the $-1 \leq P[i] - q_i \leq 0$ is satisfied, where $-99 \leq q_i \leq -1$. Thus,

$$0 < \alpha < 1/99 \tag{8.9}$$

It is obvious that only if the divisor is scaled by pre-scaling parameter, *sp*, to the region $(1, 1 + 1/99)$, the convergence of $P[i + 1]$ in the range $(-100, 100)$, the two signed quotient digits can be, therefore, obtained in each iteration.

8.2.3 Analysis of Look-up Table Size

With the increasing of the radix from 10 to 100, 3-digit MSD of divisor is used to generate the look-up table address for selecting the pre-scale parameters. Therefore, 900 corresponding pre-scale parameters need to be stored in the table. The divisor is adjusted to the range of $[50, 100)$ by multiplying with adjustment parameters 5, 3 and 2 (see Table 8.1), in order to reduce the number of pre-scale parameters stored in the ROM is reduced from 900 to 500. Meanwhile, the dividend is adjusted by the same adjustment parameters to guarantee the correct quotient results.

The less digits the pre-scale parameter *sp* has, the smaller the size of look-up table will be. However, if the 16-digit divisor can not be scaled to the range of $(1, 1 + 1/99)$, the convergence of $P[i+1]$ can not be guaranteed. Therefore, we do an evaluation to determine the minimum number of digits of *sp* as shown in (8.10):

$$1 < d_1 d_2 . d_3 .. d_{16} \times sp < 1 + 1/99 \tag{8.10}$$

The 16-digit DXP divisor is represented as $d_1 d_2 . d_3 ... d_{16}$. To guarantee all of the 16-digit divisors scaled to the range of (1, 1+1/99), for each 3-digit MSD of divisor, the largest and

the smallest 16-digit divisor are $d_1d_2.d_30...0$ and $d_1d_2.d_30...0 + (0.1 - 10^{-14})$ respectively and they should satisfy (8.10):

$$1 < (d_1d_2.d_30...0) \times sp < 1 + 1/99 \tag{8.11}$$

$$1 < (d_1d_2.d_30...0 + 0.1 - 10^{-14}) \times sp < 1 + 1/99 \tag{8.12}$$

According to the numerical analysis of (8.11) and (8.12), we obtain:

$$\frac{1}{d_1d_2.d_30...0} < sp < \frac{1 + 1/99}{d_1d_2.d_30...0 + 0.1 - 10^{-14}} \tag{8.13}$$

The digit number of sp can be determined by the distance between the smallest bound and the largest bound of condition (8.13). Since the range of adjusted $d_1d_2.d_3$ is in $[50.0, 99.9)$, it is obvious that 4-digit of sp is enough to guarantee all 16-digit divisor to be scaled to the range of $(1, 1 + 1/99)$. As a result, there are at least 500 4-digit scaling parameters needed to be stored in the look-up table. After the analysis of these scaling parameters, we found that 1) there are 200 different scaling parameters in these 500 4-digit scaling parameters, 2) the largest suitable step length is 0.00005 for every two continuous scaling parameters. The scaling parameters are illustrated in Figure 8.1.

As shown in Figure 8.1, the scaling parameters are in the range of $[0.01010, 0.02005]$ with step length of 0.00005. In order to shrink the look-up table size, first, all the scaling parameters are scaled to 5 times to reduce the digit width of sp from 4-digit to 3-digit; second, all the 3-digit sp are subtracted by the base-value 0.00202 ($0.01010/0.00005 = 0.00202$) to achieve step length of every two continuous scaled parameters; and third, the step length is converted to 8-bit in order to be stored in the look-up table. Therefore, the total size of the look-up table is shrinked from $2^{10} \times 16$ to $2^9 \times 8$ (0.5 Kbyte).

8.3 Architecture

As shown in Figure 8.2, the decimal radix-100 divider consists of three parts. The first part, as shown in Figure 8.2 (a), is pipelined into 2 levels. Level 1 mainly consists of a multiple logic for adjusting the divisor and the dividend. In the 1^{st} clock cycle, 1) the divisor needs to be adjusted to the range of [50.0, 99.9] based on Table 8.1, and 2) the scaling parameters

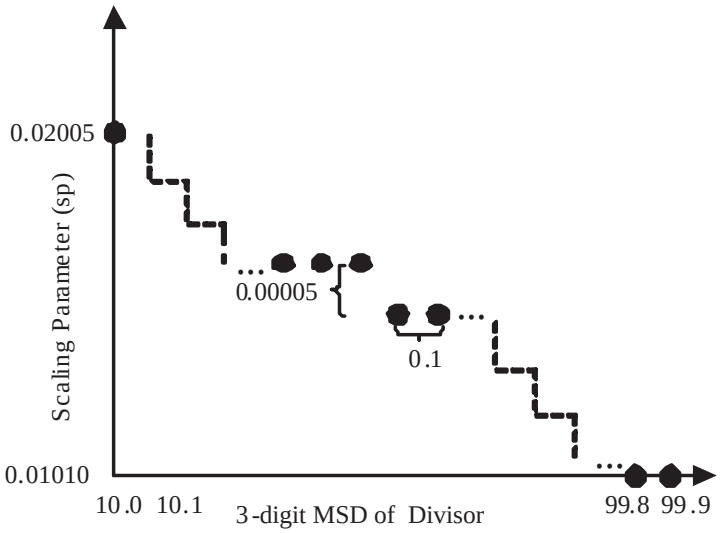

Figure 8.1: Scaling parameters in the look-up table.

stored in the look-up table are scaled 5 times, thus divisor is adjusted by multiplying with 5, 10, 15 and 25 in the multiple logic which is extended and implemented from [23]. The adjusted divisor and its 3-digit MSD are stored in *Reg3* and *Reg4* respectively. Level 2 mainly consists of an address generator, a $2^9 \times 8$ look-up table, a binary-to-BCD converter and a 3-digit decimal DCLA adder. In the 2^{nd} clock cycle, first, the 9-bit ROM address is generated by 3-digit MSD of adjusted divisor in the address generator; second, the 8-bit step length of scaling parameters is obtained from the look-up table and is converted back to 3-digit decimal value by binary-to-BCD converter; third, the scaling parameter is achieved by adding 3-digit decimal step length with base-value (202) in a 3-digit DCLA and is stored in *Reg6*. When the adjusted divisor is transferred to *Reg5*, the dividend is adjusted by multiple logic at the same time and stored in *Reg3*.

The second part, as shown in Figure 8.2 (b), is pipelined into two levels as well. Level 1 mainly consists of a decimal DCSA tree [23] and a decimal DCLA adder in order to generate a scaled divisor ($sp \times D$), a scaled dividend ($sp \times dividend$) and a multiple divisor ($q_i \times sp \times D$). In the 3^{rd} clock cycle, first, the adjusted divisor ($X1$) and the scaling parameter ($X2$) are selected by *Mux4* and *Mux5* respectively; second, the scaled divisor is achieved by multiplying the adjusted divisor with scaling parameter, $sp \times D$, in the DCSA tree and

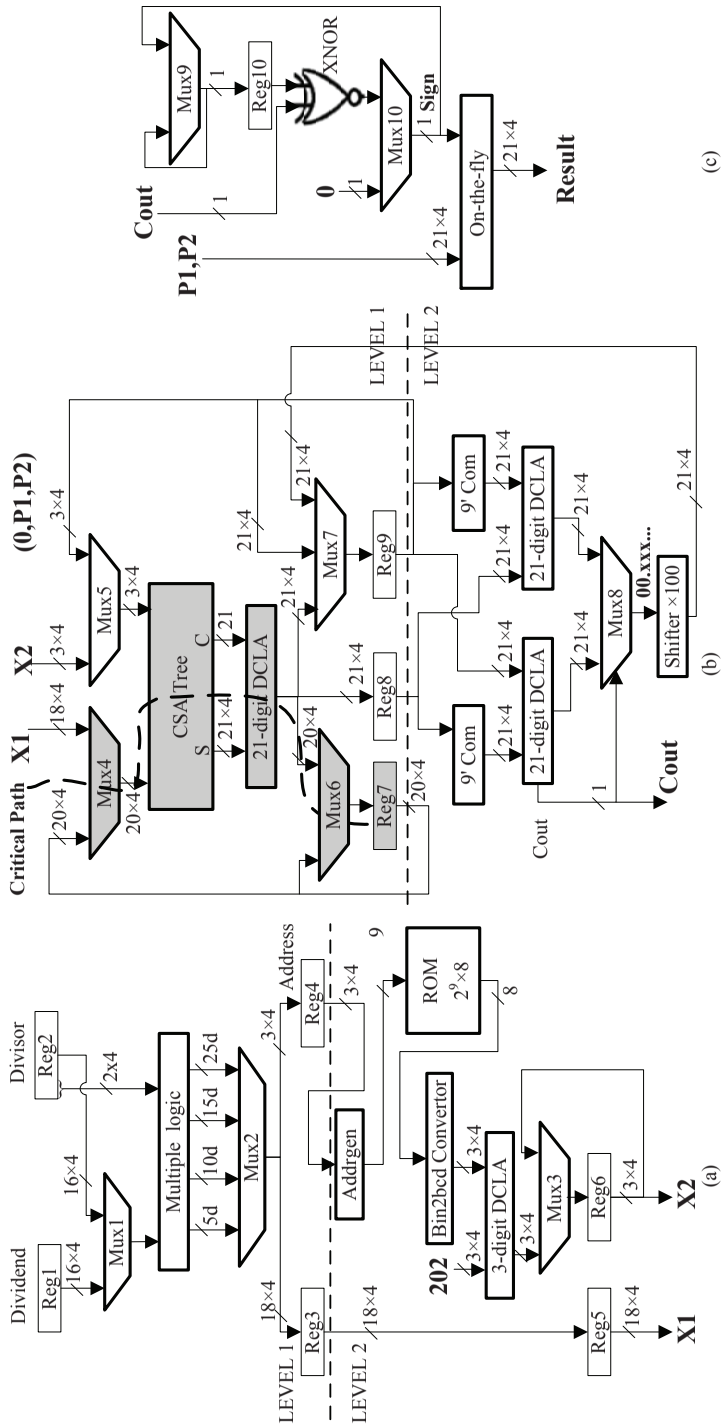

Figure 8.2: Architecture of decimal radix-100 divider.

21-digit DCLA adder; finally, the scaled divisor is selected by $Mux6$ and stored in $Reg7$. From the 4^{th} to the 21^{st} clock cycles, 9 times iterations (two clock cycles for each iteration) are processed to obtain all of quotient digits so that the 16-digit accurate quotient results

are guaranteed. In the 4^{th} clcok, the dividend is scaled by the same scaling parameter, sp, to achieve the scaled dividend, $sp \times Dividend$, in the DCSA tree and 21-digit DCLA adder. In the 5^{th} clock cycle, while the scaled divisor and the first quotient digit ($q_1 =$ 2-digit MSD of dividend) are selected by *Mux4* and *Mux5* respectively, the multiple divisor, $q_i \times sp \times D$, is thus obtained by the DCSA tree and 21-digit DCLA. In level 2, in order to achieve a positive magnitude of the partial remainder, $P[i+1]$, and to avoid the carry delay of 10's complement converter, two 9's complement converters, two 21-digit DCLA adders and one shifter are implemented to achieve positive and negative partial remainders. In the 6^{th} clock cycle, first, the positive and negative values of $sp \times P[i] - sp \times D \times q_i$ are obtained by subtracting the scaled dividend and the scaled multiple divisor in two 21-digit DCLA adders respectively. Then, the positive value is selected in *Mux8* which is shifted 2-digit to the left for obtaining $P[i+1]$. The 2-digit quotient q_{i+1} is obtained by truncating $P[i+1]$ to its 2-digit integer part. Therefore, we can get a 2-digit quotient q_{i+1} with two clock cycles in each iteration.

The third part, as shown in Figure 8.2 (c), calculates the quotient in each iteration by the On-the-fly algorithm [133]. The magnitude of the 2-digit SD quotient, P_1P_2 is obtained from *Reg*9, and the sign of P_1P_2 is determined by the XNOR operation between the carry out and the sign of the previous partial remainder. Two registers are in the On-the-fly unit, and their values are changed according to the sign and the value of each new signed digit. The operation of addition in the On-the-fly unit has only two digits delay. Since the radix-100 decimal division needs to guarantee 16-digit accurate quotient results, we need a total of 9 times iterations to achieve this goal. After 9 times iterations, the 18-digit final quotient result is generated simultaneously. Then, the 18-digit final quotient result is rounded to 16-digit accurate quotient result. Thus, the total number of clock cycles to achieve a 16-digit accurate quotient result is $3+(2 \times 9)$ cycles.

8.4 Analysis of Implementation Results

The proposed 16-digit radix-100 DXP decimal divider is modeled with Verilog and implemented on Virtex-II PRO P30 FPGA configuration. The proposed 16-digit radix-100 DXP decimal divider is synthesized with XST and placed and routed by Xilinx ISE 9.1. It occupies

Table 8.2: Details of critical path on FPGA.

Mux4	CSA tree	DCLA	Mux6	Reg7	**Total** (ns)
0.683	11.038	6.998	0.548	0.682	19.949

1 out of 16 GCLK I/O block, 203 out of 556 I/O blocks, and 3,976 out of 13696 slices (29%). The maximum clock frequency and latency are 50.1 MHz and 21 clock cycles respectively. The critical path of the proposed architecture is mainly produced in the DCSA tree and the DCLA adder which is highlighted in Figure 8.2 (b) (dotted line) and its details are available in Table 8.2.

The implementation results of the proposed 16-digit radix-100 DXP decimal divider are compared with the results of a 16-digit radix-10 DXP decimal divider [26], because they both utilize the same decimal non-restoring algorithm with pre-scaling method. To make their implementations fairly comparable, the proposed 16-digit radix-100 DXP divider is synthesized with TSMC 0.18-um standard cell library. The synthesis results show that the worst case delay path is 9.0 ns. Since we use different technologies on timing evaluation (TSMC 0.18-um) from [26] (IBM 65-nm), in order to make the delays comparable, the delays is represented in FO4 (inverter delay with a fan out of 4) [128]. For the 0.18-um technology, 1 FO4 $\approx$ 65 ps [25], therefore, the cycle time of the proposed architecture is 138.5 FO4, and the latency is 2907 FO4. The compared results are shown in Table 8.3 which indicate 1) the proposed architecture needs 2 times the size of look-up table to store all of the scaling parameters; 2) the proposed architecture is 2.72 times slower than the 16-digit radix-10 DXP divider [26] because more complex DCSA tree and DCLA adder are implemented in the proposed 16-digit radix-100 DXP decimal divider.

8.5 Summary

In this work, first, we present a radix-100 non-restoring decimal division algorithm with pre-scaling method. Second, we analyze the size of the look-up table and create an additional scaling step to reduce the size of look-up table from $2^{10}\times16$ to $2^{9}\times8$. Third, we describe the architecture of the proposed 16-digit radix-100 DXP decimal divider. Finally, the proposed

Table 8.3: Hardware performance comparison.

	Radix-100 16-digit DXP Divider	**Radix-10** 16-digit DXP Divider [26]
Look-up table size	0.5 KB	0.25 KB
Cycle time	138.5 FO4	13 FO4
No. of cycles	21	82
Latency	2907 FO4	1066 FO4

architecture is implemented on FPGA and is synthesized with TSMC 0.18-um standard cell library. The worst case delay path is mainly produced in a DCSA tree and a 21-digit DCLA adder that act as bottleneck to speedup the radix-100 decimal divider. Since the radix-100 needs to use more complex quotient digit selection part and larger size of the look-up table, the current design is slower and larger than the previous radix-10 design in terms of areas and latency. However, the radix-100 divider in this work can reduce the clock cycle to 1/4 of the previous radix-10 design, while maintaining accuracy of the result. That gives us a room to improve the presented architecture to get shorter latency by using redundant carry-save or signed-digit data-path and some new technologies well used in the basic decimal arithmetic with further investigation.

Part VI

Conclusion

Chapter 9

Summary and Future Research

9.1 Summary

In this dissertation, we have researched and developed several new decimal algorithms and architectures for the computation of decimal transcendental function. The algorithms proposed in this dissertation are simulated with MATLAB or C language, and the corresponding hardware implementations are modeled with VHDL or Verilog, and then simulated using ModelSim. The proposed architectures are verified and synthesized using FPGAs devices or by Synopsys Design Compiler with the CMOS standard cells library. It is expected that the algorithms and architectures presented in this dissertation provide a useful starting point for the future research in the computation of hardware-oriented DFP transcendental function.

This dissertation starts with the works of decimal logarithmic and antilogarithmic converters based on the table-based piecewise linear approximation method in Part III. In order to determine the fewest segments and coefficients of each segment for the first-order polynomial transcendental function approximation, we propose a dynamic non-uniform segmentation method in Chapter 3. The proposed method can approximate the transcendental functions to satisfy accuracy by the linear approximation in which the input, coefficients, and intermediate values are rounded to least bit-width, and can not be achieved by previous static non-uniform segmentation methods. In Chapter 4, we present a new approach (Alg. 2) to compute decimal logarithm and antilogarithm based on the decimal first-order polynomial approximation algorithm. The proposed architecture of logarithmic converter is implemented on an FPGA device, and then compared with a binary-based decimal linear approximation algorithm (Alg. 1). The proposed approach is an attractive method because

logarithm and antilogarithm or other transcendental functions can be evaluated by a set of simple linear approximation which can be implemented by a combinational logic with only a single clock cycle to achieve decimal results. However, when the aim is to achieve more accurate precision of results, such as Decimal64 or Decimal128 DFP transcendental function computations, the proposed architectures become inefficient because the required table lookup size is too large to be implemented in hardware.

In Part IV, the digit-recurrence algorithms with selection by rounding are studied for the DFP transcendental function computations. In Chapter 5 and Chapter 6, we present the algorithm and architecture of the DFP logarithmic and antilogarithmic converters, based on digit-recurrence algorithm with selection by rounding respectively. The proposed converters can compute faithful DFP logarithm and antilogarithm results for any one of the three DFP formats specified in the IEEE 754-2008 standard. These research works start with the algorithms and architectures based on non-redundant data-path. In order to optimize the latency for the proposed design, we include novel features shown as follows: 1) using signed-digit redundant digits, and redundant carry-save representation of the data-path; 2) reducing the number of iterations by determining the number of initial iteration; and 3) retiming and balancing the delay of the proposed architecture. The proposed architectures are synthesized with STM 90-nm standard cell libraries. To estimate the hardware performance, the delay model is obtained according to the logical effort method [128], which estimates the proposed architecture delay values in a technology independent parameter, FO4 unit (the delay of an inverter of the minimum drive strength (1x) with a fanout of four 1x inverters), and the total hardware cost of the proposed architectures are estimated as the number of equivalent 1x two input NAND gate (NAND2). The delay estimation results of the proposed architectures show that the latency of the proposed DFP logarithmic and antilogarithmic converters are close to or better than that of the binary radix-16 logarithmic and exponential converters [94, 57] respectively, and they have a significant decrease in terms of the latency in contrast with a recently published high performance CORDIC implementation [49]. In addition, compared with the software DFP transcendental function computation library [41], the proposed hardware implementation in this work is about thirty to forty times faster than the software implementation.

The Part V is included to present two research works based on the functional iteration methods. Chapter 7 presents a design and implementation of a 16-digit DXP decimal reciprocal unit for the DFP Decimal64 format, in which the decimal reciprocal result is obtained by the initial approximation of the reciprocal by using a look-up table and a multiplication, and then followed by three Newton-Raphson iterations. Since a more comprehensive research based on this approach has been presented in [28], as the main contribution of this work, the presented design utilizes a $2^{10} \times 10$ bits look-up table which is half size of the look-up table size used in [28]. In Chapter 8, a new 16-digit DXP radix-100 decimal divider is designed and implemented based on the decimal non-restoring algorithm with pre-scaling method. This design intends to make the first attempt to analyze and implement a radix-100 iteration algorithm for the complex decimal arithmetic. The radix-100 architecture created in this work is based on the non-redundant data-path, which leads to a large computation latency and is slower than that of radix-10 designs. However, the radix-100 divider can reduce the clock cycle to 1/4 of the previous radix-10 design, while maintaining accuracy of the result. That gives us a room to improve the presented architecture to get shorter latency by using redundant carry-save or signed-digit data-path and some new technologies well used in the basic decimal arithmetic with further investigation.

9.2 Future Research

This dissertation provides a starting point for researchers to further study DFP transcendental arithmetic. There could be more research topics in this area as follows:

9.2.1 Decimal Logarithmic Arithmetic Unit

The logarithmic number system (LNS) is an alternative to floating-point number system (FLP). LNS could speed up some complex computation for multiplication, division and squaring, etc, so the binary LNS is more attractive in applications where a large number of multiplication and divisions are required, such as some applications of digital signal processing and 3-D graphics processing. A future idea is to design and implement a decimal logarithmic arithmetic unit (DLAU) as shown in Figure 9.1, in which real decimal numbers

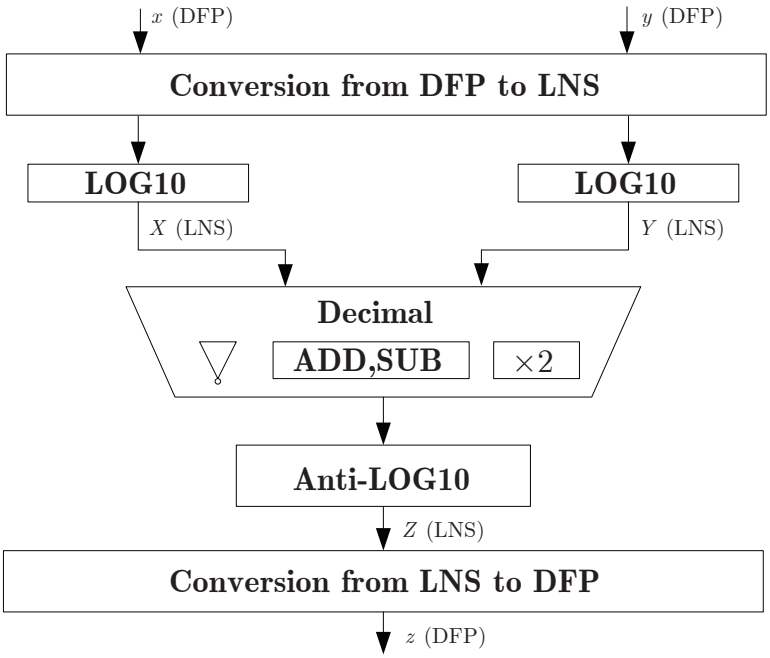

Figure 9.1: Decimal logarithmic arithmetic unit.

and their decimal associated arithmetic are performed well in it. It is assumed that DLAU can perform decimal complex arithmetic units, (such as a division, a multiplication, a reciprocal and square root etc. as shown in Table 9.1) faster and consume less power and areas than decimal DXP or DFP normal arithmetic units with an acceptable precision of the accuracy. In this dissertation, decimal logarithmic and antilogarithmic converters based on the table-based piecewise linear approximation method have been presented in Chapter 4. These two units only take a single clock cycle to obtain the results, which gives a good starting point for the research for the design and implementation of DLAU. However, since the table-based linear approximation approach presented in this dissertation is efficient only for the lower precision of accuracy, more efficient table-based approaches for higher precision, such as the higher polynomial order approximation etc., and corresponding more complex architectures could be the next research step. However, we list three problems for this proposal: 1) What is the motivation of research for DLAU? 2) How to solve the rounding for the computation of the decimal arithmetic by DLAU? 3) How to convert the DFP operand to the decimal LNS operand?

Table 9.1: Operations by decimal logarithmic arithmetic unit.

Arithmetic Operations		**Normal Arithmetic**	**Logarithmic Arithmetic**
Multiplication	**MUL**	$z = x \times y$	$Z = X + Y$
Division	**DIV**	$z = x \div y$	$Z = X - Y$
Reciprocal	**RCP**	$z = 1/x$	$Z = -X$
Squared Root	**SQRT**	$z = \sqrt{x}$	$Z = X/2$
Reciprocal Squared Root	**RSQ**	$z = 1/\sqrt{x}$	$Z = -X/2$
Square	**SQR**	$z = x^2$	$Z = 2X$

9.2.2 A Combined DFP Division/Square Root Unit

Another interesting future research work is to design and implement a single recurrence unit that can compute DFP division, square root, and as many transcendental functions as possible based on a choice of coefficient table and a minimal number of control signals. The hardware-oriented algorithms based on digit-recurrence with selection by rounding are introduced for high-radix binary division, square-root [89, 90, 91], CORDIC [93], logarithm [94] and exponential [92] operations respectively. In Part IV, the algorithm and architecture of DFP logarithmic and antilogarithmic converters based on digit-recurrence with selection by rounding have been designed and implemented separately. However, it is obvious that the most subcomponents in the hardware implementation for these two architectures can be shared in a single recurrence unit. We assume that the proposed decimal digit-recurrence algorithm with selection by rounding can be applied for many other different functions with the minimum extra resource and modification. To anticipate developments of this unit, it might be possible to design the units as loosely-coupled collections of subcomponents (adders, buffer registers etc.) where the connection pattern, sequence information and table contents are loaded as firmware. The merits of this idea would be determined by the degree to which the flexibility would degrade performance on the critical operations, mainly division.

The complication of this work is that the IEEE 754-2008 standard requires the exact rounding for the transcendental functions, and there may be some significant mathematical work required to prove rounding correctness over the value range and all rounding modes.

However, exact rounding for some arithmetic operations, especially for some transcendental factions would be a challenging problem - Table Maker's Dilemma [60]. The exact rounding may be intractable to be done in a hardware unit, so it would be acceptable for the hardware unit to produce an faithful results which could then be refined to the correct value in software. The maximum frequency, number of clock cycles, throughput and latency can be referenced from the existing DFP arithmetic logic unit, such as DFP unit in IBM POWER6 (13 FO4 per clock cycle).

Single Instruction Multiple Data (SIMD) is a technique employed to achieve data level parallelism. In the future, we plan to combine this single recurrence unit and SIMD architecture together to achieve the high performance arithmetic to support current microprocessor. The SIMD supported BFP arithmetic for general-purpose x86-compatible microprocessors has been described in literatures [134, 135, 136], which are good bases for us to build a similar mechanism for DFP arithmetic. Since the digit-width of decimal mantissa and bit-width of exponent in three decimal formats are Decimal32 (6-bit, 7-digit), Decimal64 (8-bit, 16-digit) and Decimal128 (12-bit, 34-digit), a few of the bits of the combined unit would be wasted. IBM POWER6 [9] supports Decimal64 and Decimal128 basic formats directly with arithmetic operations and provides support for converting to and from the Decimal32 storage format. The SIMD architecture provided by IBM POWER6 can operate one higher-precision data path (Decimal128) or two lower-precision data paths (Decimal64) at the same time.

9.2.3 DFP Transcendentals via BID Encoding

In IEEE 754-2008 standard, both Densely Packed Decimal (DPD) [54] and Binary Integer Decimal (BID) [55] encodings are included as two representations for DFP formats. The BID external format should be converted to binary integer, while DPD format to BCD coding during the internal operations. The research works presented in this dissertation for DFP transcendental arithmetic are all based on DPD to BCD, and then BCD coding is operated in DFP arithmetic. The BID format is firstly adopted for Intel's software DFP Math library [55]. Recently, some researchers have worked on integrating BID format into the hardware implementation. In [137], a decimal division based on BID format is presented

and compared with previous BCD one using the same algorithm. However, the architecture based on BID format is not better than previous BCD one, because the binary-based decimal normalization unit in the architecture can not achieve a high speed. A series of researches for DFP arithmetic based on BID format are designed and implemented in recent years [138, 139, 140]. The main bottleneck is on the normalization, addition alignment and rounding units because they are implemented by a binary multiplication and used iteratively in DFP arithmetic. For example, the date-path of DFP adder includes the alignment, normalization and rounding step, and all these steps need to use binary multiplications, which is not efficient for speed. However, we must admit that the advantage of internal binary is that we can use the binary adder and multiplier from the current integer ALU in the microprocessor. Recently, we have designed and implemented a binary based normalization unit which can be modified for rounding or alignment operation for BID format. In the future work, we will do more performance evaluation for BID transcendental arithmetic. We try to answer a question: "which format, BID or BCD, is more efficient for DFP transcendental arithmetic?" We believe it is an interesting research topic to carry on.

References

[1] C. Maxfield, "Binary Coded Decimal (BCD) 101 - Part 1," Apr. 2007. [Online]. Available: http://www.eetimes.com/design

[2] M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *16th IEEE Symposium on Computer Arithmetic (ARITH'16)*, June 2003, pp. 104–111.

[3] A. Tsang and M. Olschanowsky, "A study of dataBase 2 customer queries," IBM Santa Teresa Laboratory, San Jose, CA, USA, IBM Technical Report TR.03.413, 1991.

[4] M. F. Cowlishaw, "Decimal arithmetic FAQ: Part 1 - general questions," Apr. 2010. [Online]. Available: http://speleotrove.com/decimal/decifaq1.html

[5] IBM Coroperation, "The 'telco' benchmark," Mar. 2005. [Online]. Available: http://speleotrove.com/decimal/telcoSpec.html

[6] SUN Microsystem, "BigDecimal class, API specification for the Java 2 platform," 2004. [Online]. Available: http://java.sun.com/j2se/1.3/docs/api/

[7] M. F. Cowlishaw, "The decNumber C library, version 3.68," Jan. 2010. [Online]. Available: http://speleotrove.com/decimal/decnumber.pdf

[8] IEEE, Inc., *IEEE 754-2008 standard for floating-point arithmetic*, Aug. 2008.

[9] L. Eisen, J. W. W. III, H.-W. Tast, N. Mading, J. Leenstra, S. M. Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough, "IBM POWER6 accelerators: VMX and DFU," *IBM Research and Development*, vol. 51, no. 6, pp. 663–683, Nov. 2007.

[10] A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic, "Decimal floating-point in z9: an implementation and testing perspective," *IBM Research and Development*, vol. 51, no. 1/2, pp. 217–227, Jan./Mar. 2007.

[11] E. M. Schwarz, J. S. Kapernick, and M. F. Cowlishaw, "Decimal floating-point support on the IBM system z10 processor," *IBM Research and Development*, vol. 53, no. 1, pp. 4:1–4:10, Jan. 2009.

[12] R. D. Kenney and M. J. Schulte, "High-speed multioperand decimal adders," *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 953–963, Aug. 2005.

[13] L.-K. Wang and M. J. Schulte, "Decimal floating-point adder and multifunction unit with injection-based rounding," in *18th IEEE Symposium on Computer Arithmetic (ARITH'18)*, June 2007, pp. 56–68.

[14] L. Dadda, "Multi operand parallel decimal adder: A mixed binary and BCD approach," *IEEE Transactions on Computers*, vol. 56, no. 10, pp. 1320–1328, Oct. 2007.

[15] L.-K. Wang, M. J. Schulte, J. D. Thompson, and N. Jairam, "Hardware designs for decimal floating-point addition and related operations," *IEEE Transactions on Computers*, vol. 58, no. 3, pp. 322–335, Mar. 2009.

[16] L.-K. Wang and M. J. Schulte, "A decimal floating-point adder with decoded operands and a decimal leading-zero anticipator," in *19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, June 2009, pp. 125–134.

[17] A. Vázquez and E. Antelo, "A high-performance significand BCD adder with IEEE 754-2008 decimal rounding," in *19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, June 2009, pp. 135–144.

[18] S. Gorgin and G. Jaberipur, "Fully redundant decimal arithmetic," in *19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, June 2009, pp. 145–152.

[19] H. Nikmehr, B. Phillips, and C. C. Lim, "A decimal carry-free adder," in *SPIE Symposium Smart Structures, Devices, and Systems II*, Feb. 2005, pp. 786–797.

[20] M. A. Erle, M. J. Schulte, and B. J. Hickmann, "Decimal floating-point multiplication via carry-save addition," in *18th IEEE Symposium on Computer Arithmetic (ARITH'18)*, June 2007, pp. 46–55.

[21] M. A. Erle, B. J. Hickmann, and M. J. Schulte, "Decimal floating-point multiplication," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 902–916, July 2009.

[22] G. Jaberipur and A. Kaivani, "Improving the speed of parallel decimal multiplication," *IEEE Transactions on Computers*, vol. 58, no. 11, pp. 1539–1552, Nov. 2009.

[23] T. Lang and A. Nannarelli, "A radix-10 combinational multiplier," in *IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC '06)*, Nov. 2006, pp. 313–317.

[24] A. Vázquez, E. Antelo, and P. Montuschi, "Improved design of high-performance parallel decimal multipliers," *IEEE Transactions on Computers*, vol. 59, no. 5, pp. 679–693, Mar. 2010.

[25] H. Nikmehr, B. Phillips, and C. Lim, "Fast decimal floating-point division," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, pp. 951–961, Sept. 2006.

[26] E. M. Schwarz and S. R. Carlough, "Power6 decimal divide," in *18th IEEE Application-Specific Systems, Architectures, and Processors (ASAP'07)*, July 2007, pp. 128–133.

[27] A. Vázquez, E. Antelo, and P. Montuschi, "A radix-10 SRT divider based on alternative BCD codings," in *IEEE International Conference Computer Design (ICCD'07)*, Oct. 2007, pp. 46–55.

[28] L.-K. Wang and M. J. Schulte, "A decimal floating-point divider using Newton-Raphson iteration," *VLSI Signal Processing Systems*, vol. 49, no. 1, pp. 3–18, Oct. 2007.

[29] T. Lang and A. Nannarelli, "A radix-10 digit-recurrence division unit: algorithm and architecture," *IEEE Transactions on Computers*, vol. 56, no. 6, pp. 727–739, June 2007.

[30] L.-K. Wang and M. J. Schulte, "Decimal floating-point square root using Newton-Raphson iteration," in *16th IEEE Application-Specific Systems, Architectures, and Processors (ASAP'05)*, July 2005, pp. 305–319.

[31] L.-K. Wang, M. A. Erle, C. Tsen, E. M. Schwarz, and M. J. Schulte, "A survey of hardware designs for decimal arithmetic," *IBM Research and Development*, vol. 54, no. 3, paper 8, Mar./Apr. 2010.

[32] J. M. Muller, *Elementary Functions, Algorithms and Implementation*, 2nd ed. Boston, USA: Birkhäuser Verlag, 2005.

[33] H. Kim, B.-G. Nam, J.-H. Sohn, J.-H. woo, and H.-J. Yoo, "A 231-MHz, 2.18-mW 32-bit logarithmic arithmetic unit for fixed-point 3-D graphics system," *IEEE Solid-State Circuits*, vol. 14, no. 11, pp. 2373–2381, Nov. 2006.

[34] J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec, "Arithmetic on the European logarithmic microprocessor," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 702–715, June 2000.

[35] N. Ide, M. Hirano, Y. Endo, S. Yoshioka, H. Murakami, A. Kunimatsu, T. Sato, T. Kamei, T. Okada, and M. Suzuoki, "2.44-GFLOPS 300-MHz floating-point vector-processing unit for high-performance 3D graphics computing," *IEEE Solid-State Circuits*, vol. 35, no. 7, pp. 1025–1033, July 2006.

[36] S. Vassiliadis, M. Zhang, and J. G. Delgado-Frias, "Elementary function generators for neural-network emulators," *IEEE Transactions on Neural Networks*, vol. 11, no. 6, pp. 1438–1449, Nov. 2000.

[37] S.-C. Huang and L.-G. Chen, "A 32-bit logarithmic number system processor," *VLSI Signal Processing Systems*, vol. 14, no. 3, pp. 311–319, Dec. 1996.

[38] J. Harrison, "Presentation: Decimal transcendentals via binary," June 2009. [Online]. Available: http://www.ac.usc.es/arith19/sites/default/files/S7P2_DecimalTranscendentalsViaBinary.pdf

[39] J. C. Kropa, "Calculator algorithms," *Mathematics Magazine*, vol. 51, no. 2, pp. 106–109, Mar. 1978.

[40] L. Imbert, J. M. Muller, and F. Rico, "A radix-10 BKM algorithm for computing transcendentals on pocket computers," *VLSI Signal Processing Systems*, vol. 25, no. 2, pp. 179–186, June 2000.

[41] J. Harrison, "Decimal transcendentals via binary," in *19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, June 2009, pp. 187–194.

[42] M. Yamazaki, "Digital antilogarithmic converter circuit," Patent and Trademark Office, US patent 4,058,807, Nov. 1977.

[43] S. D. Trong, K. Helwig, and M. Loch, "Digital circuit for calculating a logarithm of a number," Patent and Trademark Office, US patent 5,363,321, Nov. 1994.

[44] F. Nagao and M. Fuma, "Logarithmic value calculation circuit," Patent and Trademark Office, US patent 6,345,285, Feb. 2002.

[45] B. L. Hallse, "Digital base-10 logarithm converter," Patent and Trademark Office, US patent 6,587,070, July 2003.

[46] R. W. Allred, "Circuits, systems, and methods implementing approximations for logarithm, inverse logarithm, and reciprocal," Patent and Trademark Office, US patent 7,171,435, Jan. 2007.

[47] A. J. Morenilla, H. M. Mora, J.-L. S. Romero, and F. P. Lopez, "A fast architecture for radix-10 coordinates rotation," in *3rd Southern Conference on Programmable Logic*, Feb. 2007, pp. 39–44.

[48] J. L. Sanchez, H. Mora, J. Mora, and A. Jimeno, "Architecture implementation of an improved decimal CORDIC method," in *IEEE International Conference Computer Design (ICCD'08)*, Oct. 2008, pp. 95–100.

[49] A. Vaźquez, J. Villalba, and E. Antelo, "Computation of decimal transcendental functions using the CORDIC algorithm," in *19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, June 2009, pp. 179–186.

[50] J.-L. Sanchez, H. Mora, J. Mora, F. J. Ferrandez, and A. Jimeno, "An iterative method for improving decimal calculations on computers," *Elsvier Mathematical and Computer Modelling*, vol. 50, pp. 869–878, Dec. 2009.

[51] A. Jimeno, H. Mora, J. L. Sanchez, and F. Pujol, "A BCD-based architecture for fast coordinate rotation," *Elsevier Systems Architecture*, vol. 54, pp. 829–840, Feb. 2008.

[52] IEEE, Inc., *IEEE Standard for Binary Floating-Point Arithmetic*, Mar. 1985.

[53] ——, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, Mar. 1987.

[54] M. F. Cowlishaw, "Densely packed decimal encoding," *IEE Computers and Digital Techniques*, vol. 149, no. 3, p. 102C104, May 2002.

[55] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev, "A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 148–162, Feb. 2009.

[56] I. D. Castellanos and J. E. Stine, "A 64-bit decimal floating-point comparator," in *18th IEEE Application-Specific Systems, Architectures, and Processors (ASAP'06)*, June 2003, pp. 138–144.

[57] A. Pińeiro, "Algorithms and architectures for elementary function computation," Ph.D. dissertation, University of Santiago de Compostela, Spain, 2003.

[58] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 1st ed. London, U.K.: Oxford Univ. Press, 2000.

[59] I. Koren, "Evaluating elementary functions in a numerical coprocessor based on rational approximations," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1030–1037, Aug. 1990.

[60] V. Lefévre, J. M. Muller, and A. Tisserand, "Toward correctly rounded transcendentals," *IEEE Transactions on Computers*, vol. 47, no. 11, pp. 1235–1243, Nov. 1998.

[61] V. Lefévre, D. Stehlé, and P. Zimmermann, "Worst cases for the exponential function in the IEEE 754r decimal64 format," in *Reliable Implementation of Real Number Algorithms*, Jan. 2008, pp. 114–126.

[62] W. Cody and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, 1980.

[63] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Transactions on Mathematical Software*, vol. 17, no. 1, pp. 26–45, Mar. 1991.

[64] P. Markstein, *IA-64 and Elementary Functions*. Hewlett-Packard Professional Books, 2000.

[65] S. F. Oberman, "Floating-point division and square root algorithms and implementation in the AMD-K7 microprocessor," in *14th IEEE Symposium on Computer Arithmetic (ARITH'14)*, Apr. 1999, pp. 106–115.

[66] M. J. Schulte and E. E. Swartzlander Jr., "Hardware designs for exactly rounded elementary functions," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 964–973, Aug. 1994.

[67] E. W. Cheney, *Introduction to Approximation Theory. International Series in Pure and Applied Mathematics*. New York, NY: McGraw Hill, 1966.

[68] T. J. Rivlin, *An Introduction to the Approximation of Functions*. MA, USA: Republished by Dover, 1981.

[69] M. D. Ercegovac, "A general method for evaluation of functions and computation in a digital computer," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, Urbana-Champaign, IL, 1975.

[70] A. A. Liddicoat, "High-performance arithmetic for division and the elementary functions," Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, Palo Alto, CA, 2002.

[71] M. D. Ercegovac, "A general hardware-oriented method for evaluation of functions and computations in a digital computer," *IEEE Transactions on Computers*, vol. 26, no. 7, pp. 667–682, July 1977.

[72] P. T. P. Tang, "Table look-up algorithms for elementary functions and their error analysis," Argonne National Laboratory, Report, MCS-P194-1190, Jan. 1991.

[73] W. F. Wang and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 278–294, Mar. 1994.

[74] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 842–847, Aug. 1999.

[75] F. D. Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, Mar. 2005.

[76] W. Gautschi and G. H. Golub, *Applications and computation of orthogonal polynomials, International series of numerical mathematics*, G. Opfer, Ed. Basel: Birkhäuser, 1999.

[77] D. D. Sarma and D. W. Matula, "Faithful interpolation in reciprocal tables," in *13th IEEE Symposium on Computer Arithmetic (ARITH'13)*, June 1997, pp. 82–91.

[78] D. D. Sarma and D. Matula, "Faithful bipartite ROM reciprocal tables," *IEEE Transactions on Computers*, vol. 47, no. 11, pp. 1216–1222, Nov. 1998.

[79] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," in *12th IEEE Symposium on Computer Arithmetic (ARITH'12)*, June 1995, pp. 10–16.

[80] N. Takagi, "Powering by a table lookcup and a multiplication with operand modification," *IEEE Transactions on Computers*, vol. 47, no. 11, pp. 1216–1222, Nov. 1998.

[81] J. M. Muller, "A few results on table-based methods," *Reliable Computing*, vol. 5, no. 3, pp. 279–288, Oct. 1999.

[82] S. L. SanGregory, R. E. Siferd, C. Brother, and D. Gallagher, "A fast, low-power logarithm approximation with cmos vlsi implementation," in *IEEE 39th International Midwest Symposium Circuits and Systems (MWSCAS'99)*, Aug. 1999, pp. 388–391.

[83] F. D. Dinechin and A. Tisserand, "Some improvements on multipartite table methods," in *15th IEEE Symposium on Computer Arithmetic (ARITH'15)*, June 2001, pp. 128–135.

[84] J. A. Pińeiro, J. D. Bruguera, and J. M. Muller, "Faithful powering computation using table look-up and a fused accumulation tree," in *15th IEEE Symposium on Computer Arithmetic (ARITH'15)*, June 2001, pp. 40–47.

[85] K. H. Abed and R. E. Siferd, "Cmos vlsi implementation of a low-power logarithmic converter," *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1421–1433, Nov. 2003.

[86] J. E. Stine and M. J. Schulte, "The symmetric table addition method for accurate function approximation," *VLSI Signal Processing*, vol. 21, no. 2, pp. 167–177, June 1999.

[87] K. H. Abed and R. E. Siferd, "Vlsi implementation of a low-power antilogarithmic converter," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1221–1228, Sept. 2003.

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

[89] M. D. Ercegovac, T. Lang, and P. Montuschi, "Very high-radix division with prescaling and selection by rounding," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 909–918, May 1994.

[90] T. Lang and P. Montuschi, "Very-high radix square root with prescaling and rounding and a combined division/square root unit," *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 827–841, Aug. 1999.

[91] E. Antelo, T. Lang, and J. D. Bruguera, "Computation of $\sqrt{x/d}$ in a very-high radix combined division/square-root unit with scaling and selection by rounding," *IEEE Transactions on Computers*, vol. 47, no. 2, pp. 152–161, Feb. 1998.

[92] A. Pińeiro, M. D. Ercegovac, and J. D. Bruguera, "Algorithm and architecture for logarithm, exponential, and powering computation," *IEEE Transactions on Computers*, vol. 53, no. 9, pp. 1085–1096, Sept. 2004.

[93] E. Antelo, T. Lang, and J. Bruguera, "High-radix CORDIC rotation based on selection by rounding," *VLSI Signal Processing Systems*, vol. 25, no. 2, pp. 141–153, June 2000.

[94] A. Pińeiro, M. D. Ercegovac, and J. D. Bruguera, "High-radix logarithm with selection by rounding: algorithm and implementation," *VLSI Signal Processing Systems*, vol. 40, no. 1, pp. 109–123, Mar. 2005.

[95] J.-C. Bajard, S. Kla, and J.-M. Muller, "Bkm: a new hardware algorithm for complex elementary functions," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 955–963, Aug. 1994.

[96] M. J. Flynn, "On division by functional iteration," *IEEE Transactions on Computers*, vol. 19, no. 8, pp. 702–706, Aug. 1970.

[97] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 833–854, Aug. 1997.

[98] U. Kucukkabak and A. Akkas, "Design and implementation of reciprocal unit using table look-up and Newton-Raphson iteration," in *IEEE EUROMICRO Systems on Digital System Design (DSD'04)*, Sept. 2004, pp. 249–253.

[99] K. E. Wires and M. J. Schulte, "Reciprocal and reciprocal square root units with operand modification and multiplication," *VLSI Signal Processing*, vol. 42, no. 3, pp. 257–272, Mar. 2006.

[100] R. E. Goldschmidt, "Applications of Division by Convergence," Master's thesis, Electrical Engineering Dept., Massachussets Institute of Technology (MIT), June 1964.

[101] M. D. Ercegovac, L. Imbert, D. W. Matula, J.-M. Muller, and G. Wei, "Improving Goldschmidt division, square root and square root reciprocal," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 759–763, July 2000.

[102] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A K Peters, Ltd., 2001.

[103] J.-P. Deschamps, G. J. A. Bioul, and G. D. Sutter, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*, 1st ed. Wiley, 2006.

[104] Y. You, Y. Kim, and J. Choi, "Dynamic decimal adder circuit design by using the carry lookahead," in *IEEE Design and Diagnostics of Electronic Circuits and systems (DDECS'06)*, Apr. 2006, pp. 242–244.

[105] A. Vaźquez and E. Antelo, "Conditional speculative decimal addition," in *7th Conference on Real Numbers and Computers (RNC'7)*, July 2006, pp. 47–57.

[106] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, Mar. 2005.

[107] J. Cao, B. W. Y. Wei, and J. Cheng, "High-performance architectures for elementary function generation," in *15th IEEE Symposium on Computer Arithmetic (ARITH'15)*, June 2001, pp. 136–144.

[108] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *16th IEEE Application-Specific Systems, Architectures, and Processors (ASAP'05)*, July 2005, pp. 328–333.

[109] J. Pièiro, S. Oberman, J.-M. Muller, and J. Bruguera, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 304–318, Mar. 2005.

[110] T. Sasao, S. Nagayama, and J. Butler, "Numerical function generators using LUT cascades," *IEEE Transactions on Computers*, vol. 56, no. 6, pp. 826–838, June 2007.

[111] D.-U. Lee, R. C. C. Cheung, W. Luk, and J. D. Villasenor, "Hierarchical segmentation for hardware function evaluation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 103–116, Jan. 2009.

[112] B. Lee and N. Burgess, "Some approximations on taylor-series function approximation on FPGA," in *Asilomar Conference Circuits, Systems, and Computers*, Nov. 2003, pp. 2198–2202.

[113] D. M. Lewis, "Interleaved memory function interpolators with application to an accurate LNS arithmetic unit," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 974–982, Aug. 1994.

[114] D.-U. Lee and J. D. Villasenor, "A bit-width optimization methodology for polynomial-based function evaluation," *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 567–571, Apr. 2007.

[115] F. K. Hanna and A. K. Misra, "Hardware realisation of binary search algorithm," *IET Computers and Digital Techniques*, vol. 127, no. 4, pp. 148–151, July 1980.

[116] D. Chen, Y. Choi, D. Teng, K. Wahid, and S. Ko, "A novel decimal-to-decimal logarithmic converter," in *IEEE Symposium on Circuit and System (ISCAS'08)*, May 2008, pp. 688–691.

[117] D. Chen, Y. Zhang, L. Chen, D. Teng, K. Wahid, and S. Ko, "A decimal-to-decimal antilogarithmic converter," in *IEEE Canidian Electrical and Computer Engineering (CCECE'08)*, May 2008, pp. 1223–1226.

[118] J. N. Mitchell Jr., "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, vol. 11, pp. 512–517, Aug. 1962.

[119] M. Combet, H. Zonneveld, and L. Verbeek, "Computation of the base two logarithm of binary numbers," *IEEE Transactions on Electronic Computers*, vol. 14, pp. 863–867, Dec. 1965.

[120] E. L. Hall, D. D. Lynch, and S. J. Dwyer, "Generation of products and quotients using approximate binary logarithms for digital filtering applications," *IEEE Transactions on Computers*, vol. 19, pp. 97–105, Feb. 1970.

[121] Xilinx Inc., *Xilinx University Program XUPV5-LX110T Development System, Hardware Reference Manual*, June 2009.

[122] D. Chen, Y. Zhang, Y. Choi, M. H. Lee, and S. Ko, "A 32-bit decimal floating-point logarithmic converter," in *19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, June 2009, pp. 195–203.

[123] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Elsevier Microprocessors & Microsystems*, vol. 31, no. 8, pp. 537–545, Dec. 2007.

[124] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 124–128, Mar. 1994.

[125] M. A. Erle and M. J. Schulte, "Decimal multiplication via carry-save addition," in *14th IEEE Application-Specific Systems, Architectures, and Processors (ASAP'03)*, June 2003, pp. 348–358.

[126] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, no. 8, pp. 786–793, Aug. 1973.

[127] STMicroelectronics, "90nm CMOS090 Design Platform," 2007. [Online]. Available: http://www.st.com/stonline/products/technologies/soc/90plat.htm

[128] I. Sutherland, R. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, 1st ed. Morgan Kaufmann, 1999.

[129] Intel Corporation, *Using Decimal Floating-Point with Intel C++ Compiler*. http://software.intel.com/en-us/articles/using-decimal-floating-point-with-intel-c-compiler, 2010.

[130] D. Chen, Y. Zhang, D. Teng, K. Wahid, M. H. Lee, and S. Ko, "A new decimal antilogarithmic converter," in *IEEE Symposium on on Circuit and System (ISCAS'09)*, May 2009, pp. 445–448.

[131] M. J. Schulte and J. E. Stine, "Symmetric bipartite tables for accurate function approximation," in *13th IEEE Symposium on Computer Arithmetic (ARITH'13)*, July 1997, pp. 175–183.

[132] N. Takagi, "Generating a power of an operand by a table look-up and a multiplication," in *13th IEEE Symposium on Computer Arithmetic (ARITH'13)*, July 1997, pp. 126–131.

[133] M. D. Ercegovac and T. Lang, "On-the-fly conversion of redundant into conventional representations," *IEEE Transactions on Computers*, no. 7, pp. 895–897, Aug. 1987.

[134] M. J. Schulte, D. Tan, and C. E. Lemonds, "Floating-point division algorithms for an x86 microprocessor with a rectangular multiplier," in *IEEE International Conference Computer Design (ICCD'07)*, Oct. 2007, pp. 304–310.

[135] A. Danysh and D. Tan, "Architecture and implementation of a vector/simd multiply-accumulate unit," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 284–293, Mar. 2005.

[136] D. Tan, C. E. Lemonds, and M. J. Schulte, "Low-power multiple-precision iterative floating-point multiplier with SIMD support," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 284–293, Feb. 2009.

[137] T. Lang and A. Nannarelli, "Division unit for binary integer decimals," in *20th IEEE Application-Specific Systems, Architectures, and Processors (ASAP'09)*, July 2009, pp. 1–7.

[138] C. Tsen, M. J. Schulte, and S. G. Navarro, "Hardware design of a binary integer decimal-based IEEE P754 rounding unit," in *18th IEEE Application-Specific Systems, Architectures, and Processors (ASAP'07)*, July 2007, pp. 9–11.

[139] C. Tsen, S. G. Navarro, and M. J. Schulte, "Hardware design of a binary integer decimal-based floating-point adder," in *IEEE International Conference Computer Design (ICCD'07)*, Oct. 2007, pp. 288–295.

[140] S. G. Navarro, C. Tsen, and M. J. Schulte, "A binary integer decimal-based multiplier for decimal floating-point arithmetic," in *IEEE signals, systems and computers (ACSSC'07)*, Nov. 2007, pp. 353–357.