Decimal Floating-point Fused Multiply Add with Redundant Number Systems

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

Liu Han

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
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan
Canada
S7N 5A9

Abstract

The IEEE standard of decimal floating-point arithmetic was officially released in 2008. The new decimal floating-point (DFP) format and arithmetic can be applied to remedy the conversion error caused by representing decimal floating-point numbers in binary floating-point format and to improve the computing performance of the decimal processing in commercial and financial applications. Nowadays, many architectures and algorithms of individual arithmetic functions for decimal floating-point numbers are proposed and investigated (e.g., addition, multiplication, division, and square root). However, because of the less efficiency of representing decimal number in binary devices, the area consumption and performance of the DFP arithmetic units are not comparable with the binary counterparts.

IBM proposed a binary fused multiply-add (FMA) function in the POWER series of processors in order to improve the performance of floating-point computations and to reduce the complexity of hardware design in reduced instruction set computing (RISC) systems. Such an instruction also has been approved to be suitable for efficiently implementing not only stand-alone addition and multiplication, but also division, square root, and other transcendental functions. Additionally, unconventional number systems including digit sets and encodings have displayed advantages on performance and area efficiency in many applications of computer arithmetic.

In this research, by analyzing the typical binary floating-point FMA designs and the design strategy of unconventional number systems, "a high performance decimal floating-point fused multiply-add (DFMA) with redundant internal encodings" was proposed. First, the fixed-point components inside the DFMA (i.e., addition and multiplication) were studied and investigated as the basis of the FMA architecture. The specific number systems were also applied to improve the basic decimal fixed-point arithmetic. The superiority of redundant number systems in stand-alone decimal fixed-point addition and multiplication has been proved by the synthesis results. Afterwards, a new DFMA architecture which exploits the specific redundant internal operands was proposed. Overall, the specific number system improved, not only the efficiency of the fixed-point addition and multiplication inside the FMA, but also the architecture and algorithms to build up the FMA itself.

The functional division, square root, reciprocal, reciprocal square root, and many other functions, which exploit the Newton's or other similar methods, can benefit from the proposed DFMA architecture. With few necessary on-chip memory devices (e.g., Look-up tables) or even only software routines, these functions can be implemented on the basis of the hardwired FMA function. Therefore, the proposed DFMA can be implemented on chip solely as a key component to reduce the hardware cost. Additionally, our research on the decimal arithmetic with unconventional number systems expands the way of performing other high-performance decimal arithmetic (e.g., stand-alone division and square root) upon the basic binary devices (i.e., AND gate, OR gate, and binary full adder). The proposed techniques are also expected to be helpful to other non-binary based applications.

Acknowledgements

The entire research is sponsored by the Electrical and Computer Engineering department in University of Saskatchewan and the Natural Science and Engineering Research Council (NSERC) of Canada. All the toolkits and standard cell libraries used in this research are provided by CMC Microsystems, Canada.

First of all, I would like to thank my supervisor Dr. Seok-Bum Ko. In the first year of my Ph.D. program, I took one course which is lectured by Dr. Ko. We discussed a lot about my Ph.D. project during that time. He provided me many inspirations by his experiences on not only research but also life philosophy. Without Dr. Ko's support, this research would not be finished or even started. I would like to thank other professors in our university. Without the helps from Dr. Li Chen, the evaluation works may not be finished quickly. Dr. Aryan Saadat Mehr, Dr. Anh van Dinh, Dr. Chip Hong Chang (Nanyang Technological University), and Dr. Raymond J. Spiteri provided many helpful ideas and suggestions to improve the quality of the research and the thesis. The lab manager, Trevor Zintel, also showed his patience to guarantee that the toolkits were working properly. I would like to thank my friends in our lab for their kind advices, helps, and supports. Finally, I would like to thank my wife, Lidan Hu, and my parents for their patience and supports which accompanied me during the hardest time in my life.

To my father, Han Shuyu, and my son, Han Tianen.

Rest in peace.

Contents

List of Tables

List of Figures

List of Abbreviations

ADP	Area Delay Product
BCD	Binary Coded Decimal
BFA	Binary Full Adder
BFP	Binary Floating-point
CFA	Carry Free Adder
CLA	Carry Lookahead Adder
CPA	Carry Propagate Adder
CSA	Carry Save Adder
DFP	Decimal Floating-point
DFMA	Decimal Floating-point Fused Multiply-Add
DPD	Densely Packed Decimal
DSD	Decimal Signed Digit
FA	Full Adder
FADD	Floating-point Addition
FDIV	Floating-point Division
FMA	Fused Multiply-add
FMUL	Floating-point Multiplication
FSQRT	Floating-point Square Root
FXP	Fixed-point
HA	Half Adder
LSB	Least Significant Bit
LSD	Least Significant Digit
LUT	Look-up Table
LZA	Leading Zero Anticipator
LZD	Leading Zero Detector
MSB	Most Significant Bit
MSD	Most Significant Digit
NaN	Not a Number
PDP	Power Delay Product
PPA	Partial Product Accumulation
PPG	Partial Product Generation
PPR	Partial Product Reduction
RISC	Reduced Instruction Set Computing
RTA	Round Ties to Away
RTE	Round Ties to Even
RTN	Round Toward Negative
RTP	Round Toward Positive
RTZ	Round Toward Zero
SD	Signed Digit
SDDA	Signed Digit Decimal Adder
SIMD	Single Instruction Multiple Data
TZD	Trailing Zero Detection
ulp	unit of least precision

Part I

Preface

Chapter 1

Introduction

This chapter introduces the basic principles of floating-point arithmetic. Afterwards, the necessity of decimal floating-point arithmetic is revealed by analyzing the limits of binary floating-point processing in section 1.1. High-performance computational demands are required by financial and commercial applications in which decimal computing is dominant; moreover, limited efficiency restricts the existing decimal floating-point solutions. These conditions lead to in the motivations of this research which are described in section 1.2. Finally, section 1.3 provides the layout of this thesis.

1.1 Background of the Decimal Floating-point

Computer arithmetic is a vital element in the functionality of any computer system. Two major types of data, integer and floating-point, are processed by computer to mimic computation of the real number. An integer is simply defined and processed by the basic binary devices. However, floating-point arithmetic involves procedures much more complicated than these underlying integer data.

1.1.1 Floating-point Number

A floating-point number is comprised of four elements: sign, significand, exponent, and an implicit base number. Therefore, a floating-point number F is defined as:

$$F = (-1)^S \times C \times B^E \tag{1.1}$$

where $S, C, E,$ and B represent sign, significand, exponent, and base number respectively.

In representing an infinite real number (i.e., $\pi = 3.141592\ldots$) with a finite hardware resource, the precision problem emerges. However, a floating-point number has two important attributes, which are range and precision. The range that defines the maximum and minimum representable numbers in a given floating-point format depends mainly on the number of bits to represent the exponent. Note that the exponent represents the power of the base number, and the floating-point axis is therefore not divided equally by the numbers on it. Additionally, the zero is a special case in floating-point, having an arbitrary exponent. Consequently, the maximum number far away from zero is represented as $\pm C_{max} \times B^{E_{max}}$, and the minimum number close to zero is represented as $\pm C_{min} \times B^{E_{min}}$. A fragment of the floating-point axis is shown in Fig. 1.1:

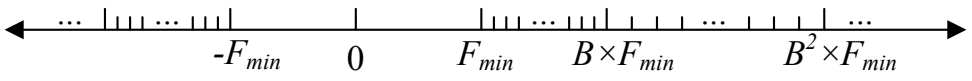


Figure 1.1: The layout of the floating-point axis

Once an exponent is given, the precision of a floating-point format indicates how many non-zero numbers (i.e., $B^P - 1$, where P is the number of digits in the significand) can be exactly represented in the given exponent window which is defined by B^E. For example, 9 numbers (e.g., .1 to .9) are exactly representable if P is 1, B is 10, and E is 0.

The precision of a given floating-point format always implies another useful parameter, which is the so-called *unit of least precision (ulp)*. The ulp is defined as the non-zero minimum value of the least significant digit. If the decimal point is implicitly on the left of the most significant digit, 1 ulp equals $1/(B^P)$. Note also that there are several slightly different definitions of ulp summarized in [1]. In this thesis, the definition mentioned above is chosen.

Since infinite real numbers (e.g., $\pi = 3.141592\ldots$) cannot be exactly represented by the floating-point format, the difference between a real number and the corresponding floating-point number termed the *round off error* is able to be measured by the ulp. In Fig. 1.2, the differences between the exact result (R_{exact}) and the possible rounded results ($R_{rounded}$)

are shown. In different rounding modes, the exact result will be rounded to the left or right closest number which is exactly represented within the format. Therefore, the maximum round off error could be 1 ulp, or 0.5 ulp at maximum, depending on different modes and the value of the exact result.

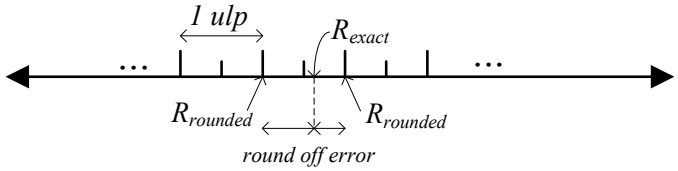


Figure 1.2: Example of round off error

Currently, the binary floating-point which employs 2 as base (i.e., it uses "0" and "1" in significand) is dominant in computer systems. A technical standard (i.e., IEEE 754) about the floating-point arithmetic was established by the IEEE in 1985, whereafter it was revised in 2008.

1.1.2 Why is Decimal Floating-point Arithmetic Necessary?

Although binary arithmetic has shown its advantages in processing speed, hardware complexity, and storage compactness (as everyone knows), decimal arithmetic has already been used in the first days of modern electronic computers [2, 3]. In the applications where binary arithmetic is dominant, either the fixed-point (e.g., integer) arithmetic is competent for the tasks, or the tasks are tolerant of the rounding error. Nonetheless, this situation is not always true in some specific applications. For example, in financial computing, both very small numbers, such as 2.6×10^{-5}, and very large numbers, such as 1.9×10^{9}, need to be dealt with according to the amount and unit in the transactions. Note that these monetary data are represented in scientific notation with a decimal base. Consequently, a *representation error* could arise when converting the decimal monetary data to binary floating-point format. For instance, a decimal number 0.4 in binary floating-point single precision format is about "0.4000000059604645...". The small difference between the binary approximation and original decimal number could cause a 1 ulp error or even more than that in some cases. Furthermore, such a tiny difference could be accumulated during the computation and may

cause serious problems.

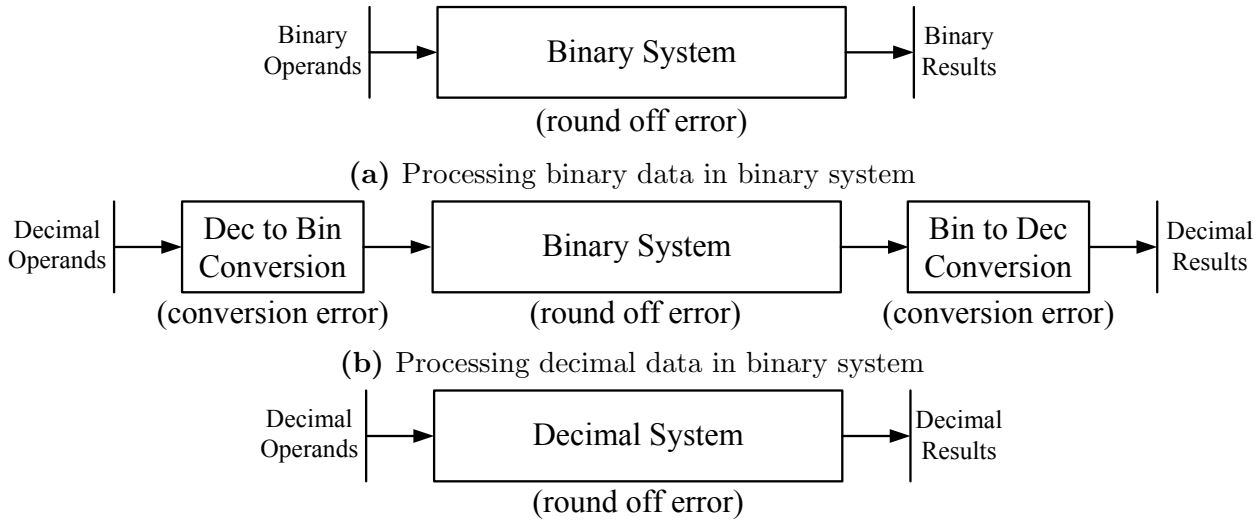


Figure 1.3: Representation errors created in the computation of decimal data in binary system

In Fig. 1.3, three computation models are given to illustrate the error generation during the decimal processing. The traditional binary floating-point processing is shown in Fig. 1.3(a). The computation error occurs mainly because of the finite precision of the hardware, which can be solved by iterative computations in software. Prior to the decimal floating-point standard, the decimal floating-point data are processed in the binary system, as shown in Fig. 1.3(b). However, the conversion error, as described in the first paragraph of this section, makes the problem complicated. First, this conversion error is fed into the binary system. After computations inside the binary system, the error could then become accumulated. Second, once rounded and converted from the binary number back to decimal form, if the exact result and the hardware result are not on the same side of the half ulp, a 1 ulp error may be created in some rounding modes. This error is shown in Fig. 1.4.

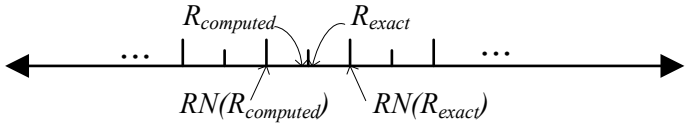


Figure 1.4: Example of the 1 ulp error in decimal processing

In financial applications, the monetary data sometimes is necessary to be rounded to

cent. For example, if a cell phone call costs 1.3 CAD, with 5 percent of tax, the bill of this call will be $1.3 \times 1.05 = 1.365$ CAD, and the rounded bill is 1.37 CAD. However, the real computation in a binary system with single precision binary floating point library can be $1.2999999523162842\cdots \times 1.0499999523162842\cdots = 1.364999771118164\ldots$, and the rounded result is 1.36 CAD. The error is much larger than 1 ulp. A benchmark demonstrated that in a large telephone bill system, the errors could be accumulated up to about 5 million dollars for every single year [4].

To solve the errors caused by representing decimal fractional numbers in binary floating point format, software solutions can be applied to obtain more accurate results by iterative algorithms or converting operands to integer space. For instance, Intel provided a software library to define and process the DFP numbers in Windows, Linux, and HP-UX [13]. Other programming languages and open source libraries that support DFP arithmetic can be found in [14–16]. The benefit of software solution is the flexibility on various platforms. However, the hardware solution is superior in processing speed and energy efficiency. The experiments on different benchmarks showed that the hardware solution can be typically 100 to 1000 times faster than the software solution [16, 18, 19].

In 2003, IBM proposed a decimal floating-point (DFP) hardware solution for financial computing and other similar commercial applications [5]. Meanwhile, some decimal fixed-point and floating-point arithmetic units were implemented and integrated into IBM's processors and mainframes [7–12]. Recently, many basic arithmetic operations of hardware solutions were proposed. For example, Wang et al. proposed a DFP adder and multifunction unit with injection-based rounding [20, 21]. Hickmann et al. provided an parallel DFP multiplier in [22], and Lang et al. announced a 16-digit decimal SRT division algorithm in [23]. The architectures and algorithms for transcendental functions are also investigated in [24, 25]. With continuously increasing demands in decimal computation and decreasing sizes of the transistor on the integrated chip, the hardware solution has gained popularity for commercial and financial applications.

Because of the importance seen in decimal arithmetic, in 2008, DFP format and operations were included in the latest version of the IEEE standard for floating-point arithmetic (IEEE 754-2008) [17]. In this thesis, a pure hardware architecture for decimal floating-point

processing as shown in Fig. 1.3(c) is discussed.

1.2 Motivation of Research

There are several different methods for implementing decimal floating-point arithmetic on chip. First, a decimal adder with necessary logics can be implemented on chip, and other functions can be computed serially (i.e., digit-by-digit) by the decimal adder. The advantage of this method is its much lessened hardware area in comparison to that taken by any other methods. However, the processing speed is worse than the levels achieved by the other methods. Second, separate function units can be implemented on chip. Therefore, the functions on chip can be performed simultaneously to achieve a better performance. In contrast to the previous method, the hardware area, local routing, and power consumption may be considerable. Third, a fused multiply add (FMA) function can be implemented on chip, and other functions can reuse the FMA hardware with necessary hardware or software support. With the on-chip FMA, the hardware requirement of other functions would be minimized or even eliminated. For example, the addition and multiplication could be implemented simply by setting the operands of the FMA function properly (i.e., $(A \times 1) + C$ and $(A \times B) + 0$). Furthermore, the other numerical functions can then be mathematically evaluated by a series of additions and multiplications, as shown in equation 1.2:

$$\begin{aligned} f(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)\ldots)) \end{aligned} \tag{1.2}$$

Since this method provides a hardware solution which is balanced on performance and cost, it has been supported in several commercial processor architectures, such as Intel Itanium and IBM POWER series microprocessors [28, 29].

In the past decade, many such decimal hardware solutions have been investigated. In these designs, unconventional encoding systems have shown the advantages on processing performance. Since the traditional binary decimal encoding (BCD) that applies 4 binary bits to represent 10 decimal numbers is not fully exploited, the representation space is wasted, which causes a lower performance. Additionally, the computational efficiency of the decimal

processing upon binary hardware devices (e.g., AND, OR gates, and full adder) is limited by the inefficient representation of the decimal numbers. In this thesis, we focus on the efficient number systems for decimal processing on binary devices. The redundant number systems that include unconventional digit sets and encodings are studied and examined, not only to represent the decimal number efficiently, but also to process the decimal data on a register transistor level and on an architectural level.

As analyzed above, the research covers the following topics:

1, the number system and related hardware design to improve the efficiency of basic decimal processing,

2, the high performance algorithm and architecture of basic decimal arithmetics, such as addition and multiplication,

3, the high performance architecture of decimal floating-point fused multiply-add.

In a word, the efficient algorithms, architectures, and encodings encouraging a better decimal floating-point fused multiply-add are studied and investigated in this research in order to improve the performance of decimal floating-point processing.

1.3 Overview of Research

In this thesis, the decimal fixed-point addition and multiplication are firstly studied. The application of the unconventional digit sets and encodings is therefore examined. Afterwards, a new two steps non-speculative carry free decimal adder is proposed to decrease the delay and hardware cost at the same time. Additionally, a new decimal fixed-point parallel multiplier for high throughput application is proposed to create less partial products without carry propagation. A hybrid carry propagation network is also created to efficiently accumulate the final product at the last step of the proposed parallel multiplier. Furthermore, a fixed-point sequential decimal multiplier utilizing two necessary multiples and an on-the-fly digit set converter is proposed to perform decimal multiplication with less hardware cost. Subsequently, the decimal floating-point fused multiply-add which exploits our fixed-point addition and parallel multiplication is investigated. With the help of the unconventional number system, the architecture of the proposed FMA can be significantly optimized. Moreover, the

proposed design follows the definition of such an operation in the IEEE standard. Thus, all necessary flags and special operands are supported in such designs. By exploiting the unconventional digit sets and encodings in our designs, not only the performance of decimal processing is improved, but the understanding of redundant number system for non-binary data processing is also expanded.

This thesis is organized as follows to present the research works:

Part I: **Preface** includes:

Chapter 1: **Introduction** presents the basics of the floating-point number, why the decimal floating-point is needed, and the motivation of the research.

Part II: **Research Background** includes:

Chapter 2: **Decimal Floating-point Standard** introduces the IEEE 754-2008 standard which defines the basics of the decimal floating-point format, the special cases of the operands, the rounding modes, and the exception handling.

Chapter 3: **Fused Multiply-Add** gives basic concepts of the fused multiply-add operation. Additionally, the state-of-the-art algorithms and architectures proposed in the binary designs are briefly introduced to draw the consideration of the decimal designs.

Chapter 4: **Number Systems** bring the fundamentals of the number system (i.e., digit-sets and encodings) which is part of the scope of this research.

Part III: **Designs** include:

Chapter 5: **Previous Designs** discuss the state-of-the-art designs, which include the decimal fixed-point addition, multiplication, and the decimal floating-point fused-multiply-add, proposed in the literature. These previous works are reviewed to analyze the possible improvement on these existing designs.

Chapter 6: **Proposed Designs** describe the new designs proposed during this

research. The algorithms and architectures of the new designs are discussed in details.

Chapter 7: **Comparison and Discussion** analyze the differences between the previous designs and our proposed designs. The improvement to achieve the better performance is discussed.

Part IV: **Conclusion** includes:

Chapter 8: **Summary and Future Research** conclude the research result and propose the future works.

Part II

Research Background

Chapter 2

Decimal Floating-point Standard

This chapter introduces the basics of the IEEE standard of floating-point arithmetic (i.e., IEEE 754-2008). As a warm up, the format is first described briefly. Furthermore, the special operands, rounding modes, exception conditions, and flags are introduced respectively. The details of the contents described in this chapter are reviewed and discussed in later chapters.

2.1 Basics of Decimal Floating-point Standard

The representation error caused by the conversion from decimal floating-point to binary floating-point was introduced in section 1.1. However, in the database systems in the fields of banking, financial analysis, retail sales, and etc, over 55% of data are in decimal format [5]. Because of the inefficiency of the current binary hardware, the decimal processing overhead could be over 90% in order to achieve accurate results in decimal format [4]. A decimal floating-point specification was introduced in 2001 for obtaining accurate decimal floating-point results with appropriate efficiency in financial and commercial applications [6]. New data type and necessary arithmetics of the data were further introduced by M. F. Cowlishaw in [5]. By the help of many researchers, a new floating-point industry standard (i.e., IEEE 754-2008) has been released in 2008 [17]. In IEEE 754-2008, a family of commercially feasible ways to perform decimal floating-point arithmetic was defined. Four basic aspects are introduced in next subsections.

2.1.1 Basic Format

In IEEE 754-2008, a decimal floating-point number is represented by sign (S), significand (M), exponent (E), and implied base ($B = 10$). A signed decimal floating-point number is

therefore represented as:

$$(-1)^S \times M \times B^E \tag{2.1}$$

where,

S is 0 or 1,

E is in $[E_{min}, E_{max}]$,

M is in [0,B), and M is represented as $d_0 \cdot d_1 d_2 ... d_{(p-1)}$ (p means the precision).

If M is represented as shown above, the decimal floating-point is viewed as a scientific form, in which a radix point is right after the first digit. In some cases, it is convenient to represent the significand as an integer. Thus, the signed decimal floating-point number is represented as:

$$(-1)^S \times C \times B^Q \tag{2.2}$$

where,

S is 0 or 1,

Q is any integer $E_{min} \leq Q + p - 1 \leq E_{max}$,

C is any integer $0 \leq C < B^p$ (p means the precision).

By defining the maximum and minimum exponents and different precisions, the decimal floating-point numbers are divided into three formats.

Table 2.1: Parameters of decimal floating-point numbers

parameter	decimal32	decimal64	decimal128
p, digits	7	16	34
E_{max}	+96	+384	+6144

In Table 2.1, only E_{max} is defined. The E_{min} is calculated by $1 - E_{max}$. In hardware,

a decimal floating-point number is represented in three segments (i.e., the base number is implicit). The layout of the hardware representation of a decimal floating-point number is illustrated in Fig. 2.1. Since the range of the exponent is fixed, a bias is added to exponent in each format in order to simplify the exponent processing. The details of the number of bits for each segment can be found in [17].

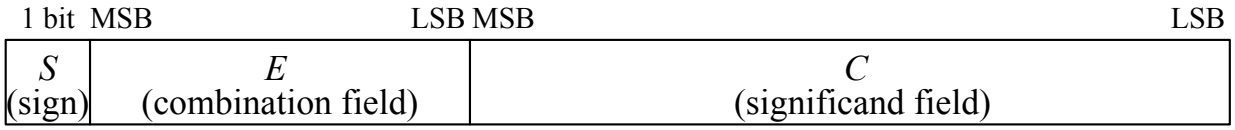


Figure 2.1: The layout of the bits to represent decimal floating-point

2.1.2 Special Format

In IEEE 754-2008, several special numbers are defined. These numbers have to be processed correctly in the hardware solution which is standard compliant.

Infinity

Every finite number can be represented in the equation (2.2). However, if the magnitude of a decimal floating-point number is larger than that of the representable largest number, it is defined as infinity. The computations with infinity are usually exact. For example, $(\infty + x) = \infty$ and $(\infty \times x) = \infty$. Therefore, the exception signals are not set for these computations. However, in some special cases, the exception signals will be set up. For example, $(x \div 0) = \infty$, or overflow raised by finite computations. The detailed definition of such a number can be found in section 6.1 in [17].

NaNs

The NaN, which stands for *Not a Number*, includes two different cases (i.e., signaling and quiet). The signaling NaN or sNaN provides representations for uninitialized variables in memory and other enhancements of arithmetic which are not defined in the standard. The quiet NaN or qNaN provides diagnostic information for invalid or unavailable results. For example, $(0 \times \infty)$, $(+\infty + -\infty)$, and $(0 \div 0)$ have no valid results. The invalid flag is therefore

set and an NaN can be given as the result. The detailed definition of such a number can be found in sections 6.2 and 7.2 in [17].

2.1.3 Rounding Modes

If the result cannot be exactly represented in a give format (i.e., decimal64 or decimal128), the result has to be rounded to the number which is the closest to the true result. The difference between the rounded result and the true result is the so-called rounding error. The standard defines five rounding modes that have different maximum rounding errors. The examples of the rounding modes are given in Table 2.2.

Table 2.2: The rounding directions of different rounding modes

inputs	RTP	RTN	RTZ	RTE	RTA
5.4	6	5	5	5	5
5.5	6	5	5	6	6
6.5	7	6	6	6	7
5.6	6	5	5	6	6
-5.4	-5	-6	-5	-5	-5
-5.5	-5	-6	-5	-6	-6
-6.5	-6	-7	-6	-6	-7
-5.6	-5	-6	-5	-6	-6

Therefore, the maximum rounding errors of "roundTowardPositive (RTP)", "roundTowardNegative (RTN)", and "roundTowardZero (RTZ)" modes are less than one ulp, and the maximum rounding errors of "roundTiesToEven (RTE)" and "roundTiesToAway (RTA)" modes are less than half ulp.

2.1.4 Flags

The standard also defines five flags to provide diagnostic information for exceptions. The *Invalid Operation* is raised once the result is not usefully definable or any operand of the operation is invalid. In the meantime, an NaN is given as the result. For example, square

root of a number less than zero or $(\infty \div \infty)$. The *Division by Zero* is simply defined as if any number is divided by zero, and an infinity is given as the result with a appropriate sign. The *Overflow* is set if the result is larger than the maximum representable magnitude. In this case, the result is rounded first and the signal is set according to the rounding direction. On the other hand, if the result is less than the minimum representable magnitude, the signal *Underflow* is set. In the given format, the precision is always fixed. However, in some cases (e.g., multiplication, division, and etc), the exact result cannot be completely represented in the given precision. Therefore, the result is rounded to the closest representable number, and the signal *Inexact* is set. The details of the exception handling and flags are introduced in section 7 in [17].

Chapter 3

Fused Multiply-Add

A high performance decimal floating-point fused multiply-add (DFMA) is one of the target in this research. Reviewing the architectures of the existing binary floating-point fused multiply-add designs may be helpful to understand why such a function is useful and to obtain inspirations for designing a DFMA. The basics of the fused multiply-add function is therefore introduced first in section 3.1. Subsequently, several previous typical binary floating-point designs are analyzed and summarized in section 3.2. Finally, the applications of such a function in binary designs are discussed in section 3.3.

3.1 Basics of Fused Multiply-Add

The Fused Multiply-Add (FMA), which is also known as Multiply-Add Fused (MAF) and Multiply-Accumulator (MAC), was first proposed by IBM in the floating-point processor on its RISC System/6000 (RS/6000) in 1990 [26]. The key feature of the floating-point FMA is a merged floating-point addition and multiplication able to minimize the rounding error for the chained binary floating-point multiplications and additions (i.e., $(A \times B) + C$), and also to reduce the hardware area and on-chip busing of binary floating-point processors.

The individual addition and multiplication could be easily implemented on FMA by setting B as one (i.e., $A + C = (A \times 1) + C$) and setting C as zero (i.e., $A \times B = (A \times B) + 0$). Moreover, many computations, such as division, square root, reciprocal, and many transcendental functions, can be iteratively calculated based on the FMA architecture with the Newtons method or other similar methods [27]. Thus, these functions are able to be implemented with very little or even no extra area cost. Because of the benefits on accuracy, latency, and hardware cost, in several high performance commercial processors, only the

FMA architecture is implemented in the floating-point unit [28, 29].

The basic data flow of binary floating-point FMA is shown in Fig. 3.1. The product of the significands of A and B is created by a fixed-point multiplier array. Mean while, the shifting amount to align the product and addend C is calculated in parallel. The following operations are similar to those of a floating-point addition. The product and addend are first swapped and shifted. An addition or subtraction is then performed by considering the effective operation. Subsequently, the number of leading zeros in the result is detected, and the result is shifted in order to make it normalized. Finally, the a rounded result is obtained by adding the possible increment to the normalized result.

Because of the advantages of binary floating-point FMA, such an architecture gained attentions in last 20 years. P. W. Markstein has proved that the accuracy improvement by FMA is the prerequisite to obtain the correctly rounded result of the division and square root in Newton-Raphson approach and the elementary function evaluation [27]. Thus, the characteristic of FMA makes it possible to cut off the hardwired division and square root unit on chip. In [30], R. M. Jessani et al. discussed the effect on area and performance of the floating-point FMA with a reduced multiply array, namely dual-pass. Such an FMA with a halved multiplier reduces 40% of the chip size, and also decreases the performance unfortunately. P. M. Seidel proposed a multiple-path algorithm on FMA by following the basic ideas used in the dual-path adder [31] in which two shifters in alignment and normalization are divided into different data paths. By reallocating the normalizer and rounder, T. Lang and J. D. Bruguera reduced the delay of the FMA in [32, 33], and further the improved architecture with two paths makes the floating-point addition faster than the floating-point multiplication and FMA [34]. E. Quinnell et al. presented a bridge architecture that reuses the components in the existing floating-point adder (FADD) and floating-point multiplier (FMUL) on chip [35, 36]. The single instruction multiple data (SIMD) feature of FMA is also discussed and implemented by L. Huang et al. in [37, 38]. So far many commercial general purpose processors support this instruction in hardware, such as the IBM PowerPC, the HP PA-8000, and the HP/Intel Itanium [39]. Since the FMA function is included in the IEEE standard 754-2008 as a primitive operator, more processors will realize this instruction in the future.

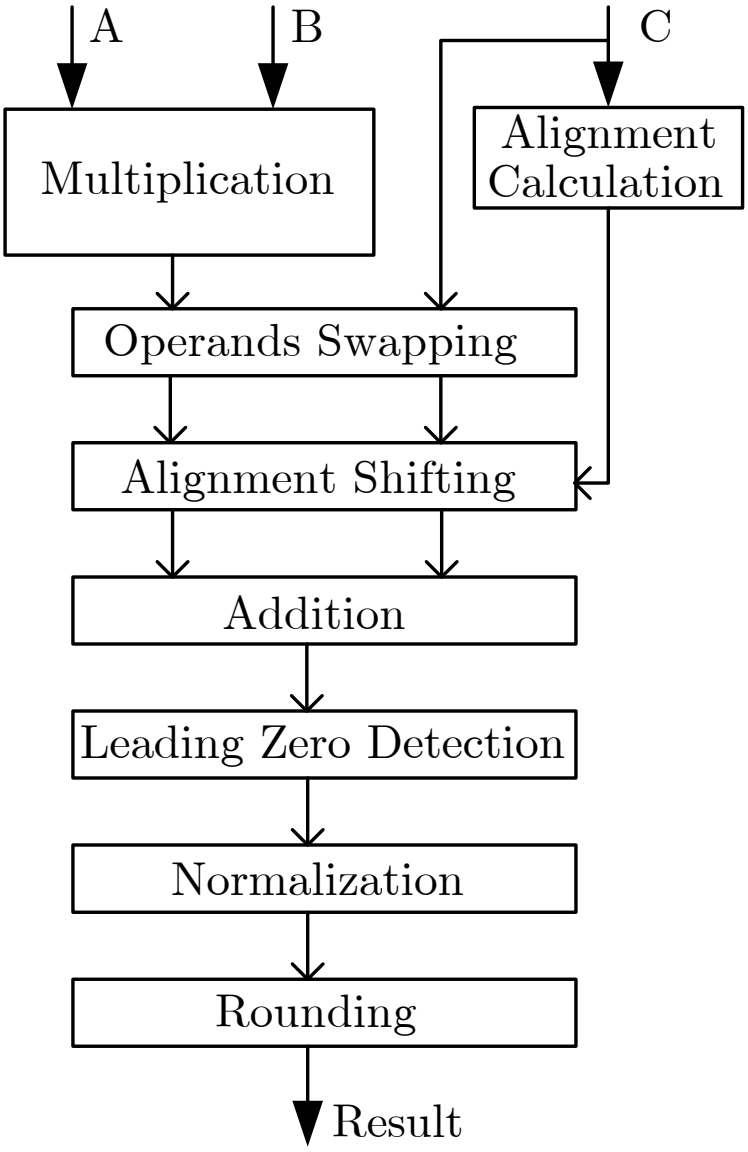


Figure 3.1: Basic FMA architecture

3.2 FMA Designs of Binary Floating-point

After the first FMA, which was introduced in 1990 [26], several novel binary designs are proposed to speed up such a function because of its superiority on floating-point processing. The typical designs are reviewed in this section to summarize the ideas in hardware design.

3.2.1 Original Binary Floating-point FMA Architecture

The first FMA arithmetic unit which was implemented in 1990 [24] is shown in Fig. 3.2. As described before, such an architecture was announced to increase the accuracy due to one less rounding operation. The multiplication inside FMA first produces a double-length product (i.e., $A \times B$). To add the product with doubled width to the third operand C with the single width, the alignment range has to be enlarged which implies a wider data path or a larger delay. This problem is shown in Fig. 3.3. However, such a wider data path which includes shifter, adder, and normalizer keeps all the necessary information to the final rounding operation. Therefore, the accuracy is improved compared to the individual FMUL and FADD. To shorten the total latency, the alignment operation, which shifts the significands of the product and addend to perform the addition on the operands with the same exponent, can be placed in parallel to the multiplication path. Since the delay of the alignment is normally less than that of the multiplier tree, the latency of FMA is reduced by hiding the delay of the alignment shifting. Moreover, the leading zero anticipation (LZA) can be applied to save part of the delay of obtaining the leading zero before normalization. Finally, only one rounding operation is performed for one fused multiplication and addition.

3.2.2 Multiple-path Binary Floating-point FMA Architecture

In 2004, P. M. Seidel proposed a theoretical analysis in [31], in which a multiple-path algorithm similar as that in his dual-path adder was applied. Since the alignment is just to shift two operands (i.e., $(A \times B)$ and C) to get rid of the difference between the exponents, an algorithm regarding the difference between the exponents is discussed. In the algorithm, only part of the data paths is enabled during the calculation, and each sub-path is simpler

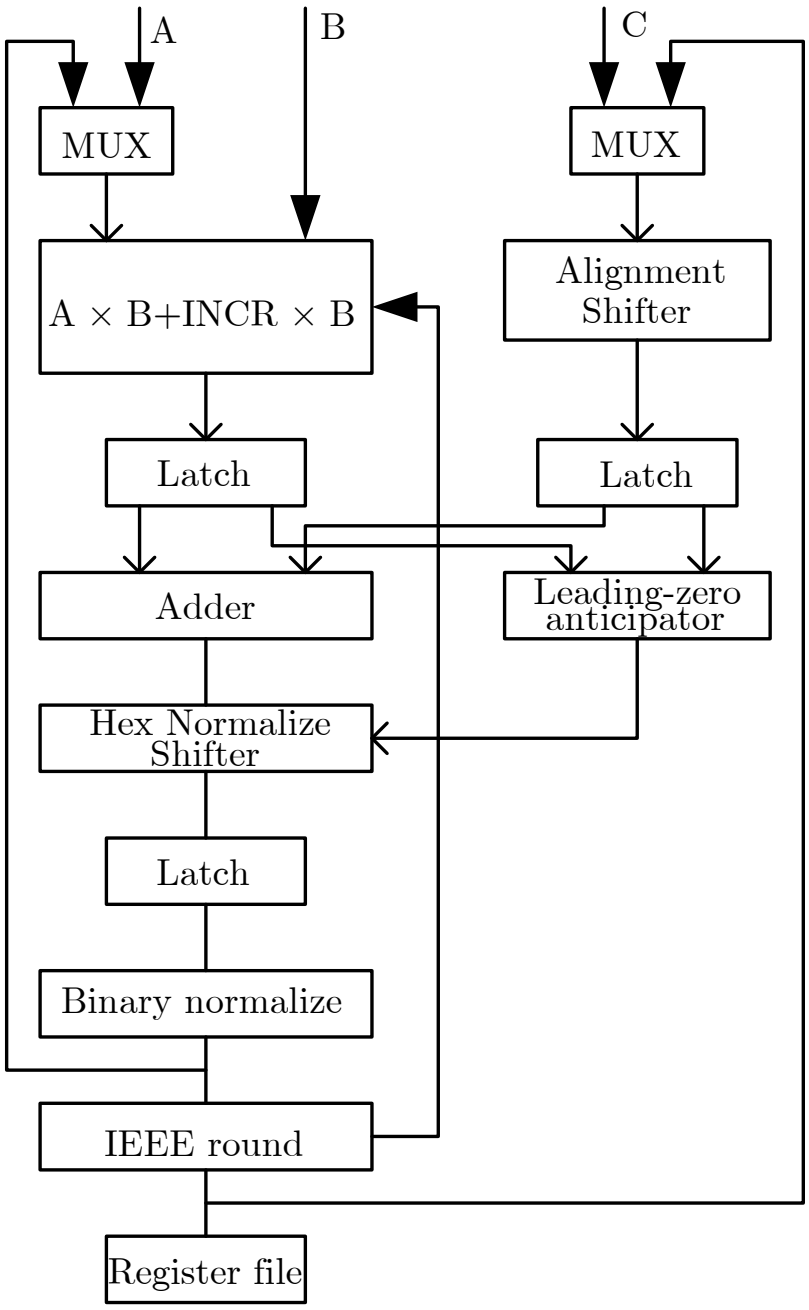


Figure 3.2: Binary FMA architecture of [26]

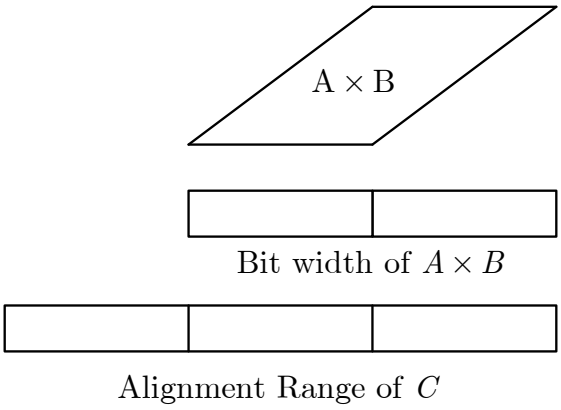


Figure 3.3: Shifting range of alignment

than the combined path to cover all cases. Thus, the delay and power could be reduced with the cost of larger area.

The cases are split based on the difference among the exponents, which is defined by δ (i.e., $\delta = (EA + EB) - EC + bias$, where $EA, EB,$ and EC are the exponents of operands $A, B,$ and C).

1. $\delta \leq -54$, where the exponent of the product is too small to affect the addend as the intermediate result, and the product only can form the sticky bit to generate the increment bit in some rounding modes. In this case, the adder can be disabled.

2. $-53 \leq \delta \leq -3$, where part of the product will be added into the low weight bits of the addend, and the exceeding bits of the product is added/subtracted to all trailing zeros, thus the operation could be simplified to an negation (for subtraction). Only the high 53 bits need to be handled in the adder.

3. $-2 \leq \delta \leq 1$, where the cancelation would generate the leading zero in the result in the subtraction, thus a relative big normalization would be applied after the adder. However, the big shifter in alignment is not needed in this case.

4. $2 \leq \delta \leq 52$, where the low-weight bits of product and the addend need to be added together. But actually only the high 53 bits form the intermediate result, and the exceeding part only affects the rounding result. The addition could be simplified to only determine the increment for rounding.

5. $53 \leq \delta$, where the intermediate result is formed totally by the product, the addend only forms the sticky bit. Thus the adder can be disabled. In [35], the authors proposed a

three-path design in which the Far path is further divided into two paths (i.e., Adder Far path and Product Far Path). This design will not be discussed since it is theoretically similar to the dual-path design proposed in [34, 40].

3.2.3 Binary Floating-point FMA with Reduced Latency

In [32, 33], the authors proposed an improved architecture to combine the addition and the rounding. In such an architecture, the final adder with carry propagation and the rounding unit are parallel by using the compound adder which can simultaneously calculate the sum and $sum + 1$. Thus the correct result can be selected by the increment bit generated by the rounding logic. In the architecture, the final addition is placed at the end of the dataflow. Therefore, the normalization has to be performed before the addition, and the LZA cannot be placed parallel with the adder. Consequently, a new normalization scheme is proposed for shifting two operands which is obtained by adding three operands (i.e., the sum and carry of the multiplier and the addend) with a carry save adder. Moreover, the authors also designed a modified LZA structure to fit the proposed normalization and avoid the increasing on delay. The brief architecture is shown in Fig. 3.4.

In all the designs referenced previously, the FADD and FMUL take the same latency as the FMA unit does. Thus designs enlarge the latency of the individual FADD and FMUL instruction compared to implementing them in individual floating-point adder and floating-point multiplier. In [34], the authors discussed a method to shorten the latency of FADD in a FMA unit by bypassing the multiplier (i.e., a recoder and a CSA tree) when running the addition instruction individually. To do so, the alignment in previous designs is no longer parallel with the multiplier and delayed. Furthermore, to avoid the two shifters in alignment and normalization on critical path, a dual-path design for the addition part separates the shifters in alignment and normalization into two paths. Finally, the compound adder, which calculates the sum and $sum + 1$ simultaneously, makes the rounder parallel with the significand adder as discussed in [32, 33].

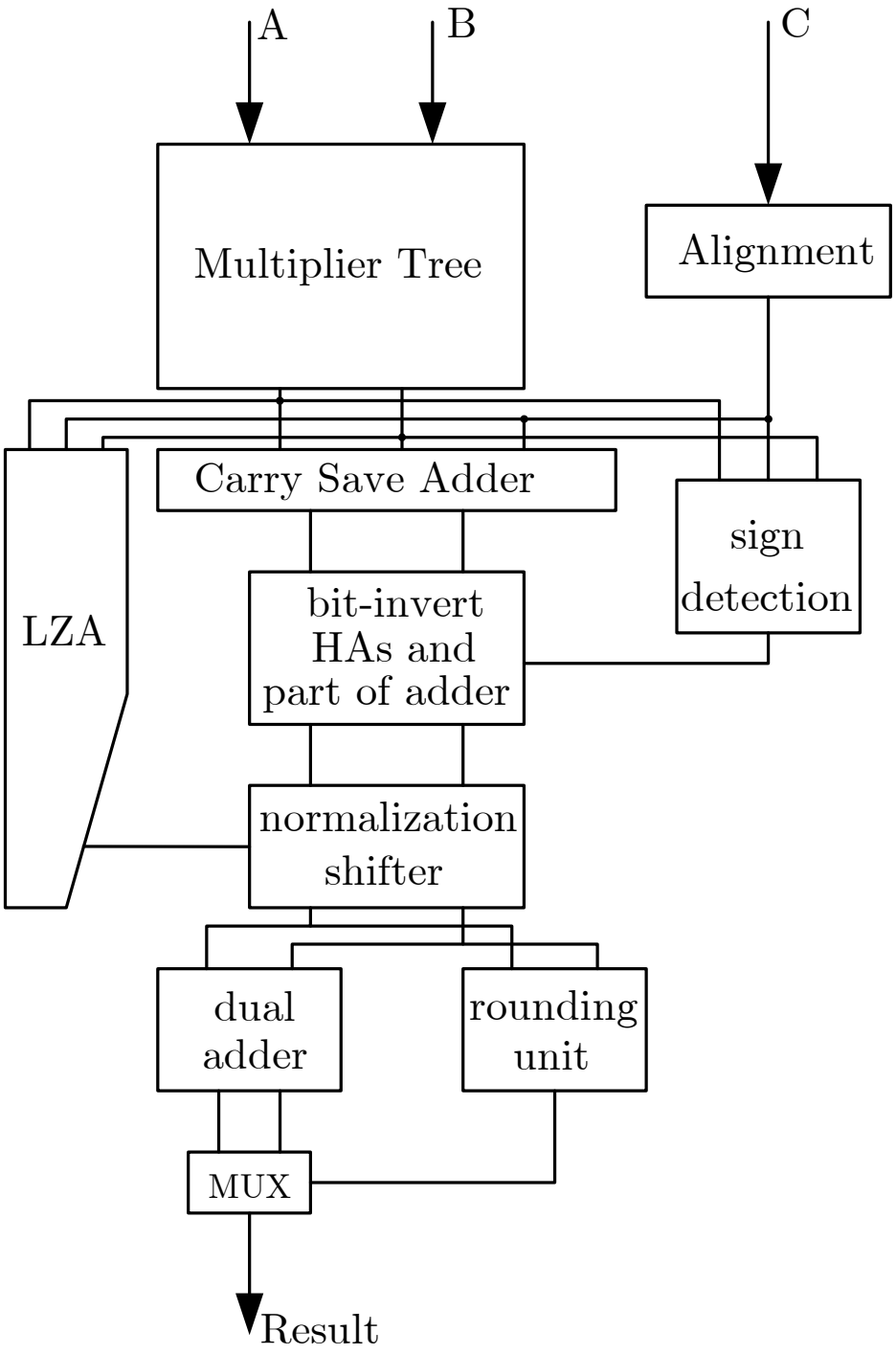


Figure 3.4: Binary FMA architecture of [34]

3.2.4 Combined Decimal and Binary Floating-point FMA

Since some designs of the decimal FADD and decimal FMUL have been proposed and implemented in last several years, the decimal FMA could borrow some ideas from those state-of-the-art decimal designs. P. K. Monsson has proposed a FMA with encodings for combined decimal and binary processing [41]. The author chose a non-speculative multi-operand adder which has been proposed in [42] to reduce the partial product array in the multiplier. The data path is divided into decimal path and binary path in the partial product generation. These two paths share the reduction tree, since the size of it is relatively big compared to the entire architecture. To support combined decimal and binary processing, the data path has to be constructed in an ordinary structure, which means less optimization. The traditional encodings in reduction tree causes an extra delay in the decimal correction unit. Furthermore, the optimization method in [34] cannot be applied to implement an elaborate rounding and addition unit. The brief architecture of the combined decimal and binary FMA is shown in Fig. 3.5. The units for decimal or binary computations are marked by the letter "**D**" or "**B**" on the right bottom.

The synthesis results of the dual-radix design show that the design costs too many transistors compared to the individual designs. The area of the combined FMA is about 282% of that of the FMUL in [22] and 1240% of that of the FADD in [20]. Furthermore, compared to the binary FMA, the area of the combined FMA is about 1267% of that of the binary FMA in [43]. The results violated one of the motivations of the FMA which is to implement FMUL and FADD with a small total area.

3.3 Applications of Binary FMA

In previous sections, only how FMA can improve the performance and accuracy of continuous individual FADDs and FMULs are introduced. Actually, other functions (e.g., floating-point division (FDIV), floating-point square root (FSQRT), and etc.) can be implemented based on the FMA unit. This is also one of the benefits of the RISC processor in which only key arithmetic units are implemented in hardware and other functions are implemented in

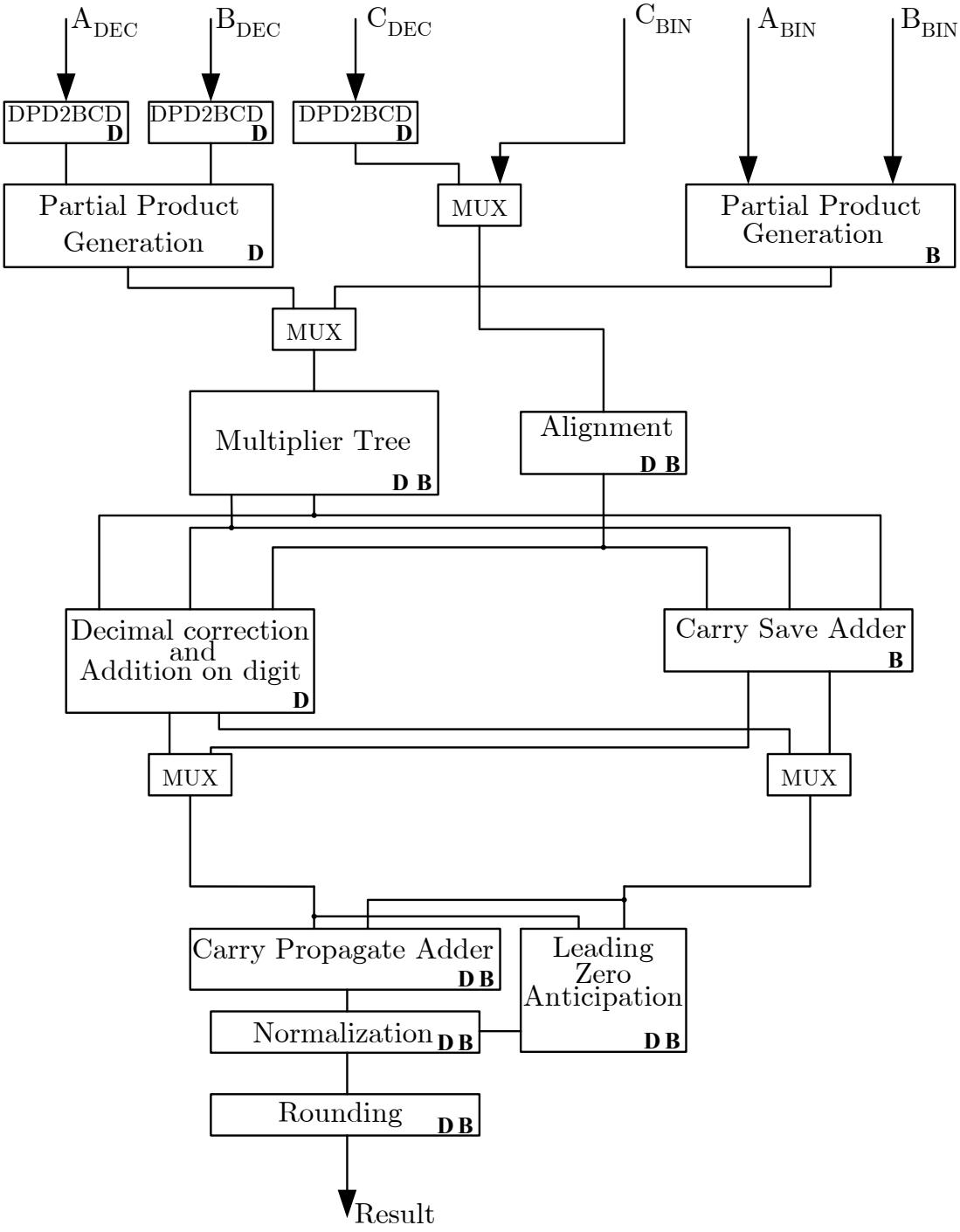


Figure 3.5: Combined Decimal and Binary FMA architecture

software with the support of the library and the assisting hardware. Additionally, several papers have demonstrated that the binary FMA is the requisite component to get correct results of some functions.

P. W. Markstein in [27] introduced the Newton-Raphson's (NR's) method to perform the division and square root with FMA instruction and analyzed the correctness of such a method. Previously, without the FMA instruction, using the NR approach for division required a special corrective action at the end of iterations to get the last bit rounded correctly. If the corrective action is not applied, the result can be rounded incorrectly [44]. Additionally, in the computation of the elementary functions, the FMA instruction is also very useful to improve the accuracy of the argument reduction which is important for the accuracy of the elementary function evaluation.

F. G. Gustavson et al. in [45] proposed an algorithm in order to correctly calculate the four basic operations (i.e., ADD, SUB, MUL, and DIV) with good performance. Because of the conversion error (representation error) which is the difference between the binary approximated number and decimal floating-point number, the error will be propagated during the calculation. The authors proposed a method to process the result in two parts (i.e., extended precision), and showed that the FMA instruction is the key to perform exact floating-point multiplication and division.

To obtain a higher frequency, more stages of pipeline can be applied. However, such a feature increases the latency of the operation. In NR's algorithm, the refining guesses of quotient and the reciprocal of divisor in one iteration can be interleaved inside the multiply-add architecture, and the different pipeline stages of the multiplier can be used in every iteration. But since the multiplications between two iterations can not be run independently, the utilization ratio of the multiplier decreases as the pipeline increases.

R. C. Agarwal et al. proposed a method based on power series approximation on the IBM POWER3 [46]. In such a method, the authors used a table to first create an approximation of the reciprocal of divisor, and applied the power series of the error to refine the approximated reciprocal of the divisor. Since the error can be calculated, and the initial approximation is obtained depending on the range of the divisor, the refining formula can be rearranged to a series of multiply-add operations on the known variables. Hence, the dependence between

the multiply-add iterations is decreased. The results also showed that the performance is much better in a longer pipelined architecture.

The key benefits obtained from such high-level functions implemented based on FMA are less area and simple hardware design that also means the faster speed. This is also why FMA alone is included in the floating-point unit of RISC processors. The algorithm for implementing high-level functions in software with the assistance of the FMA instruction may also be implemented in hardware. Such a strategy balances the complexities of the hardware design (i.e., area consumption) and the compiler design. The computing latency and memory access may be also minimized. Moreover, the exclusive usage of the dedicated unit can be solved somehow by multi-core and multi-issue architectures, which are the popular trend of RISC processors.

Chapter 4

Number Systems

For designing a high performance DFMA architecture, unconventional number systems are considered as a encouraging technique in order to improve the performance and area efficiency. Therefore, in this chapter, the basics of number system are introduced as a preliminary study before going to the decimal designs. The binary, decimal, and redundant number systems are briefly introduced in sections 4.1, 4.2, and 4.3 respectively.

4.1 Binary Number System

Number System is comprised of the methods to represent numbers and the rules to perform arithmetic operations on the numbers. Nowadays, the central processing units inside computers are abundantly built by bistable devices, which have two stable states (e.g., high level voltage and low level voltage). The binary numbers and arithmetics are therefore widely used in today's computer systems.

In the binary number system, which has 2 as the radix, two elements (i.e., "0" and "1") are used to represent numbers. For example, the decimal number 5 can be represented in binary number as 101 or 0101.

$$(101)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 1 = (5)_{10} \tag{4.1}$$

In a conventional number system, the radix always implies the elements can be used to represent numbers in such a system. Since every numbers which are larger than or equal to the radix number generate carries to the higher position, the available elements of a conventional number system with a given radix are $[0, r-1]$, where r is the radix. The set of the elements is called *digit set*. In binary system, the digit set is therefore [0,1]. Note that

the unit of the element can be special in different number systems. For example, in binary system, "bit" is applied, and in decimal system, "digit" is usually used. So in equation (4.1), we say "a 1-digit decimal number, 5, is represented in a 3-bit binary number as 101".

Another very important attribute of a number system is *encoding*, which means how the numbers in a given number system are represented with the basic elements. For example, how is the negative decimal number "-8" represented with binary bits? To answer this question, let's first look at the basic methods to represent signed numbers that can be positive or negative.

Signed magnitude representation employs one sign bit, which is usually the most significant bit, to indicate the sign of a number (e.g., "0" means a positive number, and "1" means a negative number). The rest of the bits are therefore used for the magnitude or the absolute value of a number. For example, the decimal number "-8" can be represented as "11000".

Complement representation in binary system includes one's complement and two's complement. The positive number in one's complement has a "0" at the most significant bit, and the rest of the bits are exactly the same as those of the unsigned representation. The negative number in one's complement is represented by inverting each bit of the corresponding positive number. For example, the decimal number "8" in one's complement is "01000", and "-8" in one's complement is "10111". Alternatively, the negation of a number in one's complement can be done by subtracting the number from $2^n - 1$, where n is the bit width of the number. For example, "-8" can be negated from "8" in one's complement by:

$$2^5 - 1 - \text{“01000”} = \text{“11111”} - \text{“01000”} = \text{“10111”} \tag{4.2}$$

On the other hand, two's complement represents negative number with an extra 1 at the least significant bit on the basis of the one's complement. For example, the negation of "8" is done by inverting "01000" to "10111", and adding the extra "1" to obtain "11000". Alternatively, the negation of a number in two's complement can be done by subtracting the number from 2^n, where n is the bit width of the number. For example, "-8" can be negated from "8" in two's complement by:

$$2^5 - \text{“01000”} = \text{“100000”} - \text{“01000”} = \text{“11000”} \tag{4.3}$$

Table 4.1: The signed numbers in different representations

Binary representation	Signed Magnitude	One's complement	Two's complement
"000"	+0	+0	0
"001"	+1	+1	+1
"010"	+2	+2	+2
"011"	+3	+3	+3
"100"	-0	-3	-4
"101"	-1	-2	-3
"110"	-2	-1	-2
"111"	-3	-0	-1

In Table 4.1, the signed numbers can be represented with 3 bits signed magnitude, one's complement, and two's complement are listed. In the first two representations, two '0's (i.e., positive and negative zeros) exist. Thus one representation symbol is wasted and the computation on these numbers is more complicated. However, the symmetry of these numbers may simplify the negation. On the other hand, only one '0' exists in two's complement representation. Additionally, the addition or subtraction on the numbers in two's complement is straightforward. But the representable numbers are no longer symmetrical around zero. The further knowledge about the binary number system can be found at [47, 48].

4.2 Decimal Number System

Since the decimal numbers are used by human, the binary coded decimal (BCD) numbers are used to represent the decimal numbers in radix 10 system or decimal number system. Actually, binary number system is very efficient on decimal integer calculation. However, the conversion between decimal and binary numbers has to be performed at the first beginning and the last steps. Furthermore, the binary integer is inefficient on decimal shifting and rounding. For example, $(80)_{10}$ right shifted by 1 digit is $(08)_{10}$.

In the conventional decimal number system, the digit set is [0,9] implied by the radix 10. To represent these ten digits, many encodings have been investigated. The BCD numbers

apply binary bits to represent a decimal digit as mentioned by its name. The most common one is BCD-8421 system, where "8421" means the weight on each binary bit. For example, the decimal number "9" can be represented by 4 binary bits as:

$$(1001)_2 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (9)_{10} \tag{4.4}$$

However, the weights on 4 binary bits do not have to be "8421". Some unconventional BCD encodings are available. For example, in BCD-4221, the decimal number "5" can be represented as "1001" or "0111". Another useful encoding is called Excess-3 BCD, which adds 3 to each number in BCD-8421 encoding. For example, the decimal number "0" is represented as "0011", and "9" is represented as "1100".

Table 4.2: The decimal digits in different representations

Binary representation	BCD-8421	BCD-4221	Excess-3 BCD
"0"	"0000"	"0000"	"0011"
"1"	"0001"	"0001"	"0100"
"2"	"0010"	"0010" or "0100"	"0101"
"3"	"0011"	"0011" or "0101"	"0110"
"4"	"0100"	"1000" or "0110"	"0111"
"5"	"0101"	"1001" or "0111"	"1000"
"6"	"0110"	"1010" or "1100"	"1001"
"7"	"0111"	"1011" or "1101"	"1010"
"8"	"1000"	"1110"	"1011"
"9"	"1001"	"1111"	"1100"

In Table 4.2, the decimal digits from 0 to 9 are represented in three different encodings. Note that only ten representation symbols are used in the BCD-8421 system. The unused symbols cause two disadvantages. First, 37.5% (i.e., 6/16) of representation space is wasted which means a lower encoding efficiency. Second, the carry and the sum may be generated incorrectly in the addition on these numbers. The example below shows what is the problem.

$$(3)_{10} + (9)_{10} = (0011)_{BCD8421} + (1001)_{BCD8421} = (1100)_{BCD8421} \tag{4.5}$$

In equation (4.5), the encoding "1100" is not used in BCD-8421 system. The correct result (e.g., 11) can be obtained by adding 6 on the intermediate result if it is over the representable range. For example,

$$(1100)_{BCD8421} + (0110)_{BCD8421} = (0001\ 0010)_{BCD8421} = (12)_{10} \tag{4.6}$$

The first disadvantage is solved by the unconventional BCD encodings, such as BCD-4221 mentioned above or BCD-5211. The second disadvantage can be partially solved by the excess-3 BCD encodings. Since each number in excess-3 is added by 3 to each corresponding number in BCD-8421, the addition on two excess-3 operands is added by 6. Therefore the carry is always correct in excess-3 addition. However, the correction on sum is still necessary. A notable advantage of the excess-3 encoding is that the nine's complement is simply inverting each bit as the one's complement in binary system.

4.3 Redundant Number System

In previous two sections, the radix, digit-set, and encoding of the conventional number systems are introduced. If given a radix, the number of elements in the digit set is larger than the radix, then the number system is redundant. Two examples are first provided to show the redundant number system. In the first example in Fig. 4.1, suppose that 4 binary bits are used to represent a decimal digit. But the decimal digit set is extended to [0,11]. If two operands addition is performed, the digit set of the result is therefore in [0,22]. Consequently, 2 is the largest carry in this number system (i.e., carry digit is in [0,2]). After extracting the carry, the intermediate sum is [0,9]. Finally, the result of the addition, which is the sum of the digit sets of carry and intermediate sum, is in [0,11]. With the given number system, the carry only propagates to the higher one digit. Thus, the long term carry propagation which causes a big timing delay is eliminated.

However, the digit set of the result does not have to be the same as that of input operands. In the second example shown in Fig. 4.2, the digit set of operands is extended by one digit.

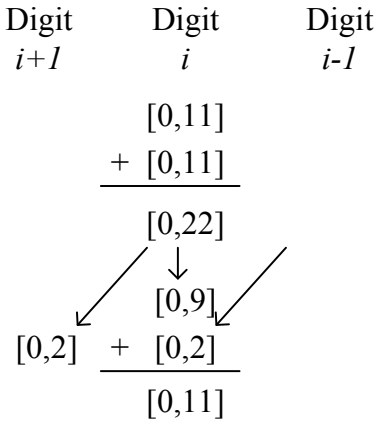


Figure 4.1: Example of calculation with redundant number system

After calculation, the digit set of the result is less than that of operands. If the digit set of the result is not larger than that of the operands, the continuous additions without long term carry propagation can be performed.

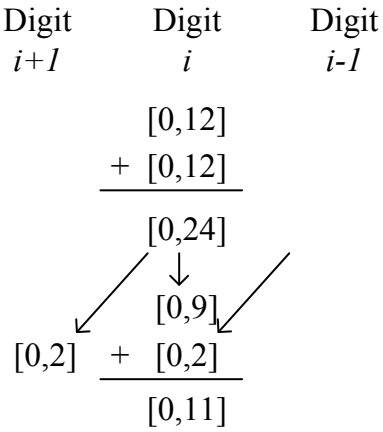


Figure 4.2: Example of calculation with redundant number system: reduced digit set in output

In binary number system, carry save addition is widely used in many applications. In binary carry save addition, three binary bits are added together, and 1 bit sum and 1 bit carry are obtained. If a digit set (e.g., [0,2]) and the corresponding encoding with 2 binary bits (e.g., 0="00", 1="01" or "10", and 2="11") are defined, the binary carry save addition can be considered as an addition on incomplete redundant operands.

The two's complement or ten's complement numbers are efficient to perform subtractions. However, in redundant number system, the subtraction can be performed easily with

symmetrical digit sets (i.e., include negative digits). Suppose a redundant decimal number system with a symmetrical digit set [-6,6]. The carry propagation of the addition is shown in Fig. 4.3. Note that if subtraction is performed on this digit set, the carry propagation is the same as that shown in Fig. 4.3. This advantage reduces the complexity of subtraction on the redundant number system by eliminating the complement operation. This kind of numbers is called *signed digit numbers*. However, the conversions from traditional digit set (redundant digit set) to redundant digit set (traditional digit set) might be needed at the first and last operations. More information about redundant number system can be found in [49].

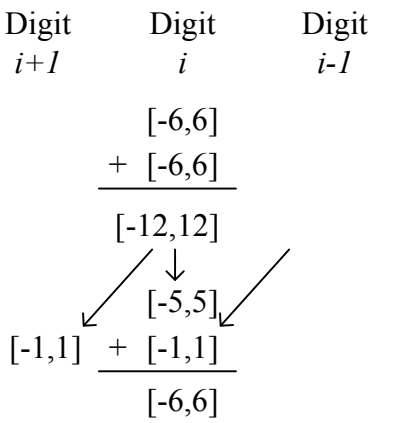


Figure 4.3: Example of calculation with redundant number system: signed digit set

When the specification is determined, what need to be considered in a number system and what are these considerations about are illustrated in Fig. 4.4. In previous sections, digit set and the corresponding encoding are discussed. Actually, an implicity factor, which is related to digit set and encoding, is not introduced. Once a digit set is applied to represent numbers or an encoding is applied to represent the digit set, how many binary bits are used is directly related to the size of the memory and the width of the data path that means the representation efficiency. Furthermore, how to perform the basic operations (e.g., addition and subtraction) on a given digit set and encoding is directly related to the complexity of the hardware design. For example, a binary full adder, which includes two XOR gates and one carry circuit, is necessary for 1 bit carry save addition. In the scope of digit set, the best contribution is eliminating the carry propagation. However, encoding is very important to determine the arithmetic rules and to further determine the complexity of the hardware

design. These two problems will be examined in later chapters.

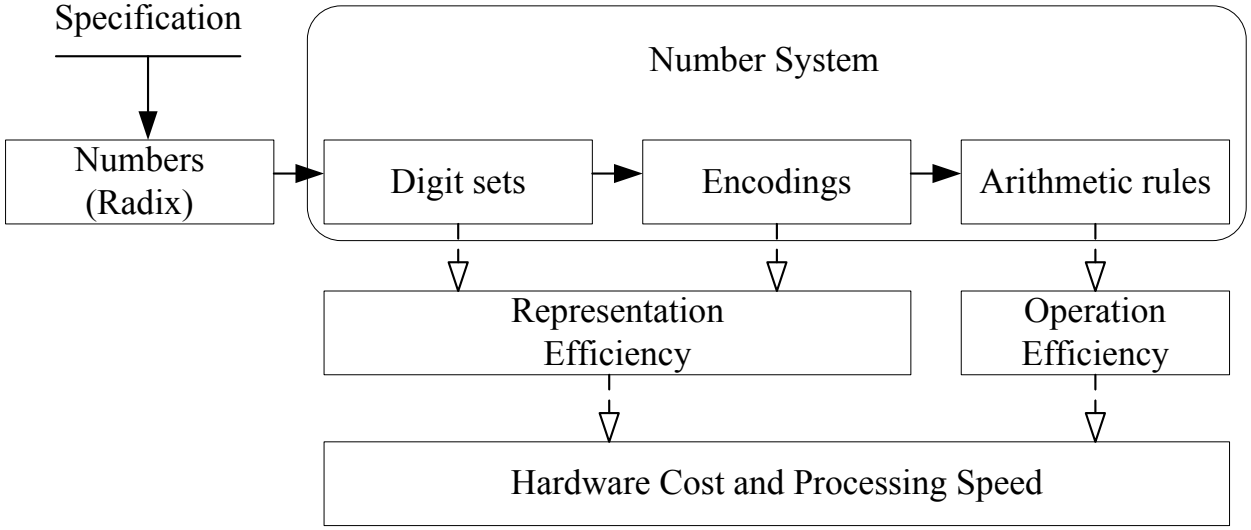


Figure 4.4: Consideration of the number system

Part III

Designs

Chapter 5

Previous Designs

Prior to presenting the proposed decimal designs, the existing decimal fixed-point and floating-point designs related to ours are introduced as a literature review. The contents in this chapter describe the leading edge of the decimal designs which are the basis and also reference for comparison of our research. Since the entire research is divided into three parts (i.e., addition, multiplication, and fused multiply-add), the following three sections describe decimal fixed-point addition, multiplication, and floating-point FMA respectively.

5.1 Decimal Fixed-point Addition

In the decimal floating point arithmetic, the operation on the significand encoded in densely packed decimal (DPD) format, which express 1000 decimal numbers in 10 binary bits instead of 12 binary bits in BCD encoding, could be implemented in the same technique as in the decimal fixed point arithmetic. Among the fixed point decimal arithmetics, addition is the most important one since it is the basis of all other operations. For instance, in the sequential multiplication and digit recurrence division, the partial product for each iteration are accumulated by the adder. Moreover, in the parallel multiplication and functional division, partial products are reduced by the adder tree. In [58], the authors show that the logics related to the significand computation units occupy 51% timing delay and 41% area of the decimal floating point adder. Since an improvement in addition can benefit to many other decimal operations, many methods and algorithms were applied to boost the performance of the decimal adder.

Traditionally, a decimal digit z_i, where $z_i \in \{0, 1, \ldots, 8, 9\}$, is represented in the 4-bit binary coded decimal (BCD) encoding. Alternatively, the signed digit set $\{-\alpha, -(\alpha -$

$1), \ldots, \alpha - 1, \alpha\}$ is applied to represent the decimal numbers. If $2\alpha + 1 > r$, where r is the radix, the number system is redundant. For a redundant number system, one number has more than one representation which allows the carry free addition. The conventional SD carry free addition/subtraction algorithm is given as follows [49]:

Algorithm 5.1: One Digit Conventional Carry Free Addition

Data: SD operands X_i, Y_i, transfer digit T_i and operation *op*.

Result: SD result S_i and transfer digit T_{i+1}.

1. Compute $P_i = X_i$ *op* Y_i
2. Divide P_i into T_{i+1} and $W_i = P_i - r \times T_{i+1}$
3. Compute $S_i = W_i + T_i$

where P_i is the position sum, T_{i+1} is the transfer digit, W_i is the temporary sum and r is the radix.

The signed digit (SD) number system could be applied to eliminate the carry chain in the carry free addition. After the first published decimal signed digit adder designed by A. Svoboda in 1969 [50], some papers were presented in the last decade.

B. Shirazi et al. in [59] proposed a redundant binary coded coded decimal (RBCD) adder in digit set $[-7, 7]$. Since BCD encoding only applies 4 binary bits to present from 0 to 9 as an unsigned number, six encoding symbols are wasted. In the RBCD, the authors treat the 4 binary bits as a signed number. Therefore [-8,7] can be represented within the 4 binary bits. However, to perform the subtraction on a symmetrical digit set, "-8" is not used in RBCD addition. To perform the addition, a binary full adder with carry propagation in 4 bits is applied first to get the intermediate sum (i.e., $a + b$) of the two operands. Afterwards, a combinational logic unit is used to detect if the intermediate sum is out of range (i.e., over [-6,6]) and to decide the transfer digit or carry to the next digit. Since the intermediate sum has to be corrected by considering the carry out to the next digit and the carry in to the current digit, another combinational logic unit is performed to create a correction signal

based on carry in and carry out. Finally, the final redundant result in [-7,7] is obtained by adding the correction signal to the intermediate sum in a 4 bits binary carry propagation adder. All these operands mentioned above are in serial. Note that, the conversions have to be performed before and after the RBCD, since the BCD numbers "8" and "9" are not in the range of the RBCD encoding.

H. Nikmehr et al. provided the decimal signed digit (DSD) adders in digit set $[-9, 9]$ in [51]. In this DSD adder, a speculative method which creates all possible results and selects the correct one based on the carry in and carry out signals is applied. For instance, in [51], the intermediate sum p is firstly created by adding the operands a and b. Subsequently, $p + 9$, $p + 10$, $p - 1$, $p - 10$, and $p - 11$ are calculated at the same time. A combinational logic is then performed to decide the transfer digit to next digit. Finally, the correct result is selected from the pre-calculated result by the carry signals. Note that, to represent a decimal signed digit, two 4-bit binary numbers are used. The extra bits to represent numbers and the pre-calculated result both mean less area efficiency.

J. Moskal et al. also provided a non-speculative method to perform decimal signed digit addition in digit set $[-9, 9]$ in [56]. The digit set and number representation are similar to the design in [51]. However, since non-speculative method is used, the final result is not selected from pre-calculated result but corrected from the intermediate result. Therefore, the two operands are firstly compressed (added) to obtain the intermediate. After that, the correction signal is created by considering the sign signal which is obtained by a bit-wise carry propagation network. Finally, a signed digit binary adder array is applied to correct the intermediate sum. However, due to the complicated representation, some combinational logics (i.e., multiplexor, inverter, reduction logic) have to be used inside the adder.

In [52], A. Kaivani provided a fully redundant decimal addition based on stored unibit transfer (SUT) encoding in $[-8, 9]$. In the SUT encoding, an extra binary bit so called unibit is applied to represent "-1" and "1". Thus, together with a 4-bit signed binary number, the digit set is enlarged to [-8,9]. Note that, the 4-bit signed binary has negative weight on each bit (i.e., "8 -4 -2 -1") compared to a traditional signed binary number. Therefore without the unibit, the signed binary is in [-7,8]. To perform the addition, the unibits of two operands are firstly extended to a 4-bit signed binary number. A binary carry save adder is therefore

applied to add up the three 4-bit operands. Subsequently, to compress the result, a 3-bit binary carry propagation has to be used after the carry save adder. Finally, the result is divided into two parts to be corrected at the last stage. The unibit is therefore "stored" in the result.

There are two branches of the architectures as mentioned above. The speculative architecture shows disadvantage on the hardware area efficiency. Additionally, if the encoding is complicated, the timing delay to compute all the necessary pre-calculated result could be large. On the other hand, the non-speculative architecture generally has a smaller hardware area. However, the selection of the encoding is also very important in this architecture.

Let's review the conventional signed digit addition/subtraction algorithm. Since the transfer digit T_{i+1} to the next stage is independent on T_i from the last stage, the carry chain is eliminated, and the delay is no longer related to the digit width of the input. However, this algorithm still could be improved in following two aspects:

Deciding T_{i+1} without P_i

Once P_i is obtained, the transfer digit T_{i+1} is generated according to the range of the position sum. Hence, a carry chain for calculating P_i limits the performance of the adder. In this thesis, an algorithm to decide the transfer digit directly on the range of the operands, X_i and Y_i, is introduced. The method for range division is discussed in section 6.1.

Calculating S_i without W_i

After generating the T_{i+1}, a compensation in decimal addition (i.e., ± 10) is applied to correctly calculate the temporary sum, W_i. Further, the transfer digit T_i from the last stage is added to obtain the final result S_i. In this process, these two continuous additions imply a complicated and slower design. A method to merge them into one add operation which leads to a better performance on area and delay is proposed in section 6.1.

5.2 Decimal Fixed-point Multiplication

Multiplication is one of the four basic arithmetic operations. An analysis of benchmarks shows that the percentage of execution time of decimal multiplication could reach over 27% in some applications [18]. Due to the importance of multiplication, some decimal fixed-point designs are proposed in [42, 63–70]. Furthermore, decimal floating-point multipliers based on those fixed-point designs are published in [22, 73, 74, 93, 94].

5.2.1 Analysis of Previous Parallel Designs

In [65], to avoid complicated multiples of X, the operand Y is recoded into two parts, $Y_i = Y_{Hi} + Y_{Li}$, where $Y_H \in \{0, 5, 10\}$ and $Y_L \in \{-2, -1, 0, 1, 2\}$. Therefore, only the $-2X, -X, 2X$, and $5X$ need to be implemented in logic gates. Since the multiples are represented in 10's complement format, the negation is implemented by a 9's complement recoder, and the incremental one is only applied on the least significant digit (LSD). Furthermore, to generate the partial products from $1X$ to $9X$ in BCD carry save (BCD-CS) format, a decimal CSA has to be applied. The parallel PPR for $2n$ partial products (i.e., n sums and n carries) is implemented by 6 levels of BCD full adders (BCD-FA) for a 16×16-digit multiplication. Half of the decimal carries of partial products are added separately by carry-counters. Two outputs of PPR, $2n$-digit sum and $2n$-bit carry, are added together by a prefix network with a conditional adder. Furthermore, an improved PPR algorithm based on a multi-operand decimal addition in [71] is provided by L. Dadda in [66]. The partial products in columns are firstly added in a binary form with the binary carry save adder. Subsequently, a binary to decimal conversion algorithm is applied to convert the binary result to decimal encoding.

In [67], G. Jaberipur et al. propose a new PPG algorithm which only generates $2X$ and $5X$ to compose other multiples from $1X$ to $7X$. The $8X$ and $9X$ are divided into two parts in which the $8X$ is implemented by $E = 10E_h + E_l$, and the $9X$ is implemented in the same way as $N = 10N_h + N_l$. Therefore, the algorithm avoids not only the negation logic for $-2X$ and $-X$, but also the $4X$ (double times $2X$) to generate $8X$ and $9X$. Furthermore, by analyzing the range of the computation and gate level representation, the BCD-FA in the PPR unit is simplified. The two outputs of the PPR unit are further reduced to one $2n$ digits

and one $(2n-l)$-digit BCD numbers, where l is the number of levels of the BCD-FA in the PPR structure. Hence, for the final product computation, only $2n-l$ digits are involved in the carry propagation adder to generate the final multiplication result.

In [70], the authors propose a redundant decimal addition algorithm based on a specific encoding, namely weighted bit-set encoding. With such an addition algorithm, a multiplier based on the redundant number system is provided. The double-BCD format multiples are firstly created by combining the easy decimal multiples (i.e., $2X$, $4X$, and $5X$). In the PPR unit, two-operand redundant adder is applied to reduce $2n$ BCD partial products to a redundant number in the range of $[0, 15]$, so called overloaded decimal digit set (ODDS). Furthermore, in the last step, the redundant product is converted to the BCD encoding by a digit set converter with a propagation process.

In [68], A. Vázquez et al. propose an improved design of their previous work published in [76]. In the improved new family parallel decimal multiplier, two unconventional decimal encodings (i.e., BCD-4221 and BCD-5211) and two architectures (i.e., radix-10 and radix-5) are applied to generate and reduce the partial product. In radix-10 architecture, the operand Y is recoded into SD digit-set $[-5, 5]$, and $n+1$ partial products are selected by the recoded Y. Alternatively, in radix-5 architecture, the second operand is encoded into two parts, $Y_i = Y_i^U \cdot 5 + Y_i^L$, where $Y_i^U \in \{0, 1, 2\}$ and $Y_i^L \in \{-2, -1, 0, 1, 2\}$. Therefore, in this scheme, there are $2n$ partial products need to be reduced. In the PPR unit, only the binary full adders and combinational recoders are applied due to the specific encodings. Finally two $2n$-digit results are added together with a quaternary tree (Q-T) adder based on the conditional speculative decimal addition proposed by the same authors in [75].

In [74], another variant of the design proposed in [76] (i.e., the radix-10 architecture) is introduced. The author applied the idea and basic architecture of Vázquez's radix-10 design to create a decimal floating-point multiplier. The only difference is that the final product accumulation is replaced by a decimal adder with a Kogge-Stone carry network.

5.2.2 Analysis of Previous Sequential Designs

The sequential multipliers generate partial products gradually (i.e., one per iteration) and accumulate them sequentially. This architecture, although not very fast, is popularly used

whenever cost efficiency is the main intention. All the sequential multipliers consist of two main steps 1) partial product generation (PPG) and 2) partial product accumulation (PPA).

The most popular algorithm for decimal PPG is based on the generation of easy-multiples of the multiplicand X i.e., the multiples which can be generated as a non-redundant decimal number via a carry-free approach (e.g., $X, 2X, 4X, 5X$). The concept of generation of these multiplies is very similar to that in the parallel multipliers.

In order to achieve the final product one needs to generate and sum up the partial products for all digits of the multiplier. This step, also known as partial product accumulation, usually consists of a carry-propagating or carry-free decimal adder which is much simpler than the partial product reduction array in parallel multiplier. However, as mentioned above, the performance (i.e., latency and throughput) is limited due to the sequential processing strategy. It should be noted that a final conversion to non-redundant representation is required, in case of using a carry-free decimal adder.

For example, Erle et al. propose a traditional method of decimal multiplication in [63]. The design borrows the idea from binary multiplication which reduces the partial products in a carry save adder (CSA) based structure. Furthermore, to reduce the complexity of the multiples generation, a so-called secondary set which contains $\{X, 2X, 3X, 4X, 8X\}$ is applied, and all the missing multiples could be generated based on the elements in the secondary set with no more than one carry save addition. The decimal 3:2 CSA and 4:2 compressor are described in [63]. Furthermore, the partial product for each iteration could be added iteratively within the delay of a decimal 4:2 compressor. An $n \times n$-digit multiplication can be finished in $n + 4$ cycles.

Another sequential decimal multiplier with easy multiples (i.e., $\{X, 2X, 4X, 5X\}$) is proposed in [42]. Additionally, a 2-stage overloaded decimal adder which can sum two partial products and one iteration result with less delay than a decimal 4:2 compressor is presented. By doing so, a clean-up block has to be applied to finally correct the decimal encoding before the carry propagated addition in the final step. Thus, in such a multiplier, the latency of one operation is up to $n + 8$ cycles.

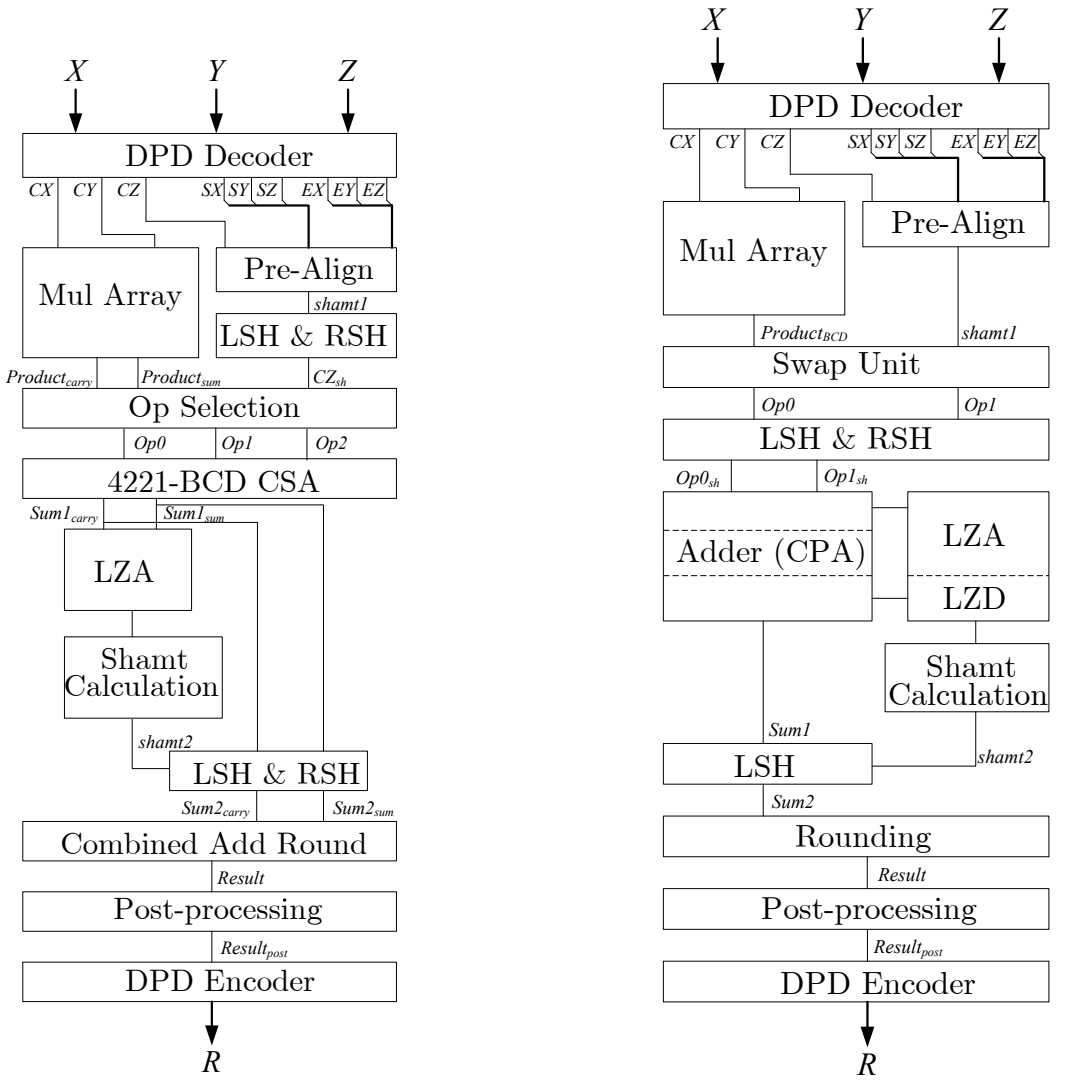
An alternative sequential redundant multiplication is described in [64]. The authors present an algorithm which recodes both operands into the SD digit-set $[-5, 5]$ to generate the

SD operands with simple logic. Further, a digit multiplier block on the range of $[2,5] \times [2,5]$ is proposed to generate the partial products in SD format. Hence, a Svoboda's signed digit adder with a restricted range is consequently applied to add signed digit partial products iteratively. The SD sequential multiplier takes $n+4$ cycles to finish one multiplication.

5.3 Decimal Floating-point FMA

To the best of our knowledge, four pure decimal floating-point fused multiply-add (DFMA) designs have been announced previously. In [103], the author described a top level design of the DFMA which is based on a previous binary FMA architecture with decimal sub-modules [33]. In [93], the authors provided a conventional DFMA comprising a parallel decimal multiplier proposed by the same research group in 2008 [94], a decimal carry propagation adder (CPA) to accumulate the intermediate results, and a decimal rounder which creates the final significand after the decimal FMA core. In [95], the authors proposed a new leading zero anticipation (LZA) algorithm which starts the detection and decision of the post-alignment shift amount in parallel with the decimal adder. In [102], the author introduced a new FMA architecture which combines the addition and the rounding operations into a single unit. Since there are no descriptions about the detailed structures in the first two designs, only the latter two designs are referred to be introduced in this section.

Two previous architectures of the DFMA operation proposed in [102] and [95] are shown in Figures 5.1(a-b). In Fig. 5.1(a), the alignment shifting is performed in parallel with the multiplier array. Therefore, the pre-alignment is excluded from the critical path. To figure out the rounding position before the final addition is performed, in the following selection and adder modules, the proper range of the operands are selected and a 4221-BCD decimal carry save adder is applied to add one product in carry-save format and one addend. A leading zero anticipator and shift controller are therefore created to decide the possible exponent and rounding position. The final result is then obtained by a combined adder and rounder. On the other hand, in Fig. 5.1(b), a new leading zero anticipation algorithm is introduced. In this architecture, a swapping module and a shifting module are placed after the multiplier array. Afterwards, the carry propagation adder and the leading zero anticipation unit are



(a) Architecture proposed in [102] **(b)** Architecture proposed in [95]

Figure 5.1: Decimal floating-point fused multiply-add architectures

applied in parallel. The following post-alignment shifter creates and transmits a $2n$-digit intermediate result to the final rounder.

Chapter 6

Proposed Designs

This chapter describes the details of the proposed designs. The fixed-point addition and multiplication with redundant internal encodings are firstly designed to reduce the latency of the major components in a DFMA. Additionally, the architecture and algorithms of the proposed DFMA are beneficial from not only the new addition and multiplication but also the specific number system. The detailed descriptions of the addition, multiplication, and DFMA are provided in sections 6.1, 6.3, and 6.4.

6.1 Decimal Fixed-point Addition

The addition is the basis of all the other arithmetics. Therefore, a fixed-point carry free addition is firstly studied and investigated to increase the performance of decimal floating-point processing. Additionally, a new final digit-set conversion is described in order to apply the carry free addition solely.

6.1.1 Carry Free Addition

In the conventional carry free addition algorithm, to obtain the transfer digit T_{i+1}, the operands X_i and Y_i have to be added together and compared with the threshold value. A carry chain in this process limits the performance of the carry free adder. To improve it, a speculative method could be used (e.g., [51], [53] and [54]). In this method, all possible results which depend on different transfer digits T_i and T_{i+1} are calculated simultaneously and the correct one is selected by the value of the transfer digits. The redundancy on hardware in aforementioned designs implies the bigger area and higher power consumption. On the other hand, the nonspeculative method for maximally redundant SD addition (e.g., [55])

provided a faster design in binary world (i.e., radix-2^h). In this section, a new nonspeculative decimal SD addition which directly calculates the result without the hardware redundancy is introduced. The proposed adder which works in digit set $[-9, 9]$ has a simple range division logic. Moreover, the operands and result are encoded in 5-bit two's complement to reuse the binary circuit as much as possible.

The Algorithm

To unify the addition and subtraction, a new operand Yop_i is defined in equation (6.1). Hence, the adder and subtractor could be represented by a unified model as shown in equation (6.2).

$$Yop_i = \begin{cases} Y_i & \text{if operation is } add \\ -Y_i & \text{if operation is } sub \end{cases} \tag{6.1}$$

$$X_i \pm Y_i = X_i + Yop_i \tag{6.2}$$

In the traditional carry free algorithm, the transfer digit T_{i+1} and the temporary sum W_i are generated from the position sum P_i. The process could be represented by equation (6.3).

$$\begin{aligned} T_{i+1} &= f(P_i) = f(X_i + Yop_i) \\ W_i &= g(P_i) = g(X_i + Yop_i) \end{aligned} \tag{6.3}$$

To reduce the timing delay and parallelize the transfer digit generation with the position sum calculation, the temporary sum W_i and transfer digit T_{i+1} could also be directly expressed in terms of X_i and Yop_i as shown in equation (6.4).

$$\begin{aligned} T_{i+1} &= f'(X_i, Yop_i) \\ W_i &= g'(X_i, Yop_i) \end{aligned} \tag{6.4}$$

In decimal sign digit number system, ± 9 should be avoided in temporary sum, otherwise, an incoming transfer digit could lead to a carry to the next digit. The position sum which is equal to ± 9 is called an exception in the proposed design. Furthermore, the exception detection will pull down the performance of the decimal SD adder compared with its binary counterpart. Hence, the less number of exceptional cases, the better it is.

Table 6.1: Range division directly based on operands

	Range of X_i and Yop_i	T_{i+1}	W_i	S_i		Correction Signal $cor^{4\ldots1} =$
				$T_i = -1 =$ "1**1**"	$T_i = 0/1 =$ "0**0/1**"	
$case_1$	$X_i \geq 1, Yop_i \geq 1$ and $(0,9),(9,0)$	1	$P_i - 10$	$P_i + (-11) = P_i + 1010\mathbf{1}$	$P_i + (-10)/(-9) = P_i + 1011\mathbf{0/1}$	1010, if $T_i^1 = 1$ 1011, if $T_i^1 = 0$
$case_2$	$X_i \geq 0, Yop_i \leq 0$ $X_i \leq 0, Yop_i \geq 0$ exclude $(0,\pm9),(\pm9,0)$	0	P_i	$P_i + (-1) = P_i + 1111\mathbf{1}$	$P_i + 0/1 = P_i + 0000\mathbf{0/1}$	1111, if $T_i^1 = 1$ 0000, if $T_i^1 = 0$
$case_3$	$X_i \leq -1, Yop_i \leq -1$ and $(0,-9),(-9,0)$	-1	$P_i + 10$	$P_i + 9 = P_i + 0100\mathbf{1}$	$P_i + 10/11 = P_i + 0101\mathbf{0/1}$	0100, if $T_i^1 = 1$ 0101, if $T_i^1 = 0$

To implement the range division and exception detection efficiently, a scheme to divide the cases for generating different values of transfer digit T_{i+1} is proposed in Table 6.1. In this method, there are four pairs of X_i and Yop_i need to be detected as an exception. Therefore, only six multi-input gates are applied to implement the exception detecting circuit. Furthermore, these gates could be reused to decide the transfer digits. Consequently, besides the exception detecting logic, only the most significant bits for X_i and Yop_i and simple logic are needed to determine the transfer digit.

In Algorithm I, once the transfer digit is obtained, W_i is generated by adding a correction value to P_i, and then, the result S_i is calculated by adding W_i with T_i. In this process, those two serial computations cause a limit on speed. To combine these two additions into one computation efficiently (i.e., the three operands addition $S_i = P_i - r \times T_{i+1} + T_i$), an analysis for the effect of the input range and incoming transfer bits on the decimal correction value is provided in Table 6.1.

To generate the correction signal, the transfer digit T_i from the last digit is used. Therefore, the decimal correction signal can be decided directly by T_i^1 and the range of input operands X_i and Yop_i, then the further computation for adding T_i is removed. The bold numbers in Table 6.1 show that the least significant bit of the correction signal is equal to T_i^0.

The Hardware Implementation

To reuse the well optimized circuits in binary world as much as possible, the operands X_i and Y_i are encoded in 2's complement. Therefore, the Yop_i is obtained by inverting Y_i with XOR gates controlled by the operation signal *op* which is the penalty of the subtraction. The increment on the least significant bit could be added as an incoming carry to the right most full adder.

In this design, the exception logic is minimized to four pairs of operands detection, and to improve the speed, the operands X_i and Y_i are directly used for the exception handling. In equation (6.5), the signals E_p and E_n are for positive exception (i.e., $(0, 9)$ or $(9, 0)$) and negative exception (i.e., $(0, -9)$ or $(-9, 0)$) respectively.

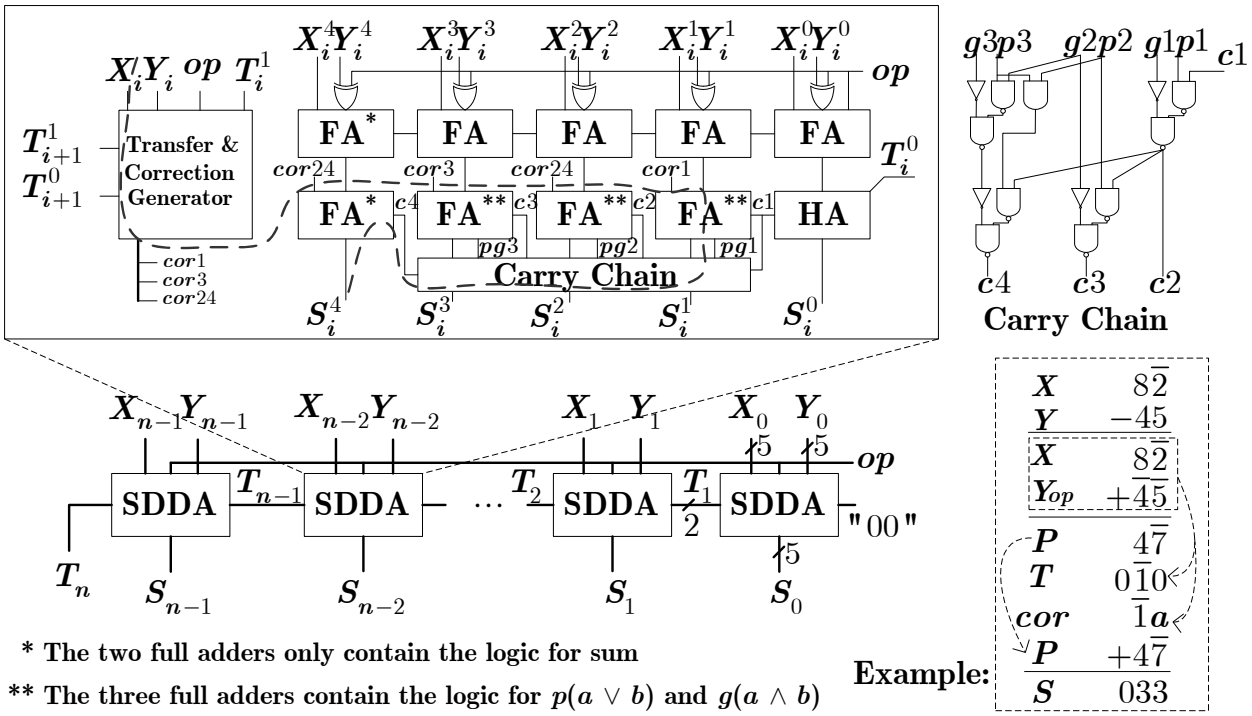


Figure 6.1: Proposed n-digit signed digit decimal adder

$$
\begin{aligned}
E_p = &\Big(X_i = 0 \wedge \big((Y_i = 9 \wedge op) \vee (Y_i = -9 \wedge \overline{op})\big)\Big) \vee \\
&(X_i = 9 \wedge Y_i = 0) \\
E_n = &\Big(X_i = 0 \wedge \big((Y_i = -9 \wedge op) \vee (Y_i = 9 \wedge \overline{op})\big)\Big) \vee \\
&(X_i = -9 \wedge Y_i = 0)
\end{aligned}
\tag{6.5}
$$

$$
\begin{aligned}
case_1 &= \big((\overline{X_i^4} \wedge \overline{X_i = 0}) \wedge (\overline{Yop_i^4} \wedge \overline{Y_i = 0})\big) \vee E_p \\
case_3 &= \big(X_i^4 \wedge (Yop_i^4 \wedge \overline{Y_i = 0})\big) \vee E_n \\
case_2 &= \Big(\big(\overline{X_i^4} \wedge (Yop_i^4 \vee Y_i = 0)\big) \vee \big((X_i^4 \vee X_i = 0) \wedge \\
&\qquad (\overline{Yop_i^4} \vee Y_i = 0)\big)\Big) \wedge \overline{E_p} \wedge \overline{E_n}
\end{aligned}
\tag{6.6}
$$

In Table 6.1, the operands' range division for generating T_{i+1} is not right on zero, thus, the zero input should be excluded for some cases. The zero detection logic in equation (6.5) could be reused, and the range division logic is given in equation (6.6).

The transfer digit only depends on the range division, and it can be obtained at the same time as P_i is ready. Thus, the critical path only passes one of the two units for transfer digit generation and position sum addition. Equation (6.7) shows the logics to generate the

transfer bits.

$$\begin{aligned} T_{i+1}^{1} &= case_3 \\ T_{i+1}^{0} &= case_1 \vee case_3 \end{aligned} \tag{6.7}$$

According to the analysis in Table 6.1, the decimal correction signals are decided by operands range and incoming transfer bits. The conditional adder with multiplexor which is controlled by T_i and $case_i$ could be applied. Nevertheless, to reduce the area, combinational logics shown in equation (6.8) are used to directly generate the correction signal and to connect it to the second level of binary full adders. An example of the process of the proposed adder is shown in Fig. 6.1.

$$\begin{aligned} cor^4 &= case_1 \vee (T_i^1 \wedge case_2) \\ cor^3 &= case_3 \vee (T_i^1 \wedge case_2) \\ cor^2 &= cor^4 \\ cor^1 &= \overline{T_i^1 \oplus case_2} \\ cor^0 &= T_i^0 \end{aligned} \tag{6.8}$$

Since the critical path passes through the second level of full adders, to further improve the performance of the proposed design, a simplified carry chain which is similar to the prefix network is applied as shown in Fig. 6.1.

Finally, the hardware implementation of the proposed decimal SD adder is given in Fig. 6.1. The bold dash line is the critical path which passes through the transfer and correction logic and an optimized carry chain. The full adders with the asterisk only contain the logics for sum. Furthermore, in the second level of full adders, the critical path only pass through one XOR gate in the left most full adder.

6.1.2 Absolute Value Digit-Set Conversion

Since the decimal data stored in memory are generally encoded in BCD format, to use the decimal carry free adder, the operands in BCD encoding should be converted into the internal encoding scheme used in the proposed design. Similarly the final result coming from the SD adder needs to be converted back to BCD format before sending to the memory.

In the IEEE 754-2008 Standard, the absolute value of the mantissa is represented in the significant digits section. However for the signed digit subtraction, the result could be less than zero. Hence, before sending to the memory, the result which is less than zero must be converted to it's absolute value.

In [58], a negation unit and prefix network are applied to correctly calculate the final result in BCD format. In [59] and [52], to convert the BCD encoding to the internal encoding, 9-level and 1-level of gates are used respectively. Further, the authors proposed two algorithms to convert from internal encoding to BCD encoding with a carry propagation chain. The negation unit for the redundant number system could be implemented digit by digit.

In this section, a merged algorithm which can directly convert the negative result to it's absolute value in BCD encoding with a less penalty on delay is introduced.

The Algorithm

In our design, since the digit set is encoded in 5-bit 2's complement, and the input operands in BCD encoding are always in digit set $[0, 9]$, thus the front conversion which is only a 1-bit sign extension does not cost any logic. For converting from the internal 5-bit format to the BCD encoding, a borrow (negative carry) propagation which passes through the entire word-width logics is involved.

For the negative SD result, before converting to the BCD encoding, the absolute value of it is obtained by inverting all signs on each digit. For example,

$$\left|(\bar{1}2\bar{3}\bar{4}010)_{SD}\right| = (1\bar{2}340\bar{1}0)_{SD} = (0833990)_{BCD}.$$

To merge the negation algorithm with the digit set conversion algorithm and improve the performance of the converter, an absolute value digit set conversion algorithm which includes a prefix network and a correction unit is proposed. The algorithm leads to a logarithmical timing delay which is more suitable for high precision computation. An example is provided in Fig. 6.2.

Algorithm 6.1.2: Absolute Value Digit Set Conversion

Data: SD number S.

Result: BCD number R ($R = |S|$).

1. Compute generate bit (G_i) and propagate bit (P_i) for each digit of the result.

$$G_i = \begin{cases} 1 & \text{if } S_i < 0 \\ 0 & \text{otherwise,} \end{cases} \quad P_i = \begin{cases} 1 & \text{if } S_i = 0 \\ 0 & \text{otherwise,} \end{cases}$$
$$G_{i:j} = \begin{cases} G_i & \text{if } i = j \\ G_i \vee (P_i \wedge G_{i-1:j}) & \text{if } i > j, \end{cases} \tag{6.9}$$
$$P_{i:j} = \begin{cases} P_i & \text{if } i = j \\ P_i \vee P_{i-1:j} & \text{if } i > j. \end{cases}$$

2. Compute the negative carry C_i.

$$C_{i+1} = G_{i:j} \vee (P_{i:j} \wedge C_j),\ C_0 = \begin{cases} 0 & \text{if } S \geq 0 \\ 1 & \text{if } S < 0. \end{cases} \tag{6.10}$$

3. Generate the result R_i in BCD format.

$$R_i = \begin{cases} S_i & \text{if } C_{i+1}C_i = 00 \text{ and } S \geq 0 \\ S_i - 1 & \text{if } C_{i+1}C_i = 01 \text{ and } S \geq 0 \\ S_i + 10 & \text{if } C_{i+1}C_i = 10 \text{ and } S \geq 0 \\ S_i + 9 & \text{if } C_{i+1}C_i = 11 \text{ and } S \geq 0 \\ \overline{S_i} + 10 & \text{if } C_{i+1}C_i = 00 \text{ and } S < 0 \\ \overline{S_i} + 11 & \text{if } C_{i+1}C_i = 01 \text{ and } S < 0 \\ \overline{S_i} & \text{if } C_{i+1}C_i = 10 \text{ and } S < 0 \\ \overline{S_i} + 1 & \text{if } C_{i+1}C_i = 11 \text{ and } S < 0, \end{cases} \tag{6.11}$$

where the symbol $\overline{S_i}$ is the bit inversion of the digit S_i.

The Hardware Implementation

The architecture of the proposed convertor for a p-digit input is given in Fig. 6.2. The C'_{msb} which is the sign of the SD result is obtained at the output of the prefix tree. Therefore if the SD result contains trailing zeros, then the C_0 which is C'_{msb} cannot be propagated

correctly. To fix this problem, a trailing zero detection is placed in parallel with the prefix tree, and a two-gate logic is applied to adjust the result coming from the prefix tree as shown in Fig. 6.3a.

In BCD encoding, only 4 bits are used to represent a number. Thus, the fifth bit S_i^4 of each digit in the SD result is discarded in the final 4-bit adder to compensate the result. The Fig. 6.3b shows the logic for the 4-bit correction signal for each digit which is obtained based on the equation (6.21) in Algorithm 6.1.2.

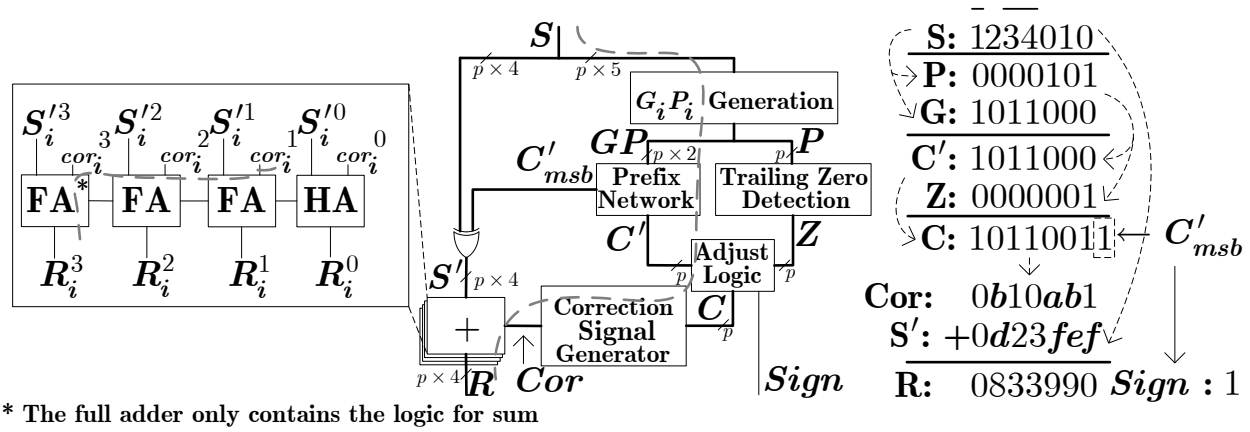
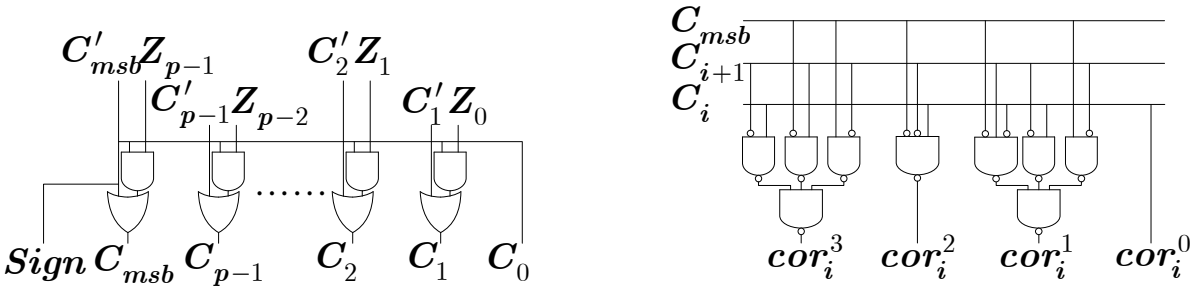


Figure 6.2: Proposed absolute value digit-set converter



(a) Adjust logic **(b)** Correction logic

Figure 6.3: Adjust and correction logics of the proposed digit-set converter

Before converting SD result to the BCD encoding, an exclusive-OR gate is applied on each bit of S_i to do the bit inversion for the negative result. In Fig. 6.2, the bold dashed line is the critical path which passes through the PG generation block, the prefix network, the adjustment logic, the correction signal generator and three full adders in the final converting unit. On the critical path, only the delay in prefix network is proportional to the width of

the input.

6.2 Parallel Decimal Fixed-point Multiplication

In the proposed parallel multiplication, one of the two operands is encoded into the digit-set $[-5, 5]$, and represent the multiples of the other operand from $-5X$ to $5X$ in the digit-set $[-8, 8]$. By doing so, all the multiples could be obtained in a constant delay, and only $n + 1$ partial products, namely PP, are generated. Furthermore, to reduce the $n+1$ levels of partial products into the final SD result, a multi-level multi-operand SD addition is discussed. To reduce the delay and area of the hardware in PPR unit, binary arithmetic units and combinational recoders are applied in the multi-operand SD adder. Finally a digit-set converter with hybrid carry propagation network is applied to convert the product from SD to BCD encoding. In the proposed hybrid prefix tree, different prefix trees with less digit width are applied to construct a big prefix carry propagation network. Consequently, in the prefix tree, the levels of prefix nodes after the longest column in the PPR unit are reduced. Overall, the structures of the PPG, PPR and final converter are balanced, and the delay of the proposed multiplier is optimized. The top level architecture of the proposed multiplication is shown in Fig. 6.4.

In the first stage, the n-digit operand Y_{BCD} is recoded into $(n + 1)$-digit Y_{SD} in digit-set $[-5, 5]$. The 5-bit "one hot" selection signal Ys_i for each digit is generated based on the recoded operand, Y_{SD}. In the proposed design, only the positive multiples, X through $5X$, are implemented by logic gates. The negative multiples, $-X$, $-2X$, $-3X$, $-4X$, and $-5X$, could be represented in the similar way. However, to reduce the area of the multiplier, the negative multiples are generated by inverting the sign on each digit of the positive multiples. Since the digit in the proposed SD multiples is represented in 2's complement encoding, the inversion is done by an XOR gate controlled by Yn_i which is also the increment bit for each digit to invert the sign of the multiples. Note that only one bit is enough to invert a partial product, therefore the increment bits for all digits in a partial product are identical.

The second stage in the proposed multiplication is a PPR unit implemented by multiple levels of multi-operand SD adders. For example in a 16×16-digit multiplication, after

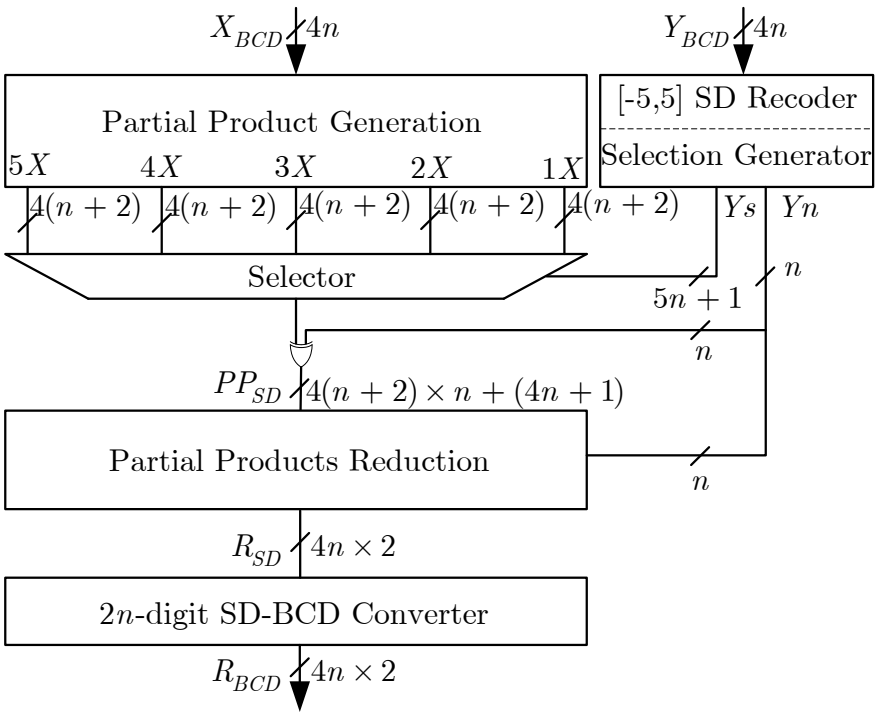


Figure 6.4: Top level architecture of the proposed parallel decimal multiplication

the PPG unit, there are 17 partial products need to be reduced. the layout of the partial product array is rearranged to apply two levels of SD adders to generate the final product in SD format. In such an multi-operand SD addition, the operands could be firstly reduced by the binary arithmetic unit, and in the end, a recoder is applied to correct the transfer digit and interim sum within the decimal manner. Thus in the proposed multiplication, the decimal correction is compacted as much as possible.

In the third stage, to convert the SD product back to BCD encoding efficiently, a hybrid carry propagation network which consists of several small carry prefix networks is provided to counterbalance the different delays on different bits of the result of the PPR unit. Compared to traditional methods, the hybrid prefix tree has less level by more level of nodes on the middle and less significant digits of the result from the PPR unit. Since the middle columns in a partial product array consume more delay than the ending columns, the overall delay of the multiplier could be further reduced with the proposed hybrid prefix tree. An example to illustrate each component of a proposed 4×4-digit multiplier is provided in Fig. 6.5.

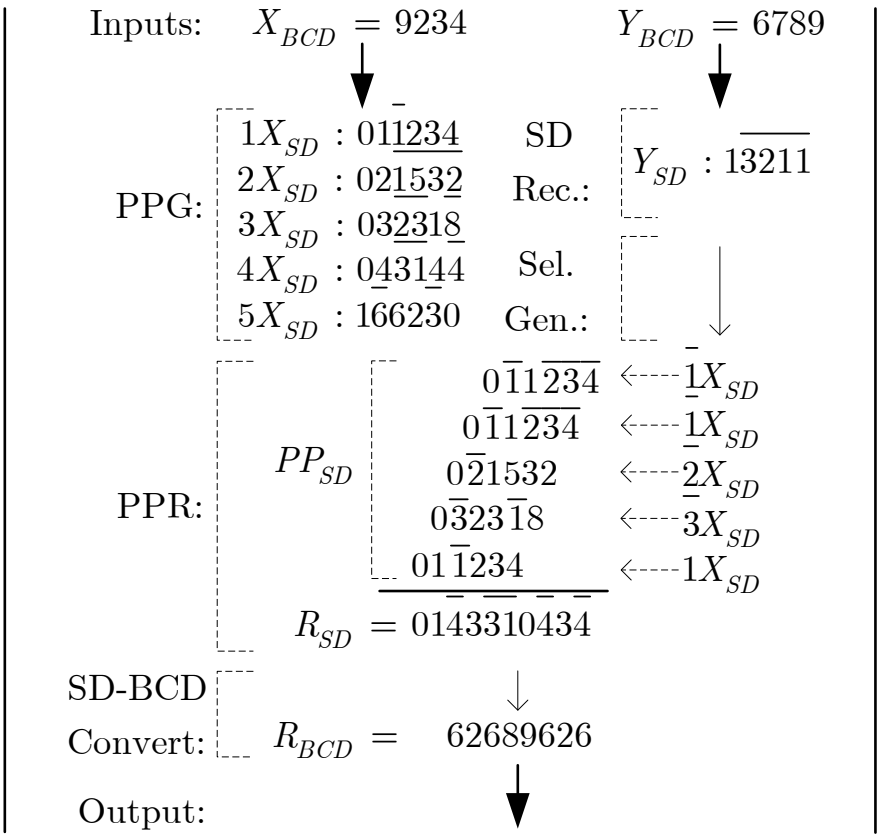


Figure 6.5: Example of the proposed 4×4-digit multiplication algorithm

6.2.1 Signed Digit Partial Product Generation

In the proposed multiplier, we follow the SD radix-10 method described in [68] to recode Y_{BCD} into the SD digit-set $[-5, 5]$. To represent multiples, a new method which generates $n + 1$ partial products without carry propagation is proposed. In Table 6.2, the positive multiples from $1X$ to $5X$ are represented in the SD digit set $[-8, 7]$. Thus a 4-bit 2's complement number could be applied to represent each digit in the multiples from $1X$ to $5X$. To reduce the area of the PPG unit, the negative multiples are generated by inverting the sign on each digit of the corresponding positive multiples, and the digit-set for all multiples from $-5X$ to $5X$ is extended to $[-8, 8]$ (i.e., $\{[-8, 7] \cup [-7, 8]\} \in [-8, 8]$). Unlike the binary signed digit encoding proposed in the decimal signed digit number system in [51], to invert the sign of each digit, an increment bit is involved in the proposed encoding system. Therefore one signed digit in the proposed multiples is represented by 5 bits. However, the penalty on the hardware area is minimized, since the increment bits for all digits in a partial product are

Table 6.2: Signed digit representation of the proposed multiples

BCD	$1X_i$		$2X_i$		$3X_i$		$4X_i$		$5X_i$			Y_i	
Operand	T_{i+1}	W_i	T_{i+1}	W_i	T_{i+1}	W_i	T_{i+1}	W_i	K_{i+2}	T_{i+1}	W_i	T_{i+1}	W_i
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	2	0	3	1	-6	0	0	5	0	1
2	0	2	1	-6	1	-4	1	-2	0	1	0	0	2
3	0	3	1	-4	1	-1	1	2	0	1	5	0	3
4	0	4	1	-2	2	-8	2	-4	1	-8	0	0	4
5	0	5	1	0	2	-5	2	0	1	-8	5	1	-5
6	1	-4	1	2	2	-2	3	-6	1	-7	0	1	-4
7	1	-3	1	4	2	1	3	-2	1	-7	5	1	-3
8	1	-2	2	-4	3	-6	3	2	1	-6	0	1	-2
9	1	-1	2	-2	3	-3	4	-4	1	-6	5	1	-1
$T_i + W_i$	$[-4, 6]$		$[-6, 6]$		$[-8, 6]$		$[-6, 6]$					$[-5, 5]$	
$K_i + T_i + W_i$									$[-8, 7]$				

identical.

Generation of Multiples

In Table 6.2, all the multiples could be divided into two parts except the $5X_i$ which is divided into three parts. To simplify the representation of the multiplies generation, three variables are defined in Table 6.2, where W_i represents the residual number which has the same weight as the current BCD digit. T_{i+1} and K_{i+2} are the transfer digits to the next two digits which have 10 and 100 times weight of the current BCD digit respectively. The sum of the three variables is restricted in the range of $[-8, 7]$ to form one digit in SD number. Since the variables can be directly generated according to different inputs, and the carry (transfer digit) never propagates exceeding three neighbor digits, the delay of the proposed PPG is independent on the width of the operand. In addition, for an n-digit operand, each multiple contains $n+2$ SD digits. The SD multiple could be obtained by adding K_i, T_i and W_i with a

4-bit adder after a recoder generating these variables. Due to the specific converting pattern in Table 6.2, the conversion can be assumed as a constant addition. Thus, the 4-bit add operation is optimized and converted to the combinational logic to reduce area and delay. The equations of one digit of the positive multiples are listed below. Note that the signals on the right side of the equal sign is in BCD encoding, and the signals on the left side of the equal sign is in proposed SD encoding. The hardware implementation can be optimized with logic gates with less delay (e.g., NAND, NOR, XNOR gates).

1X: Since the digit-set $[-8, 7]$ is applied to generate the positive multiples in the proposed PPG algorithm, $1X$ has to be converted to the target digit-set. In the equation, the signal T_i represents the incoming transfer digit which is determined by the previous one digit.

$$\begin{aligned}
T_i = \quad & X_{i-1}^3 + X_{i-1}^2 X_{i-1}^1 \\
1X_i^3 = \quad & X_i^3(\overline{X}_i^0 + \overline{T}_i) + X_i^2 X_i^1 \\
1X_i^2 = \quad & T_i X_i^1 X_i^0 + X_i^3(\overline{X}_i^0 + \overline{T}_i) + X_i^2 \\
1X_i^1 = \quad & T_i X_i^0(\overline{X}_i^3 \overline{X}_i^1 + X_i^2) + \\
& (\overline{X}_i^0 + \overline{T}_i)(\overline{X}_i^2 X_i^1 + X_i^3) \\
1X_i^0 = \quad & T_i \oplus X_i^0
\end{aligned} \tag{6.12}$$

2X: In the proposed algorithm, since the transfer digit from last digit in multiple $2X$ could be from 0 to 2, two bits (i.e., T_i^1 and T_i^0) are needed to represent the incoming transfer digit.

$$\begin{aligned}
T_i^1 = \quad & X_{i-1}^3 \\
T_i^0 = \quad & X_{i-1}^2 + X_{i-1}^1 \\
2X_i^3 = \quad & \overline{X}_i^0(\overline{T}_i^1 X_i^2 \overline{X}_i^1 + X_i^3) + \overline{X}_i^2 X_i^1 + \overline{T}_i^1 X_i^3 \\
2X_i^2 = \quad & X_i^0(T_i^1 \overline{X}_i^3 \overline{X}_i^2 + X_i^1) + T_i^1 X_i^1 + \\
& \overline{X}_i^0(\overline{T}_i^1 X_i^2 \overline{X}_i^1 + X_i^3) + \overline{T}_i^1 X_i^3 \\
2X_i^1 = \quad & \overline{X}_i^2 \overline{X}_i^1(T_i^1 \overline{X}_i^0 + \overline{T}_i^1 X_i^0) + \\
& (\overline{T}_i^1 \overline{X}_i^0 + T_i^1 X_i^0)(X_i^1 + X_i^2) \\
2X_i^0 = \quad & T_i^0
\end{aligned} \tag{6.13}$$

5X: To generate multiple $5X$ in digit-set $[-8, 7]$, two transfer digits which have 10 and 100 times weight of the current digit are needed. Since only two elements are in the digit-sets

of the residual number W_i and the transfer digit K_i, the logic could be simplified as shown in equation (6.14).

$$\begin{aligned}
W_i = &\quad X_i^0 \\
K_i = &\quad X_{i-2}^3 + X_{i-2}^2 \\
5X_i^3 = &\quad X_{i-1}^3(\overline{K}_i + \overline{W}_i) + X_{i-1}^2 \\
5X_i^2 = &\quad W_i(\overline{X}_{i-1}^3 + \overline{K}_i) \\
5X_i^1 = &\quad W_i K_i \overline{X}_{i-1}^3 + X_{i-1}^3(\overline{K}_i + \overline{W}_i) + \\
&\quad X_{i-1}^1(K_i + W_i) \\
5X_i^0 = &\quad X_{i-1}^1 \oplus (W_i \oplus K_i)
\end{aligned} \tag{6.14}$$

3X: By applying the redundant number system to represent the partial product, the $3X$ logic does not contain the carry propagation in digit level any more. Thus a constant delay in PPG could be achieved.

$$\begin{aligned}
T_i^1 = &\quad X_{i-1}^3 + X_{i-1}^2 \\
T_i^0 = &\quad X_{i-1}^3 + \overline{X}_{i-1}^2 X_{i-1}^1 \\
3X_i^3 = &\quad X_i^3(\overline{X}_i^0 + \overline{T}_i^0 + \overline{T}_i^1) + X_i^2\overline{X}_i^1 + \\
&\quad X_i^1(\overline{T}_i^0\overline{T}_i^1\overline{X}_i^2 + \overline{X}_i^0(\overline{X}_i^2 + \overline{T}_i^1)) \\
3X_i^2 = &\quad T_i^1 X_i^3 \overline{X}_i^0 + X_i^1(\overline{T}_i^0\overline{T}_i^1\overline{X}_i^2 + \overline{X}_i^0(\overline{X}_i^2 + \overline{T}_i^1)) + \\
&\quad X_i^0(T_i^0 T_i^1 X_i^2 + \overline{X}_i^1(T_i^0\overline{X}_i^3 + T_i^1\overline{T}_i^0) + \overline{T}_i^1 X_i^3) \\
3X_i^1 = &\quad \overline{X}_i^3(\overline{T}_i^1\overline{T}_i^0 X_i^0 + T_i^1(\overline{X}_i^0 + T_i^0))(\overline{X}_i^1 + \overline{X}_i^2) + \\
&\quad (T_i^1\overline{T}_i^0 X_i^0 + \overline{T}_i^1(\overline{X}_i^0 + T_i^0))(X_i^2 X_i^1 + X_i^3) \\
3X_i^0 = &\quad T_i^0\overline{X}_i^0 + \overline{T}_i^0 X_i^0
\end{aligned} \tag{6.15}$$

4X: The multiple $4X$ in the proposed work is not generated based on two times of $2X$

as in other works. A direct and simple method is shown below.

$$
\begin{aligned}
4X_i^3 = \;& \overline{X}_i^3\overline{X}_i^2\overline{X}_i^1X_i^0 + X_i^2X_i^1\overline{X}_i^0 + \\
& \overline{X}_{i-1}^0(\overline{X}_i^2\overline{X}_i^1X_i^0 + X_i^2\overline{X}_i^0) + \\
& \overline{X}_{i-1}^3(X_i^1\overline{X}_{i-1}^2(\overline{X}_i^0 + X_i^2) + \overline{X}_i^2\overline{X}_i^1X_i^0 + X_i^2\overline{X}_i^0) \\
4X_i^2 = \;& X_i^3(X_{i-1}^3\overline{X}_{i-1}^0 + X_{i-1}^2) + \\
& X_i^0(\overline{X}_i^3X_{i-1}^3(\overline{X}_i^1X_{i-1}^0 + \overline{X}_i^2) \\
& + \overline{X}_{i-1}^3(X_i^2X_i^1\overline{X}_{i-1}^2 + X_i^3) + \overline{X}_i^2X_{i-1}^2) + \\
& \overline{X}_i^0(\overline{X}_i^2(\overline{X}_i^1X_{i-1}^3X_{i-1}^0 + X_i^1\overline{X}_{i-1}^3\overline{X}_{i-1}^2) \\
& + X_i^2(X_{i-1}^3(\overline{X}_{i-1}^0 + X_i^1) + \overline{X}_i^1\overline{X}_{i-1}^3 + X_{i-1}^2)) \\
4X_i^1 = \;& \overline{X}_{i-1}^2(X_{i-1}^0 + \overline{X}_{i-1}^3)(X_i^3\overline{X}_i^0 + \overline{X}_i^3\overline{X}_i^2X_i^0 + X_i^1) + \\
& (X_{i-1}^3\overline{X}_{i-1}^0 + X_{i-1}^2)(\overline{X}_i^1(\overline{X}_i^3\overline{X}_i^0 + X_i^2) + X_i^3X_i^0) \\
4X_i^0 = \;& \overline{X}_{i-1}^3\overline{X}_{i-1}^2X_{i-1}^0 + X_{i-1}^3\overline{X}_{i-1}^0 + X_{i-1}^1
\end{aligned}
\tag{6.16}
$$

Selection of Partial Product

In the proposed multiplier, a minimally redundant radix-10 digit-set $[-5, 5]$ is applied to represent the operand Y. Since the recoded set is symmetrical, and the multiples are encoded in signed digit number, the selection signals for the negative multiples are the same as the positive multiples (i.e., $Ys_i^{4\cdots0}$ indicate the signals to select $\pm5X, \ldots, \pm1X$). If a negative multiple is selected, a one-bit negation signal Yn_i for each selected partial product is applied to invert the signs of all digits in the corresponding positive multiple. The equations for the selection signal and negation signal are given in equation (6.17).

$$
\begin{aligned}
T_i = \;& Y_{i-1}^2(Y_{i-1}^0 + Y_{i-1}^1) + Y_{i-1}^3 \\
Ys_i^4 = \;& \overline{Y}_i^2\overline{Y}_i^1(T_i\overline{Y}_i^0 + \overline{T}_iY_i^0) \\
Ys_i^3 = \;& T_iY_i^0(\overline{Y}_i^3\overline{Y}_i^2\overline{Y}_i^1 + Y_i^2Y_i^1) + \\
& \overline{T}_i\overline{Y}_i^0(\overline{Y}_i^2Y_i^1 + Y_i^3) \\
Ys_i^2 = \;& Y_i^1(T_i\overline{Y}_i^0 + \overline{T}_iY_i^0) \\
Ys_i^1 = \;& T_iY_i^0(Y_i^2\overline{Y}_i^1 + \overline{Y}_i^2Y_i^1) + \overline{T}_iY_i^2\overline{Y}_i^0 \\
Ys_i^0 = \;& Y_i^2\overline{Y}_i^1(T_i\overline{Y}_i^0 + \overline{T}_iY_i^0) \\
Yn_i = \;& Y_i^3(\overline{Y}_i^0 + \overline{T}_i) + Y_i^2(Y_i^0 + Y_i^1)
\end{aligned}
\tag{6.17}
$$

In Table 6.2, the column for the T_{i+1} of Y_i shows that an n-digit operand Y_{BCD} could generate an $(n+1)$-digit SD recoded operand Y_{SD}, and the $(n+1)^{\text{th}}$ digit in Y_{SD} can only be 0 or 1. Thus for the $(n+1)^{\text{th}}$ partial product can only be $0X$ (all zeros) or $1X$. Furthermore, since the $(n+2)^{\text{th}}$ digit of the multiple $1X$ is always zero, and the $(n+1)^{\text{th}}$ digit of the $1X$ can only be 0 or 1, only 1 bit is enough to represent the most significant two digits in the $(n+1)^{\text{th}}$ partial product, PP_n. Thus, the selection logic for PP_n could be simplified. Additionally, the actual bit-widths on the output of the PPG are $4\times(n+2)\times n+(4\times n+1)$ for the partial product PP and n for the inversion signal Y_n. The detailed structure of the PPG is shown in Fig. 6.6.

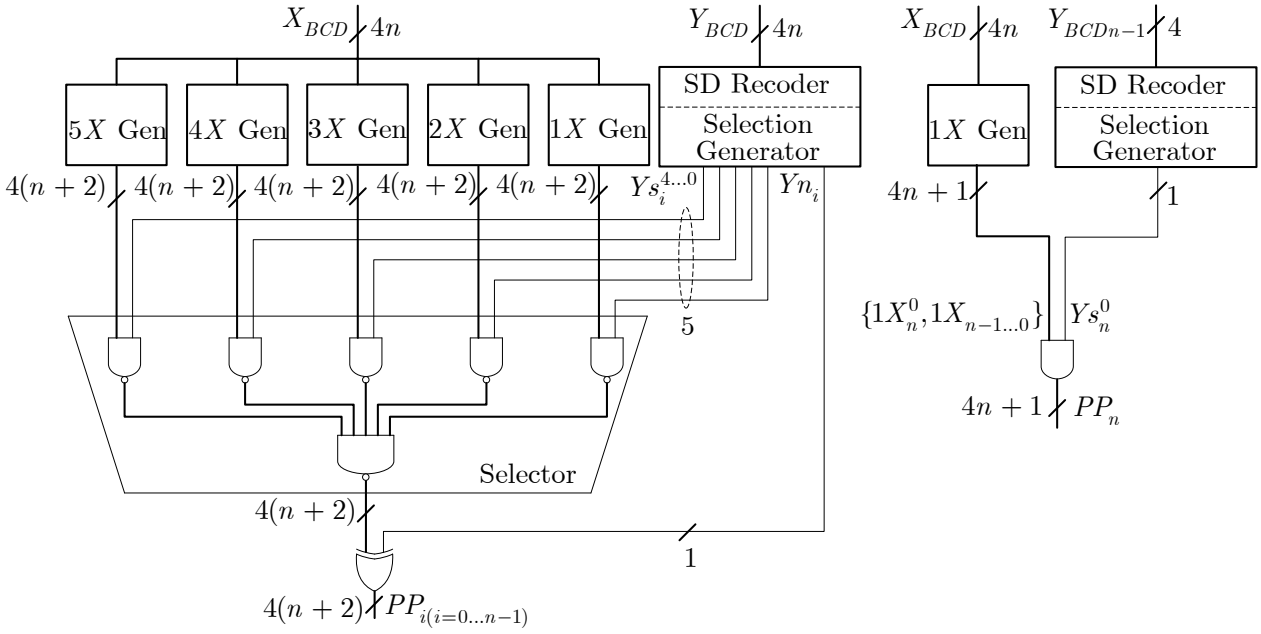
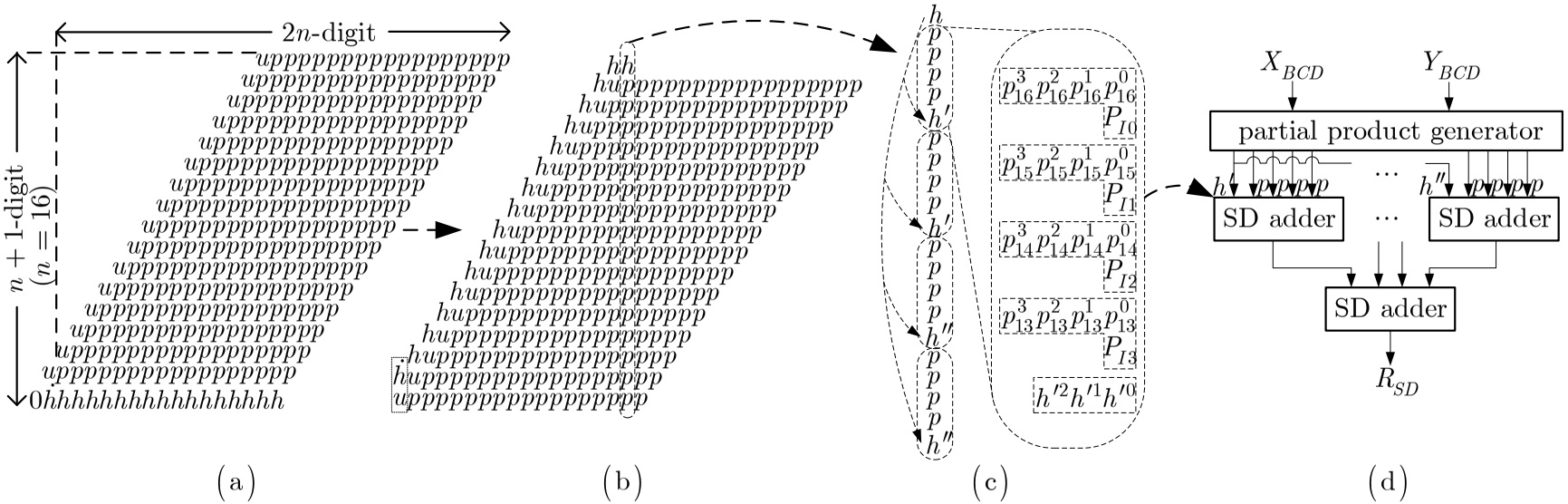


Figure 6.6: Proposed architecture of partial product generation

6.2.2 SD Partial Product Reduction

To illustrate the proposed algorithm, a PPR scheme of a 16 × 16-digit multiplier is implemented and discussed. First, the layout of the partial product array and the basic structure of the PPR unit are introduced. Subsequently, a PPR algorithm based on multi-operand SD addition is discussed. Finally, a hardware implementation of the proposed PPR unit for a 16 × 16-digit multiplier is addressed. Additionally, the delay model in terms of the delay of a binary full adder is analyzed to guide in designing of the proposed SD-BCD converter.



p: one digit of partial products (PPs),$p \in [-8,8]$.

h: one digit of the 17^{th} partial product,$h \in [-4,6]$.

h', h'': combinations of one digit $h, h' \in [-2,2], h'' \in [0,1]$.

P_{Ii}: an increment bit of digits in the i^{th}partial product.

$\dot{h}$: the 17^{th} digit of the 17^{th} partital product,$\dot{h} \in [0,1]$.

u: the most significant digit of a partital product, $u \in [-1,1]$.

Figure 6.7: Restructure of the proposed partial product reduction

Partial Product Reduction Array

As described in section 4, for the multiplication of two n-digit operands, $n+1$ partial products in $(n+2)$-digit are generated from the PPG unit in the proposed algorithm. Then the $n+1$ partial products need to be shifted according to the weight of each digit in the second operand. Finally, these $n+1$ shifted partial products are added by the SD multi-operand adders.

An example of the layout of partial products for the proposed 16×16-digit multiplication is shown in Fig. 6.7(a). The partial product $0\dot{h}h \ldots hh$ indicates the partial product generated by the most significant digit (MSD) of the recoded operand Y_{SD}. Recalling the description in section 6.3, the MSD of Y_{SD} only can be 1 or 0, and the 18^{th} digit of PP_{16} is always zero for a 16×16-digit multiplication. Furthermore, the 17^{th} digit of PP_{16}, $\dot{h}$, is in $[0, 1]$ as shown in Table 6.2. The $up \ldots pp$ represents the partial products generated according to the least 16 significant digits of Y_{SD}. Since the 18^{th} digit may be ± 1 only in $\pm 5X$, the range of u is restricted to $[-1, 1]$.

In Fig. 6.7(b), the layout of the partial product array is rearranged. Thus, except the middle two partial product columns, all other columns are not more than 16 digits. For a 16×16-digit decimal multiplication, the result is maximally in 32 digits which is the product of two operands with 16 consecutive nines. If the product in SD format is not going to be used by other SD arithmetic units before converting back to BCD format, then the digits beyond the least 32 digits can be discarded (e.g., the digits in a dashed rectangular). For example, in a 16×16-digit multiplication, the least 33 digits of the SD product can only be in one of the formats shown in equation (6.18). Otherwise, after converting back to BCD format, it will be larger than 32 digits.

$$R_{SD} = \begin{cases} 1\overline{d}\ldots & \text{, or} \\ 10\ldots0\overline{d}\ldots & \text{, or} \\ 0d\ldots & \text{, or} \\ 00\ldots0d\ldots & . \end{cases} \tag{6.18}$$

where d is a positive decimal digit, and $\overline{d}$ is the negation of d. The range of d is dependent on the digit set applied (e.g., $d \in [1, 5]$ for the digit set [-5,5]).

The leading one in the first case in equation (6.18) will be reduced by one to form a ten in the less significant digit to cancel out the negative digit $\overline{d}$. The 32^{nd} digit of the SD result should be converted to $(10+\overline{d})$ or $(10+\overline{d}-1)$ only depending on the value of the digits on the same position and less significant positions. In the second case, the 0-sequence on the right side of the leading one should be converted to a 9-sequence to cancel out the first negative digit on its right side with the same manner as mentioned in the first case. In the latter two cases, the most significant positive digit will be converted to d or $d-1$ depending on less significant digits. Furthermore, the most significant positive digit guarantees that no extra borrow is propagated to its left side. Hence, the conversion of an SD digit only depends on the sign of itself and the less significant digits. Consequently, in all of the cases the 33^{rd} digit is always zero in the result in BCD format. The details of the conversion algorithm to correctly generate the result in BCD format is discussed in section 6.

16 partial products can be divided into four groups in which a 4-operand SD adder is applied. Subsequently, 4 results of the first level of 4-operand SD adders are summed up by a 4-operand SD adder in the second level. In the middle two columns in the partial product array, there are 17 partial products which potentially cause a complicated design. In the proposed algorithm, the 17^{th} operand is recoded into four subtle numbers (i.e., h', h'') as shown in Fig. 6.7(c), and issue them into four SD adders in the first level. The maximum number of operands for the first level of SD adder is shown in the dashed circle. Note that the increment signals P_{Ii} for all digits in one partial product are identical. The dataflow of the PPR algorithm is given in Fig. 6.7(d). As shown in Fig. 6.7(d), two levels of SD adders are applied in the proposed PPR unit. Furthermore, as shown in the next section, the multi-operand adder to process four p and one h' has the same complexity as the adder for four p.

Multi-Operand SD Addition Algorithm

In the proposed multiplier, the $n+1$ partial products are encoded in SD digit-set $[-8, 8]$ within 4 bits 2's complement number and 1 bit increment. The partial product reduction is indeed a multi-operand SD addition. Although in principle, the result of the multi-operand SD addition could be in the same digit-set as the input operands, to reduce the number of

internal wires, the result of the SD addition is retained in $[-8, 7]$ in which the 1-bit increment signal is removed. An SD addition could be simply summarized into three steps, which are adding operands to get position sum ps_i, extracting transfer digit t_{i+1} and obtaining interim sum $w_i = ps_i - 10t_{i+1}$ (suppose radix is 10), and computing final sum $s_i = w_i + t_i$. Actually, a two-operand SD adder can be applied as the minimum element in the PPR unit, and the position sum is corrected (i.e., $ps_i - 10t_{i+1}$) for each addition. However, the correcting operation is not immediately needed, and can be postponed to reduce the delay and area of the PPR. In Table 6.3, the cases for multiple operands in the SD addition are shown. The range of ps_i limits the selection of t_i, and the range of w_i cannot be decreased infinitely to cover all the digits in a decimal range [0,9]. Table 6.3 shows that as the range of ps_i increases, the ranges of t_i and s_i increase. To restrict the range of s_i in $[-8, 7]$, the maximum number of operands in $[-8, 8]$ is four.

Table 6.3: Analysis of the number of operands of SD addition

#Op.	Range of ps_i	Range of t_i	Range of w_i	Range of s_i
2	$[-16, 16]$	$[-1, 1]$	[-6,6]	[-7,7]
3	$[-24, 24]$	$[-2, 2]$	[-5,5]	[-7,7]
4	$[-32, 32]$	$[-3, 3]$	[-5,4]	[-8,7]
5	$[-40, 40]$	$[-4, 4]$	[-5,4]	[-9,8]

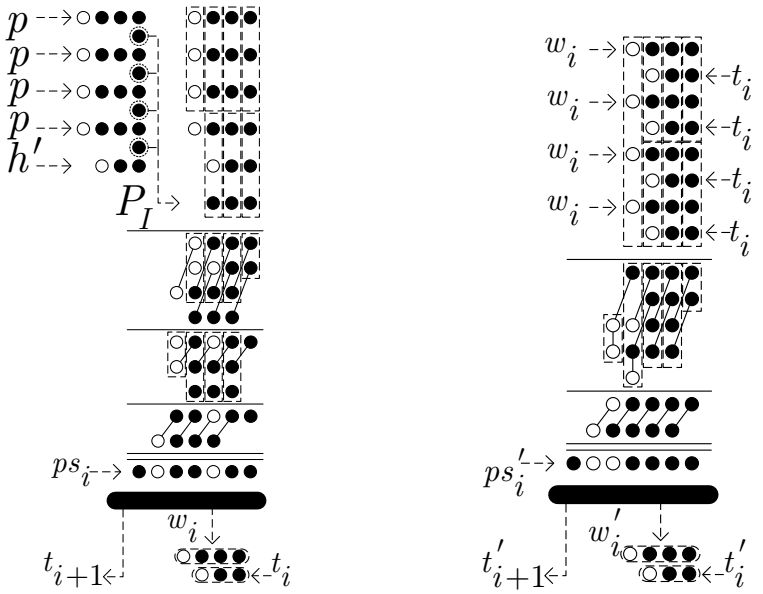
If the t_i and w_i are in the ranges of $[-3, 3]$ and $[-5, 4]$ in the proposed algorithm, the maximum range of the position sum ps_i can reach to $[-35, 34]$. The extra range out of $[-32, 32]$ (i.e., sum of four numbers in $[-8, 8]$) implies that the number of operands of the addition on $[-8, 8]$ may be between 4 and 5. In Fig. 6.7(c), the 17^{th} operand, $h \in [-4, 6]$, is recoded into four parts, and the maximum range of the subtle numbers (i.e., the h' and h'') is $[-2, 2]$. Thus, it is possible to add four operands with the subtle number together without overflow on the number system. The process of the SD addition according to our proposed number system is listed in the Table 6.4. In the proposed SD addition, the operands are summed up with binary arithmetic. To do the decimal correction, a recoder which maps the binary position sum ps (ps') to the decimal transfer digit t (t') and interim sum w (w') is applied in each level of SD addition.

Since the signed digit operands are involved in the multi-operand addition, the addition algorithm of weighted bit-set (WBS) encoding proposed in [78] is applied and extended for multiple operands and multiple bit-widths in our algorithm. In Fig. 6.8, the proposed two levels of SD additions are illustrated by the dot notation representation which is proposed in [78]. In Fig. 6.8 the white circle represents a binary bit with negative weight, namely negabit, and the black circle represents a binary bit with positive weight, namely posibit. Additionally, the carry save half adder, full adder, and 4:2 compressor are respectively represented by the dashed rectangles with 2, 3, and 4 circles. The solid line, solid double-line, and bold solid line represent one level of carry save arithmetic units, a carry lookahead adder, and a recoder, respectively.

Table 6.4: Proposed SD addition algorithm

Addition Steps	SD addition operands		Symbols
	Digit i+1	Digit i	
level1-step1: sum the partial products $ps = p + p + p + p + h'$	$4\times[-8,8]$ $+\ [-2,2]$ $[-34,34]$	$4\times[-8,8]$ $+\ [-2,2]$ $[-34,34]$	p h' ps
level1-step2: generate t_{i+1} and w_i calculate $s_i = t_i + w_i$	$[-5,4]$ $+\ [-3,3]$ $[-8,7]$	$[-5,4]$ $+\ [-3,3]$ $[-8,7]$	w t s
level2-step1: sum the four SD results $ps' = s + s + s + s$	$4\times[-8,7]$ $[-32,28]$	$4\times[-8,7]$ $[-32,28]$	ps'
level2-step2: generate t'_{i+1} and w'_i calculate $s'_i = t'_i + w'_i$	$[-5,4]$ $+\ [-3,3]$ $[-8,7]$	$[-5,4]$ $+\ [-3,3]$ $[-8,7]$	w' t' s'

As shown in Fig. 6.8, the transfer digits and interim sums from the first level of SD addition are summed up directly in the second level of SD addition to avoid the delay cost of a carry lookahead adder to add w and t. Therefore, the step 2 of the first level of addition and the step 1 of the second level of addition proposed in Table 6.4 are merged together. Furthermore, to reduce the number of the arithmetic units in the hardware implementation, the sign bit of the operands (i.e., h' and P_I) is not extended. Thus the position sum ps (ps') is given in hybrid posibit-negabit encoding. For example, the third bit and sixth bit of ps have negative weight -2^2 and -2^5. Note that in Fig. 6.8(a), the increment signal P_I for



(a) First level (b) Second level

Figure 6.8: Dot notation of the proposed two levels of multi-operand SD additions

each digit is summed up by a binary counter to reduce the number of operands in the least significant bit of each SD adder. Such a counter can be applied right after the Radix-10 operand recoder of the operand Y, thus it cannot affect the critical path. Additionally, since the increment bits for all digits in a partial product are identical, the number of the counters can be minimized.

The hybrid posibit-negabit encoded binary to signed digit decimal recoder which is a one-to-one mapping can be implemented in the combinational logic. A segment of the map in binary bits to recode ps_i and ps'_i is given in Table 6.5. As shown in Fig 6.8, the ps is represented in hybrid posibit-negabit encoding, and the negative weighted bits are placed at the third and sixth binary positions. Thus, in the recoder, an input of "1100010" (34) generates "011" (3) as t and "0100" (4) as w.

Hardware Implementation and Delay Model of the Proposed PPR

As shown in Fig. 6.8(a), the maximum bits of operands of the first level SD adder are six, which can be reduced to one carry-sum pair by 3 levels of binary full adders (FA) and half adders (HA) as shown in Fig. 6.9. By applying the WBS adder, the inverters are placed on the input or output of the traditional arithmetic unit, such as a full adder. As shown in [79],

Table 6.5: Proposed transfer digit and interim sum recoder

Recoder in 1st-level SD adder			Recoder in 2nd-level SD adder		
ps_i	t_{i+1}	w_i	ps'_i	t'_{i+1}	w'_i
"1100010"	"011"	"0100"	"1111100"	"011"	"1110"
"1100001"	"011"	"0011"	"1111011"	"011"	"1101"
"1100000"	"011"	"0010"	"1111010"	"011"	"1100"
"1100111"	"011"	"0001"	"1111001"	"011"	"1011"
"1100110"	"011"	"0000"	"1111000"	"010"	"0100"
"1100101"	"011"	"1111"	"1110111"	"010"	"0011"
"1100100"	"011"	"1110"	"1110110"	"010"	"0010"
"0011011"	"011"	"1101"	"1110101"	"010"	"0001"
"0011010"	"011"	"1100"	"1110100"	"010"	"0000"
"0011001"	"011"	"1011"	"1110011"	"010"	"1111"
"0011000"	"010"	"0100"	"1110010"	"010"	"1110"
.....					
"0100110"	"101"	"1100"	"0100000"	"101"	"1110"

[80], and [78], the inverters in between the arithmetic units can be canceled. The remaining inverters at the input and output of the calculation unit could be absorbed by the previous logic. For example, the inverters of the negabits $\overline{p}^3_{i...i+3}$ can be removed by the XOR gates at the output port of the PPG with inverted logic (i.e., XNOR gate). To save the delay on the critical path, the transfer digit t_i and the interim sum w_i generated by the first level of SD adders is kept. Additionally, the eight internal parameters are added by two levels of binary 4:2 compressors as shown in Fig. 6.10. Since the recoder inside the PPR unit is a simple one-to-one mapping from the inputs to the outputs, the recoders described in Table 6.5 are simply created by the combinational logic gates. Note that, except two middle columns of operands as shown in Fig. 6.7(b), all other columns can be reduced with elements no more complicated than the adders on the critical path which are shown in Fig. 6.9 and Fig. 6.10. For example, 12 operands can be reduced by four 3-operand SD adders on the first level and one 4-operand SD adder on the second level. Thus the area of the PPR is potentially reduced. Finally, a segment of the top level architecture of the SD adders in the PPR unit for a 16×16-digit multiplier is given in Fig. 6.11.

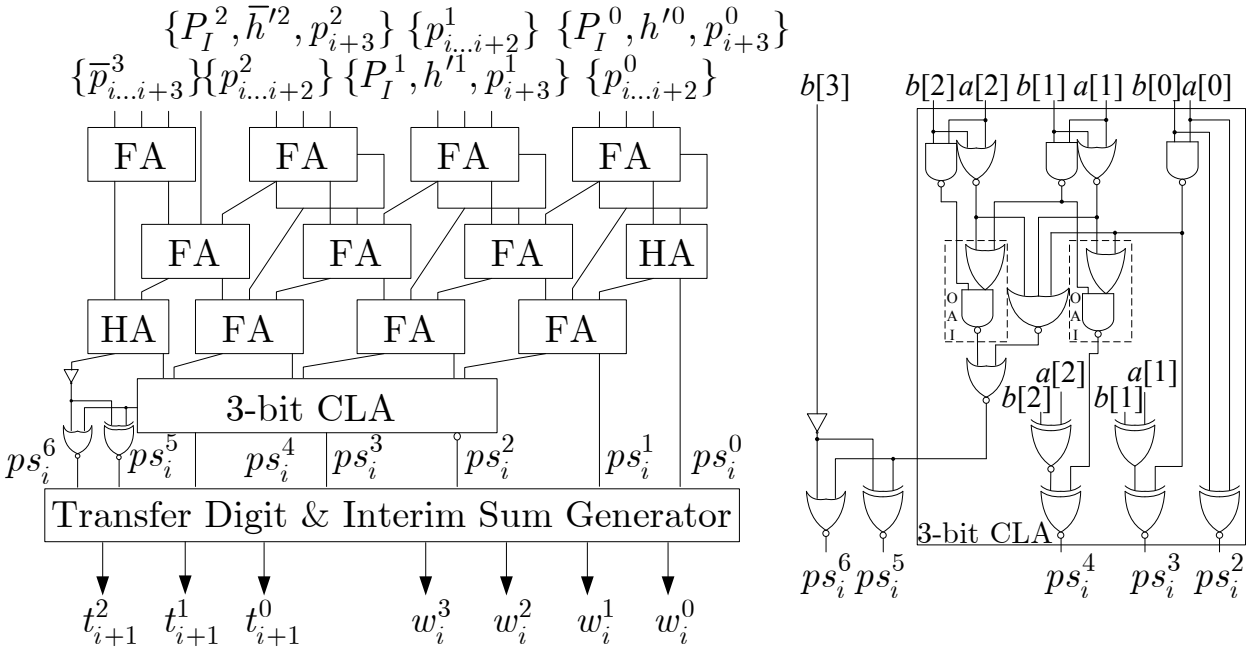


Figure 6.9: Hardware structure of the proposed 1st level multi-operand SD adder

In addition, the different structures of columns of PPR unit make the result signals of different digits of the PPR available at different time. To analyze the delay on each digit

Table 6.6: Delay analysis of each digit of the proposed partial product reduction

Logic	Column Position															
Module	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Bin 3:2	1	1		3	3	3	3								3	3
Bin 4:2*	1.5	1.5	3	1.5	1.5	1.5	1.5	3	4.5	4.5	3	3	3	3	3	3
Bin 5:2*								2			2	2	2	2		
3-bit CLA*	2	2	1	1	1	1	1								1	1
4-bit CLA*			1.25	1.25	1.25	1.25	1.25	2.5	2.5	2.5	2.5	2.5	2.5	2.5	1.25	1.25
Recoders*	2.5	2.5	2.75	3	3	3	3	3	3	3	3	3	3.25	3.25	3.5	3.5
equivalent BFAs	7	7	8	9.75	9.75	9.75	9.75	10.5	10	10	10.5	10.5	10.75	10.75	11.75	11.75

Logic	Column Position															
Module	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bin 3:2								3							1	
Bin 4:2*	3	3	3	3	4.5	4.5	4.5		1.5	1.5	3	1.5		1.5		
Bin 5:2*	2	2	2	2				2	2	2			2			
3-bit CLA*								1	1	1	1				1	
4-bit CLA*	2.5	2.5	2.5	2.5	2.5	2.5	2.5	1.25	1.25	1.25	1.25	2.5	1.25	1.25		
Recoders*	3.25	3.25	3	3	3	3	3	3	2.5	2.5	2.75	2.5	1.5	1.5	1.25	1
equivalent BFAs	10.75	10.75	10.5	10.5	10	10	10	10.25	8.25	8.25	8	6.5	4.75	4.25	3.25	1

* The delay is represented in the number of equivalent BFAs.

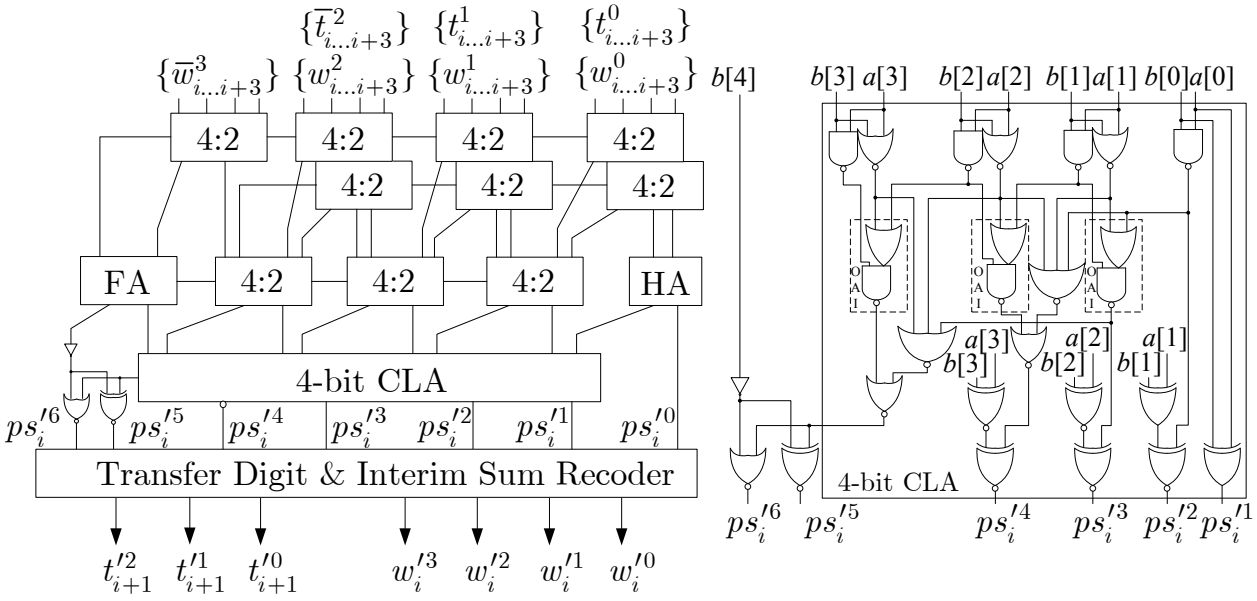


Figure 6.10: Hardware structure of the proposed 2nd level multi-operand SD adder

of the output of the PPR, a list of equivalent binary full adders in modules on the critical paths in each column is shown in Table 6.6. We assume that the binary 4:2 compressor has a delay of 1.5 binary full adder (BFA), and the binary 5:2 compressor has a delay which equals to 2 BFAs' delay [77]. According to the delay analysis, we assume that the 3-bit and 4-bit carry lookahead adder (CLA) have delay of 1 BFA and 1.25 BFAs on the critical path which passes through ps_i^4 and $ps_i'^4$, respectively. The delay of the combinational recoders is also represented in terms of the 3:2 BFA which is obtained by the delay analysis. Thus, the brief estimation of the delay on each digit of PPR could be obtained in terms of the equivalent binary full adders. In Table 6.6, the delay from connected neighbor columns is considered. Additionally, since the latency to generate each partial product in PPG and the delay of the CLA to add w' and t' for each digit are almost the same, the influence of the delay of the PPG stage and the final CLA in Table 6.6 is not considered.

For a 34×34-digit multiplication, at most 35 partial products should be reduced. The 35 partial produces could be divided into three groups (i.e., double 17 partial products and one extra partial product). For the double 17 partial products, the proposed structure could be applied to obtain two SD results. Thus one more level of 3-operand SD adders are applied on the critical path to reduce the two SD results of the $17 : 2$ SD addition and the extra one

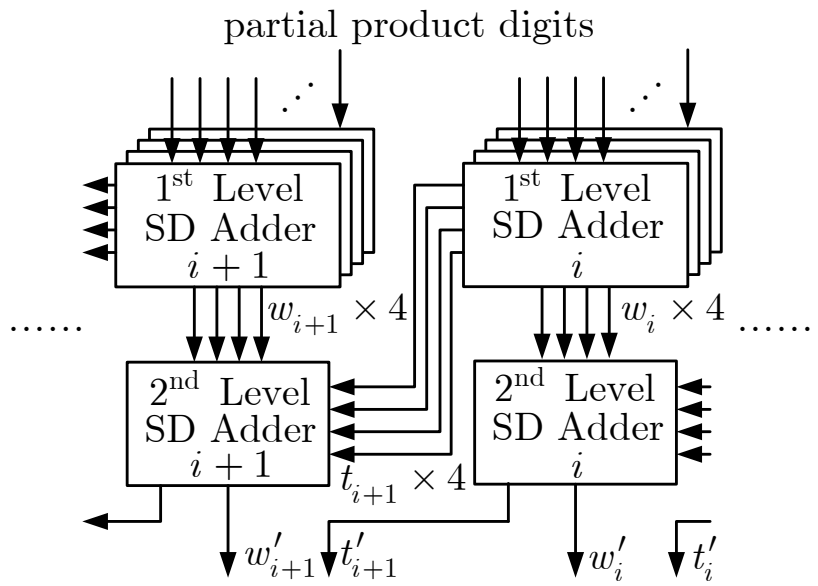


Figure 6.11: Top level architecture of the proposed partial product reduction unit

partial product.

6.2.3 SD-BCD Conversion

The partial products in a signed digit-set can be reduced to one SD result with the multi-operand SD adders. Unlike in other works, a digit-set converter is proposed to convert back the SD result into the conventional BCD encoding. Moreover, in such an SD-BCD conversion algorithm, a hybrid carry propagation network is discussed in detail.

SD-BCD Conversion Algorithm

In the proposed multiplier, the $2n$-digit result of the PPR is in digit-set $[-8, 7]$. If the digit is negative, a borrow (i.e., negative carry) occurs. To convert it back to the digit set $[0, 9]$ in BCD encoding, the negative digit is increased by 10, and the first non-zero digit with higher weight is reduced by one. All the continuous zeros in between the current negative digit and the first non-zero digit on its left side are converted to 9. An example is provided below:

$$(100\overline{4}802\overline{3})_{SD} = (09952017)_{BCD}$$

Thus to convert the SD result into BCD encoding, the negative digit (i.e., generates

the negative carry) and zero digit (i.e., propagates the negative carry) need to be detected. Furthermore, a carry propagation network and necessary logics are applied to determine and convert the SD digits into BCD encoding. The conversion algorithm is shown as follows:

Algorithm 6.2.3: SD to BCD Encoding Conversion

Data: SD number S.

Result: BCD number R.

1. Detect borrow generation bit (G_i) and propagation bit (P_i) for each digit of S.

$$
\begin{aligned}
G_i &= \begin{cases} 1 & \text{if } S_i < 0 \\ 0 & \text{otherwise,} \end{cases} \qquad P_i = \begin{cases} 1 & \text{if } S_i = 0 \\ 0 & \text{otherwise,} \end{cases} \\
G_{i:j} &= \begin{cases} G_i & \text{if } i = j \\ G_i + (P_i \cdot G_{i-1:j}) & \text{if } i > j, \end{cases} \\
P_{i:j} &= \begin{cases} P_i & \text{if } i = j \\ P_i \cdot P_{i-1:j} & \text{if } i > j. \end{cases}
\end{aligned} \tag{6.19}
$$

2. Compute the negative carry C_i of S ($C_0 = 0$).

$$C_{i+1} = G_{i:j} + (P_{i:j} \cdot C_j). \tag{6.20}$$

3. Convert the result S to BCD encoding.

$$
R_i = \begin{cases} S_i & \text{if } C_{i+1}C_i = 00 \\ S_i - 1 & \text{if } C_{i+1}C_i = 01 \\ S_i + 10 & \text{if } C_{i+1}C_i = 10 \\ S_i + 9 & \text{if } C_{i+1}C_i = 11 \end{cases} \tag{6.21}
$$

Hardware Implementation of the Converter

In Algorithm 6.2.3, the first step is to detect the negative and zero digits. Since in the proposed multiplier, the outputs of the PPR can be added into an SD number in the 4-bit two's complement encoding, the negative detection is simply a fourth-bit detection. To detect

a zero digit in two's complement encoding, all four bits are needed. Since inside the 4-bit CLA to sum up the final transfer digit t' and interim sum w', the results on different bits in a digit are available at different time, only one extra OR gate on critical path for the zero detection could be achieved by connecting three OR gates in cascade as shown in Fig. 6.12.

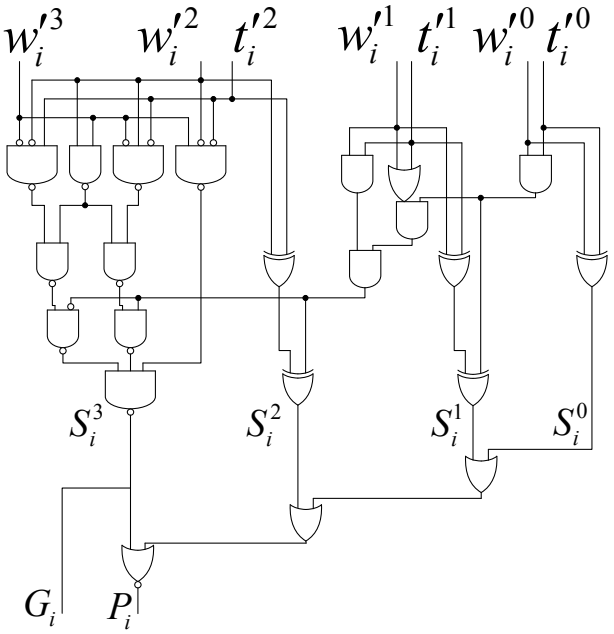


Figure 6.12: Simplified 4-bit CLA and G, P generation circuit

For the traditional method in the carry propagation step, a $\lceil \log(an) \rceil$-level prefix network is applied to quickly generate the final carry. The parameter a depends on the processing scope (e.g., in [75] the proposed quaternary tree unit works in bit level, thus a $\lceil \log(4n) \rceil$-level prefix tree is applied). No matter how many levels are in the prefix tree, the critical path passes through all levels of internal nodes. On the other hand, in the PPR stage, the longest path is potentially on the middle columns of the partial product array, and the rest of columns have shorter paths. It implies that the digits in final product which are close to the least and most significant digits are available earlier and can be processed before the digits in the middle part of the partial products array are ready. In section 5.3, a delay model on each digits of the final product is shown in Table 6.6. According to the estimated delay, the 32-digit SD result is divided into five groups which are $gp_0 = \{S_{11}, \ldots, S_0\}$, $gp_1 = \{S_{15}, \ldots, S_{12}\}$, $gp_2 = \{S_{17}, S_{16}\}$, $gp_3 = \{S_{21}, \ldots, S_{18}\}$, and $gp_4 = \{S_{31}, \ldots, S_{22}\}$. For each group, a small traditional carry propagation tree is applied. Thus the well optimized prefix tree circuit for

binary design could be reused. The carry propagation process is described in the following equations:

$$C_i = \begin{cases} 0 & \text{if } i = 0 \\ G_{i-1:0} & \text{if } 12 \geq i \geq 1 \\ G_{i-1:12} + P_{i-1:12} \cdot C_{12} & \text{if } 16 \geq i \geq 13 \\ G_{i-1:16} + P_{i-1:16} \cdot C_{16} & \text{if } 18 \geq i \geq 17 \\ G_{i-1:18} + P_{i-1:18} \cdot C_{18} & \text{if } 22 \geq i \geq 19 \\ G_{i-1:22} + P_{i-1:22} \cdot C_{22} & \text{if } 32 \geq i \geq 23 \end{cases} \tag{6.22}$$

where the C_i is the carry-in of the i digit, and note that the carry-in to the least significant digit is always zero.

In Fig. 6.13, a detailed structure of the proposed prefix network is shown. The white dot represents the logic to create the generation bit G_i and propagation bit P_i for each digit. The black dot represents the logic to create the group generation bits $G_{i:j}$ and group propagation bits $P_{i:j}$ described in equation (6.19). For the lower 12 digits, a Ladner-Fischer network is applied to minimize the number of levels and the area cost. Since the carry-in on the least significant digit is always zero, the carry-in to 13^{th} digit equals to $G_{11:0}$. For the digits from S_{12} to S_{15}, a two-level Ladner-Fischer network is used to create the group-carry-in generation and propagation signal, $G_{15:12}$ and $P_{15:12}$. To further calculate the carry, only an AND-OR gate is needed. For carry C_{18} and C_{17}, a 2-bit carry look-ahead structure is used. In higher 14 digits, the same technique as the one in the lower 16 digits is used. Note that to reduce the fanout of gates from low weight inputs through high weight outputs, a Han-Carlson network is applied to calculate the group-carry propagation and generation signals. In the 16-digit multiplication, at least 5-level of internal nodes should be on the critical path in a conventional method. In the proposed architecture, about 3-level of nodes are connected after the outputs of the middle columns in partial products array, and the level of nodes after the most significant columns are kept as 5. Although for less significant columns, the connected prefix tree nodes would be greater than five, the shorter delay on those columns could counterbalance the delay of the nodes in the prefix network. Note that the architecture

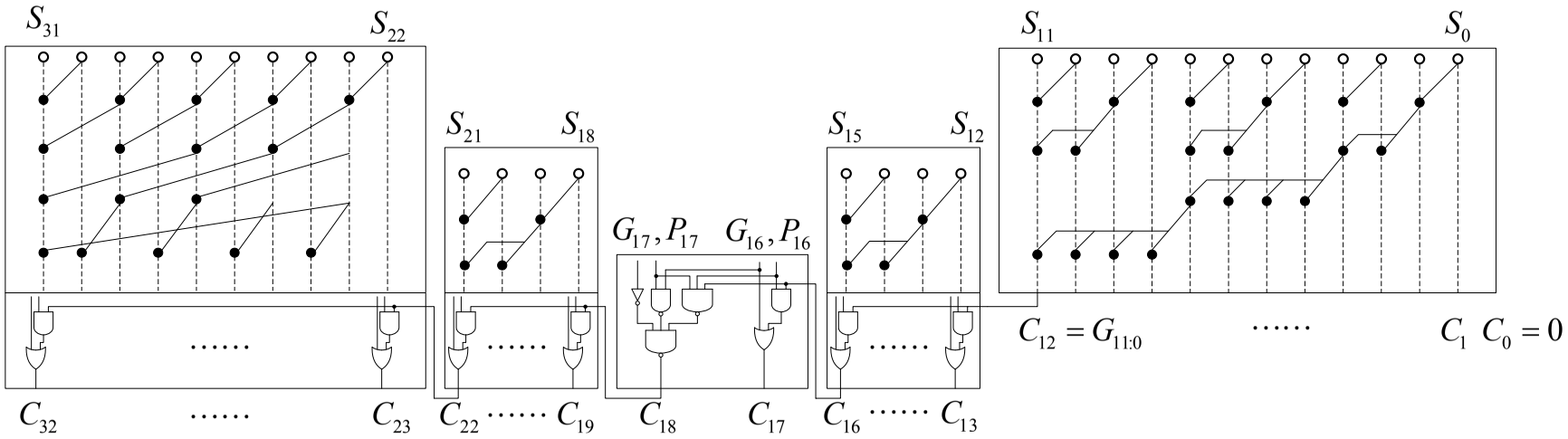


Figure 6.13: Proposed hybrid prefix network in the SD-BCD converter

of the hybrid prefix network highly depends on the structure of PPR. An improved structure would provide a better performance if the PPR structure is changed.

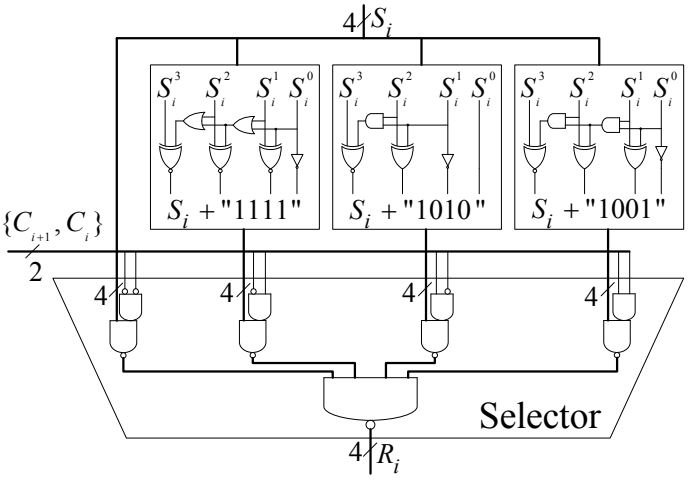


Figure 6.14: Final conditional constant adder

In the third step of the Algorithm 6.2.3, the SD result which is converted into BCD encoding by the conditional adder is selected by the carry signals of two neighbor digits in S. To convert the SD result into BCD encoding, since the correction signals (i.e., "0000", "1111", "1010", and "1001") for the four different carry-in cases are constant, the correction process could be designed as a conditional constant addition which could be comparatively simplified. In Fig. 6.14, the circuit of one digit conditional constant adder which consists of three constant adder and a combinational selector is shown.

6.3 Sequential Decimal Fixed-point Multiplication

In contrast to the parallel multiplication described in section 6.1, the sequential multiplication shows the advantage of the area efficiency. Thus, if the hardware cost is more sensitive, the sequential design could be applied in DFMA to achieve a new balance between cost and performance.

6.3.1 Signed Digit Partial Product Generation

The PPG of the proposed multiplier is based on the generation of easy-multiples of the multiplicand; thus, the required easy-multiples have to be determined. The representation of the multiplier Y plays a pivotal role in selecting the appropriate easy-multiples. Consequently, digit-set [-4,5] is selected to represent the multiplier Y in order to reduce the number of required easy-multiples and hence ameliorate the complexity of the PPG. This, however, calls for a recoder to convert the multiplier from digit-set [0,9] to [-4,5]. The recoder is implemented based on equation (6.23) where $y_i^c + y_i^s$ constitute the i^{th} digit of the multiplier Y in [-4,5] digit-set.

$$\begin{cases} y_i^s = y_i; y_{i+1}^c = 0 & \text{if } y_i \leq 5 \\ y_i^s = y_i - 10; y_{i+1}^c = 1 & \text{if } y_i > 5 \end{cases} \tag{6.23}$$

Given that the recoded multiplier needs to be ready iteratively (one digit per iteration), the carries in equation (6.23) (y_{i+1}^c) are stored in a latch and used in the next iteration as shown in Fig. 6.15.

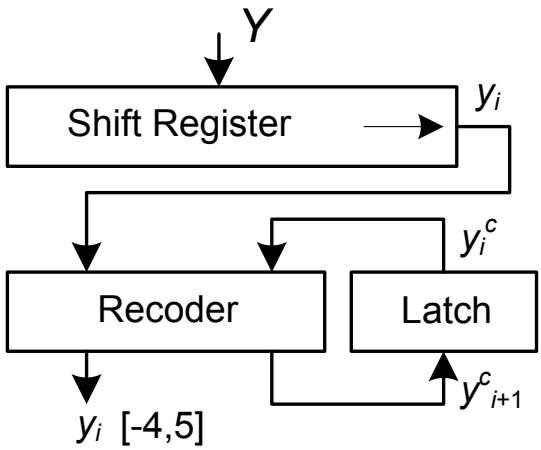


Figure 6.15: Recoding of the multiplier

Given the digit-set of the multiplier i.e., [-4,5], computing $X, \pm 2X$ and $\pm 4X$, as easy-multiples, is sufficient for generating a partial product as a sum of two decimal numbers (i.e., $P_i = U_i + V_i$). It should be noted that the addition $U_i + V_i$ is actually performed in the PPA step. Finally, a combinational logic is required to select the appropriate easy-multiples based

on the value of the multiplier's digit. Table 6.7 describes the selection rules for generating U_i and V_i.

Table 6.7: Selection of the easy-multiples

y_i	-4	-3	-2	-1	0	1	2	3	4	5
U_i	0	$1X$	0	$1X$	0	$1X$	0	$1X$	0	$1X$
V_i	$-4X$	$-4X$	$-2X$	$-2X$	0	0	$2X$	$2X$	$4X$	$4X$

With the intention of reducing the complexity of the PPG, the easy-multiples is generated in the encodings shown in Table 6.8; thereby simplifying the carry-free addition of the PPA step (see details on Section 6.3.2). Particularly, easy-multiple X is kept as BCD and $\pm 2X$,$\pm 4X$ are encoded into digit-set [-6,6] and represented as a signed-digit two's complement. In this approach, first, each digit (e.g., i^{th}) is divided into a transfer t_{i+1} and a sum w_i (as shown in Table 6.8); next $w_i + t_i$ generates the converted i^{th} digit.

Table 6.8: Conversion from BCD to the specific digit set

X_i	$2X$		$4X$	
	t_{i+1}	w_i	t_{i+1}	w_i
0	0	0	0	0
1	0	2	1	-6
2	1	-6	1	-2
3	1	-4	1	2
4	1	-2	2	-4
5	1	0	2	0
6	1	2	3	-6
7	1	4	3	-2
8	2	-4	3	2
9	2	-2	4	-4

According to Table 6.8, the generation of the easy-multiples $2X$ and $4X$ is performed via the logical expressions which are similar to equations (6.13) and (6.16).

Regarding the symmetric signed-digit 2's complement representation of $2X$ and $4X$, the $-2X$ and $-4X$ multiples are generated through a simple two's complement per digit. However, the two's complement operation is partially deferred until the PPA step. The overall architecture of the proposed PPG is illustrated in Fig. 6.16, where c_i is stored for the two's complement operation (per digit) performed in the PPA step.

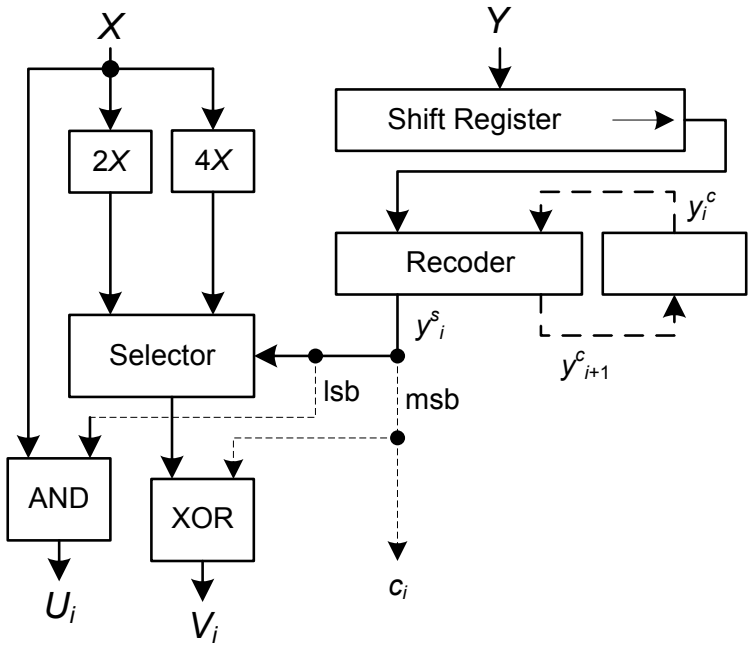


Figure 6.16: The proposed partial product generation

6.3.2 Partial Product Accumulation

Partial product accumulation is meant to add, properly, the generated partial product (i.e., $U_i + V_i + C_i$, according to Section 6.3.1) to the accumulated previous products $P[i]$. This is resembled in the recurrence equation 6.24, where C_i is the word-wide extension of c_i (1-bit c_i per digit).

$$P[i+1] = 0.1 \times P[i] + U_i + V_i + C_i \tag{6.24}$$

With the intention of reducing the latency of the PPA step, one can use a multi-operand redundant adder as to implement equation 6.24, where $P[i]$ and $P[i+1]$ are represented in a carry-save format. Figures 6.17 and 6.18 illustrates the dot-notation and the circuitry of the

multi-operand redundant addition used in the proposed PPA where (4:2) compressors with asterisk are the simplified one.

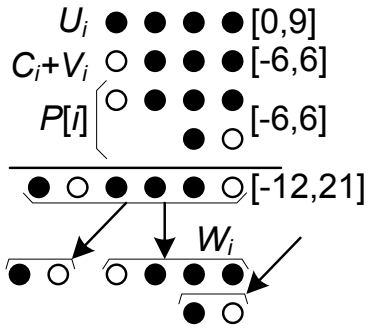


Figure 6.17: The dot-notation of partial product accumulation (digit-slice)

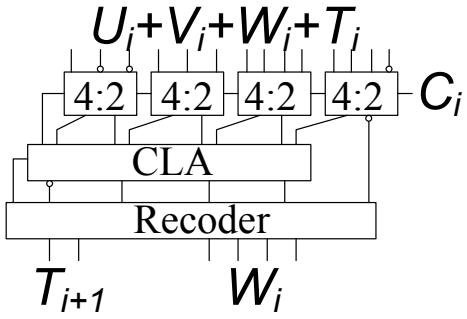


Figure 6.18: The circuitry of partial product accumulation (digit-slice)

Finally, after n iterations, the generated product $P[n+1]$ should be converted to the standard BCD format. This conversion is performed iteratively (a digit per iteration) for the lower part of the product P_L (based on Table 6.9), and in parallel (in two cycles) for the higher part P_H.

The parallel conversion consists of two main parts each of which with the following duties.

Part I: Preparing generate and propagate signals (i.e., g and p) to be used by the parallel prefix tree in Part II. Moreover, A 4-bit carry-look-ahead adder (CLA) is responsible to generate the appropriate digit value.

Part II: A parallel prefix tree computes the carry of each digit position; then a combinational logic (based on Table 6.9) produces the final converted product.

Fig.6.19 depicts the architecture of the proposed parallel conversion.

Table 6.9: Iterative Conversion

Digit in	Carry in	Digit out	Carry out	Digit in	Carry in	Digit out	Carry out
4	0	4	0	4	-1	3	0
3	0	3	0	3	-1	2	0
2	0	2	0	2	-1	1	0
1	0	1	0	1	-1	0	0
0	0	0	0	0	-1	9	-1
-1	0	9	-1	-1	-1	8	-1
-2	0	8	-1	-2	-1	7	-1
-3	0	7	-1	-3	-1	6	-1
-4	0	6	-1	-4	-1	5	-1
-5	0	5	-1	-5	-1	4	-1

In a nutshell, the whole architecture of the proposed sequential multiplier (including the PPG and PPA) is shown in Fig. 6.20, where concatenating P_L and P_H produces the final product.

6.4 Decimal Floating-point FMA

The top level architecture of the proposed DFMA is shown in Fig. 6.21. After the operand decoder, two significands CX and CY are fed into the multiplier array. Meanwhile, the alignment shifting operation is done in parallel with the multiplication. Subsequently, a decimal carry free adder sums up the redundant product and the addend which is inverted according to the effective operation. With the internal redundant number system, the carry propagation in the final digit-set converter of the multiplier array is eliminated. Moreover, a simpler leading zero decision algorithm can be applied on the carry free result. The propagation in the decimal addition is therefore eliminated before the rounding position is obtained. In the post-alignment shifter, the digits which exceed the required precision are moved out, and the $(n+1)$-digit result is sent to the final rounder. In the final rounding unit, the absolute value conversion, the digit-set conversion, and the rounding operation are performed at the same

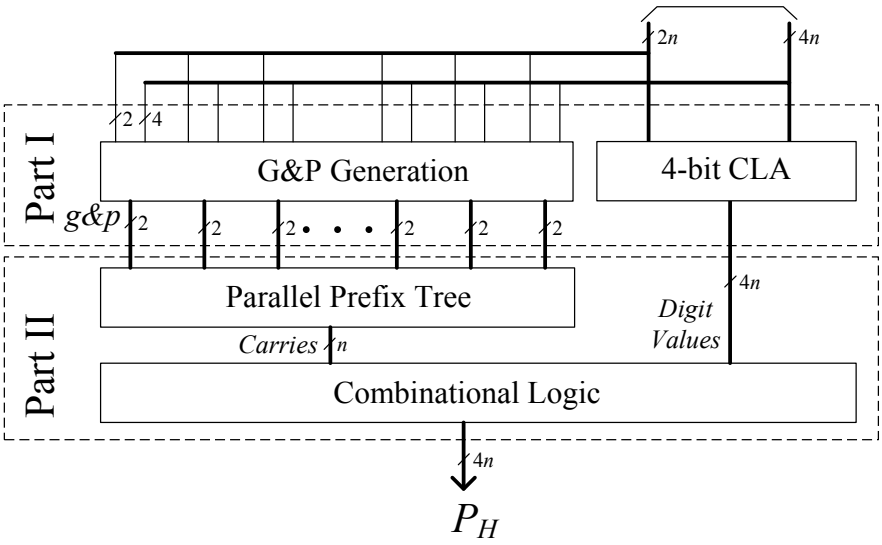


Figure 6.19: The proposed parallel conversion

time. A detailed structure of the proposed DFMA for Decimal64 format is given in Fig. 6.22. The characteristics and process of the proposed FMA computation are described as follows.

1. In the multiplier array, the multiplication structure which has been proposed in [96] is exploited. Since the redundant intermediate product is further used in following units, the final digit-set conversion proposed in [96] which involves a carry propagation is not needed anymore. A $(2n+1)$-digit product on digit-set $[-8, 7]$ is therefore retained.

2. In the meantime, the pre-alignment of the addend which is performed in parallel with the multiplier array no longer exists on the critical path. First, the exponent difference between the product and addend is obtained by binary prefix tree adders. Subsequently, the addend is shifted to right or left depending on the sign and absolute value of the exponent difference. Since the product consists of $2n+1$ redundant digits, to guarantee the required precision and rounding information of the final result, the shifting range of alignment is extended to $4n+2$ digits. After all, the XOR gates are applied to negate the shifted addend for effective subtraction.

3. In the addition module, the nonspeculative decimal adder proposed in [97] is modified to add two operands in $[-8, 7]$ and $[-9, 9]$ and create a result in $[-8, 7]$. Since the number of shifting digits in post-alignment is detected based on the redundant result, the carry propagation is not necessary anymore.

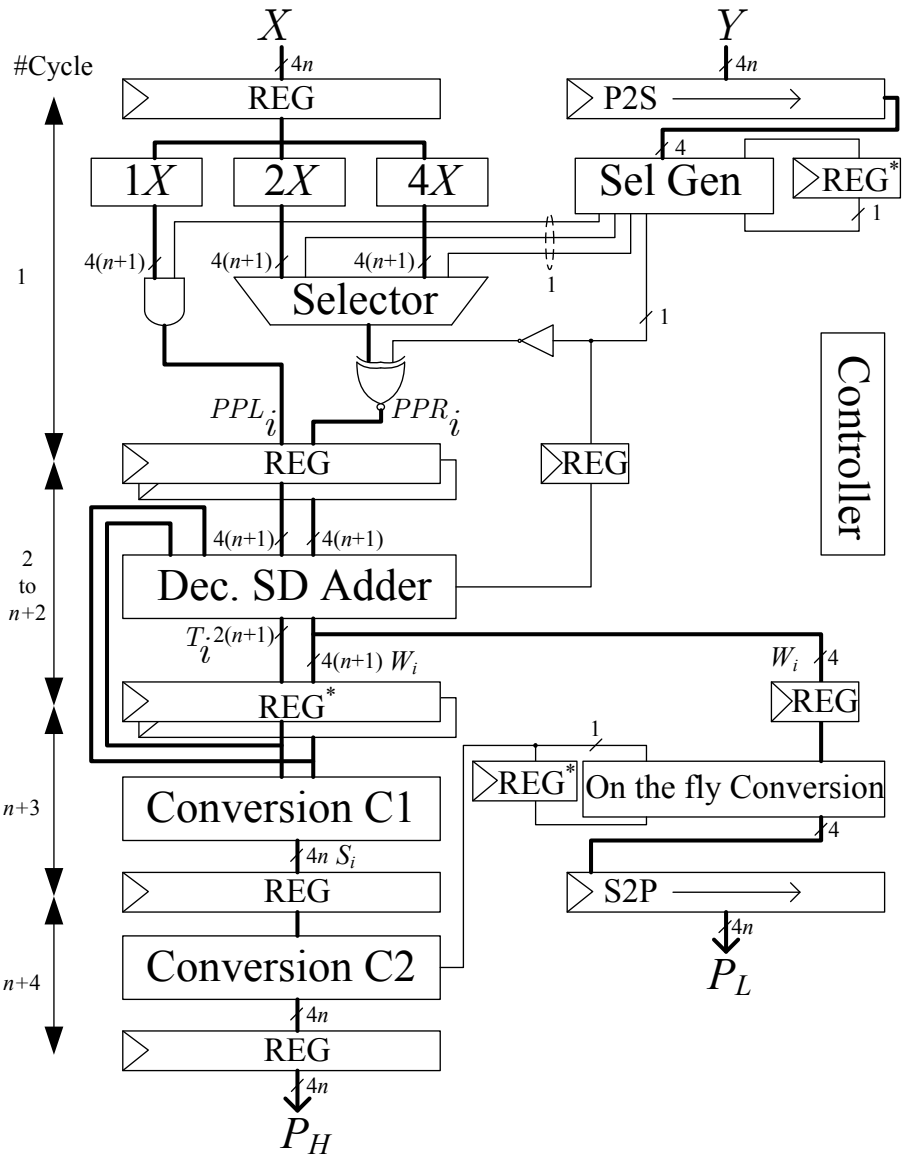


Figure 6.20: The proposed sequential decimal multiplier

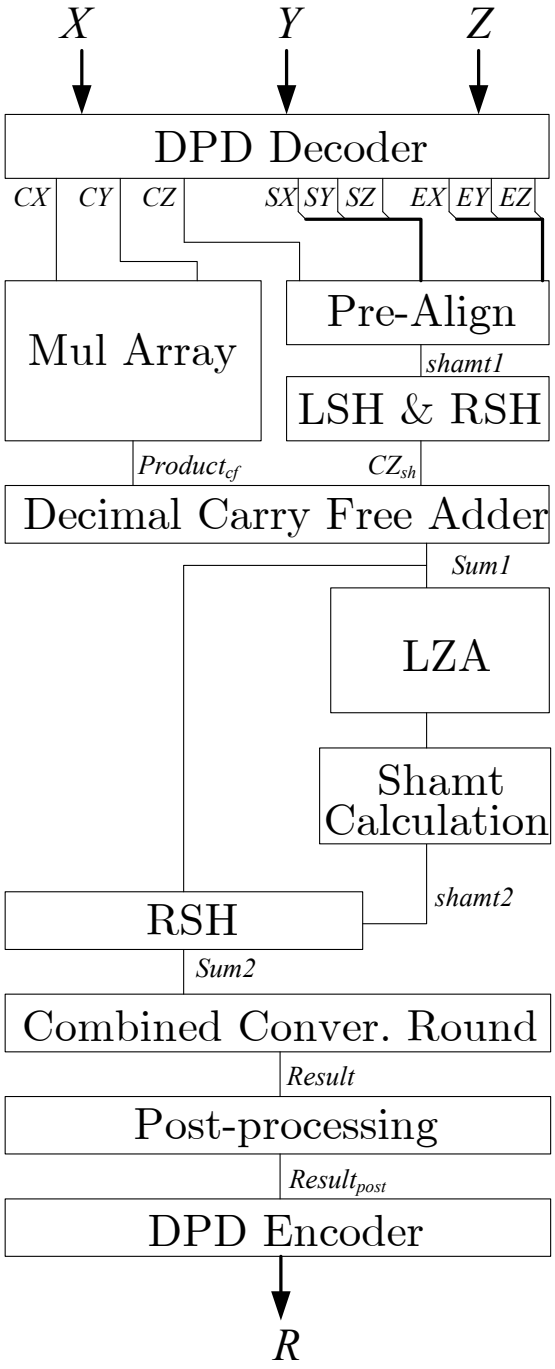


Figure 6.21: Proposed architecture

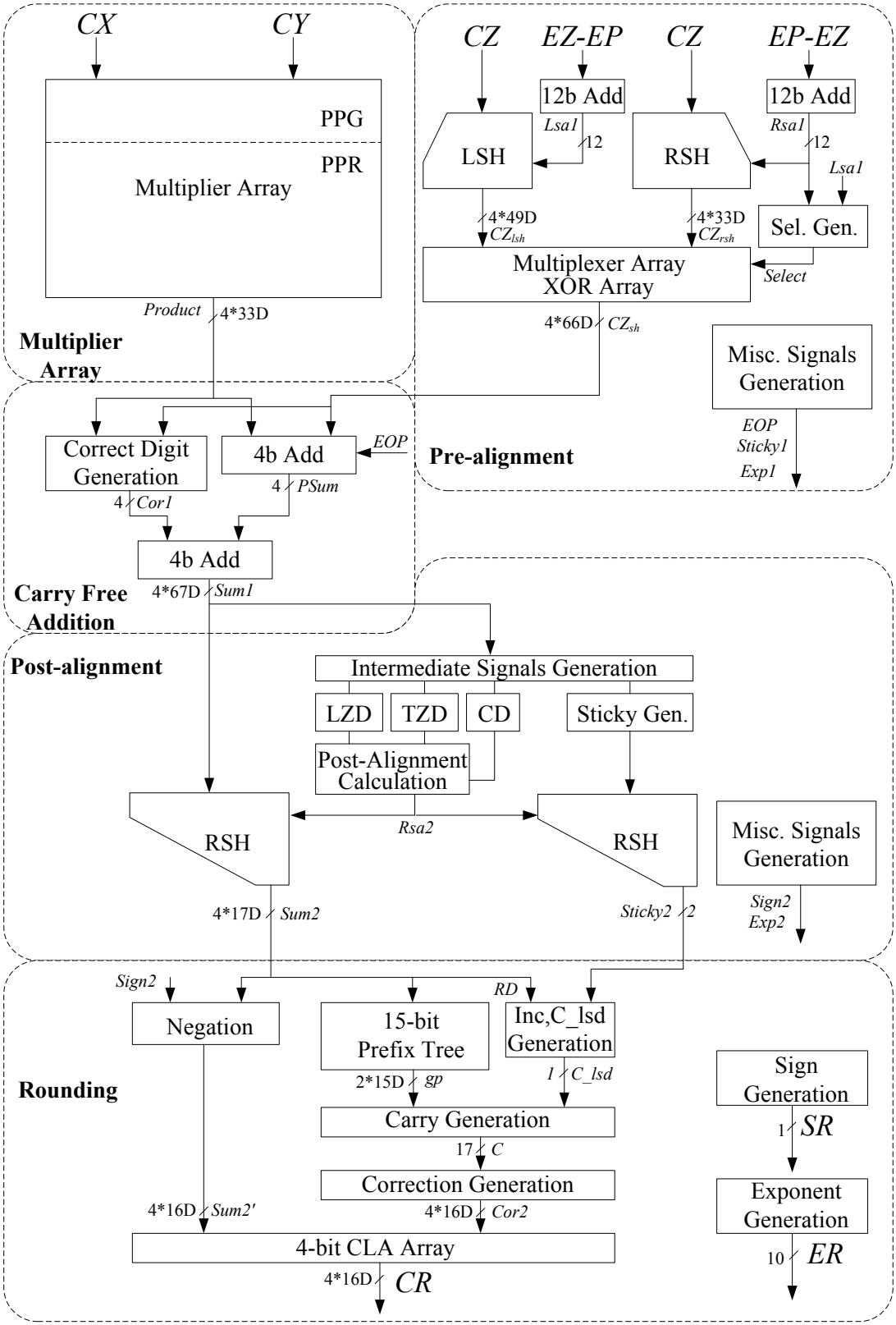


Figure 6.22: Details of structure

Input:

$SX = 0 \;\; CX = 0963625485443960 \;\; EX = 18$
$SY = 0 \;\; CY = 7828178241591672 \;\; EY = -1$
$SZ = 1 \;\; CZ = 9999888877665432 \;\; EZ = 31$

Calculation:

1, Multiplication:

$Product =$
$012463432142204420125102041301120$

2, Pre-Alignment:

$EP = 17 \;\; EZ = 31$

$Lsa1 = 14$ (active) $Rsa1 = -14$

$CZsh = 11...11aaaa99998877654311...11$

$EOP = 1 \;\; Sticky1 = 00$ (zero) $\;\; Exp1 = 1$

3, Addition:

$$\begin{array}{r} 012463432142204420125102041301120 \\ +00...00999998887766543200...00 \\ \hline 00...00134565632304311231251020413011200...00 \end{array}$$

4, Post-Alignment:

$Rsa2 = 31$

$Sum2 = 345656323043112 31$

$Sign2 = 0 \;\; Sticky2 = 01$ (positive) $\;\; Exp2 = 32$

5, Rounding:

$RD = 1$

$RD_{inc} = 0 \qquad C_{lsd} = 1$

$C = 11111100100010110$

Output:

$SR = 0 \;\; CR = 6543443170429077 \;\; ER = 32$

Figure 6.23: Details of calculation

4. The post-alignment unit shifts the intermediate result after addition to achieve the preferred exponent and guarantee the required precision. Since the digits $\pm(radix - 1)$ (i.e. ± 9 in this radix-10 system) are not used, the long-term cancelation does not exist. Consequently the leading zero detection of the proposed digit-set is simple. However, the sticky digit is harder to be examined than it in other architectures, since the moved out digits may represent a negative value. A method to obtain the sticky bits, which represent a signed sticky digit, with almost the same delay as the post-alignment shifting is introduced.

5. The result from the post-alignment shifter can be positive or negative, and the digit-set is redundant. Therefore, the absolute value of the non-redundant result has to be obtained before performing the rounding decision in a straightforward method. More than one carry propagation might be involved in this process. In this work, an algorithm which negates, converts, and rounds the redundant intermediate result into the BCD format with one long-term carry propagation and constant delay logics is described.

To illustrate the principle of the computation in the proposed FMA, an example is given in Fig. 6.23. Once all the operands are ready, $CX \times CY$ is firstly performed in the multiplier array, and the redundant $Product$ is obtained. In the meantime, the difference on exponent is calculated by two adders. If $EZ - (EA + EB)$ is positive, the left shifting is active, otherwise the right shifting is active and selected. Since the effective operation EOP is subtractive, the shifted addend is further negated by the XOR gates after the multiplexer array to achieve two's complement on every digit. The missing increment one for every digit is therefore sent to the adder by EOP. Note that the negated addend digit $\overline{a}$ actually means $\overline{10}$ (i.e $\overline{9}$ if includes the increment 1 in EOP). However, in hardware only 4 bits are used for each digit, and $\overline{a}$ is represented as "0110" without the fifth bit. After alignment, the intermediate exponent $Exp1 = 1$ is calculated by adding or subtracting the right or left shift amount on EZ and subtracting 16 for moving decimal point. Subsequently, two operands from alignment unit and multiplier are added, and a redundant result is obtained in $Sum1$. At the same time, the right shift amount for the post-alignment is calculated. Since the result $Sum1$ has more than 16 significant digits which is larger than the required precision, only a 16-digit significand, a 1-digit rounding digit and a 2-bit sticky digit are retained in $Sum2$ after the shifter. Due to the post-alignment, the intermediate exponent $Exp2$ is updated

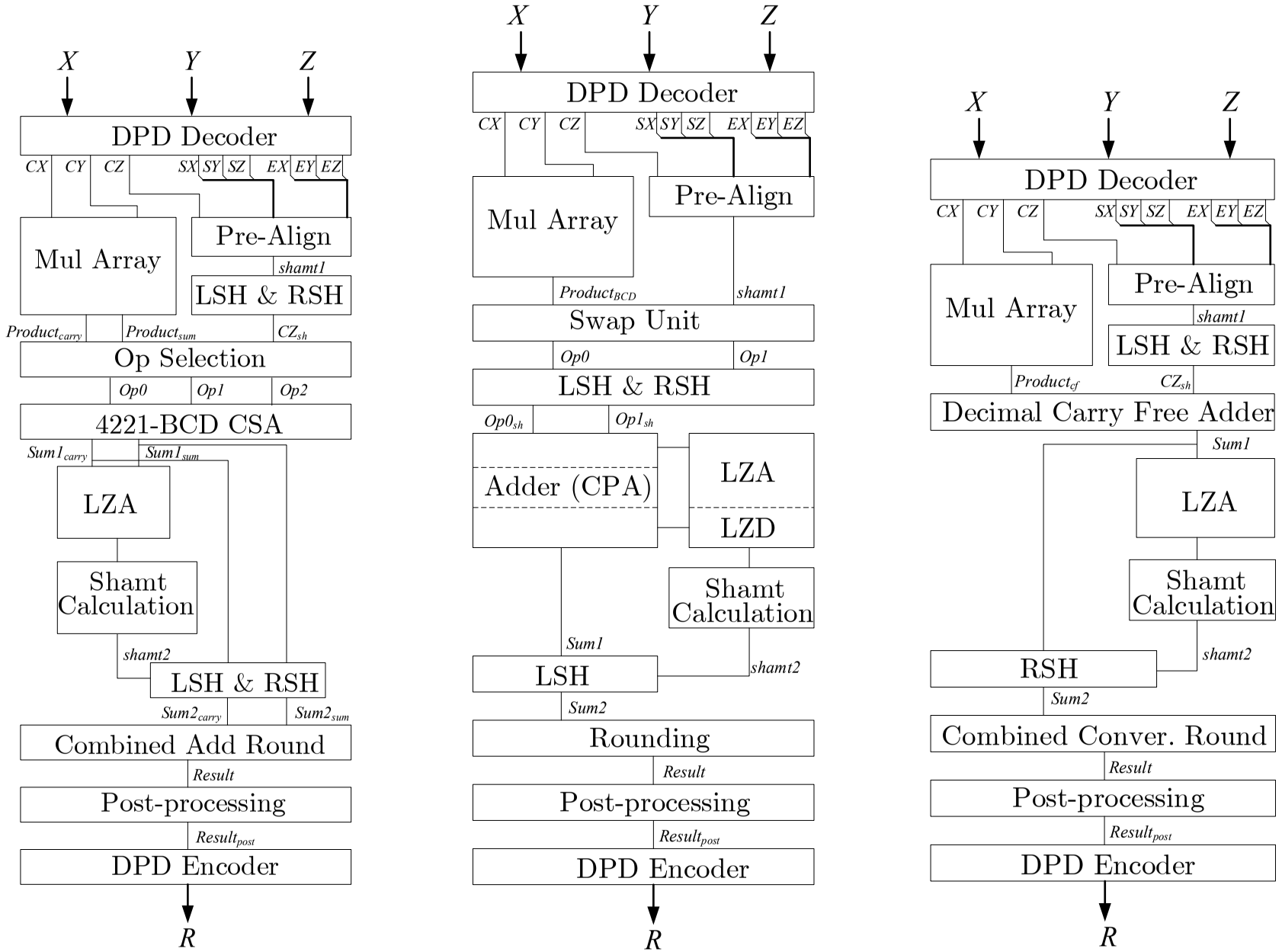
by adding $Exp1$ with the $Rsa2$. In the final rounding unit, the rounding digit $\overline{1}$ and the positive $Sticky2$ cause a zero increment in the least significant digit (LSD) of the significand. Consequently, the negative carry C for converting the digit-set with the consideration of the rounding increment is obtained. Finally, the absolute value of the rounded result in the conventional digit-set is achieved by adding the correction value of 10's complement decided by C to the negated $Sum2$ obtained by the XOR gate.

To illustrate the differences between our proposed design and other previous designs on the top level architecture. The simplified architectures of three designs are given in Fig. 6.24.

The core architecture of the proposed DFMA is partitioned into five sub-modules, which are multiplication, pre-alignment, addition, post-alignment, and rounding unit. Since the major works of the multiplier and adder have already been proposed in previous sections, in this section, these two basic computations are simplified by two models which create $(n+m+1)$-digit result for n-digit$\times m$-digit redundant multiplication and $(k+1)$-digit result for n-digit+m-digit redundant addition, where $k = max\{n, m\}$. The rest components of the design, which include addend alignment, decision of the rounding position, and rounding the redundant result with direct conversion, are described in details including algorithms and hardware structures in this section.

6.4.1 Pre-Alignment

In the proposed DFMA, the pre-alignment block is in parallel with the multiplier array. Therefore the pre-alignment shifting is only processed on the addend CZ, and the product is kept in its position while CZ is shifting. In principle, the pre-alignment algorithm shifts the operand to make the following addition upon two operands which have the same exponent or guarantees the result of the addition equals to what it is supposed to be once the shifting range is too large. Since the decimal operand is not normalized, the number of significant digits of the non-zero product obtained after the multiplier can be from 1 to $2n + 1$. Thus, to guarantee the precision and the correct rounding digit, the necessary shifting width of the addend can be $4n + 2$ digits, which are decided in two extreme cases. In the first case, the least significant digit of the addend is shifted $3n$ digits to left, and n digits precision are therefore guaranteed on the left of the product. Note that if more digits are required to be



(a) Architecture proposed in [102] **(b)** Architecture proposed in [95] **(c)** Proposed architecture

Figure 6.24: Decimal floating-point fused multiply-add architectures

shifted to left, the first digit lower than the most 16 significant digits are implied to be zero. In the second case, the most significant digit of the addend is shifted $2n$ digits to right. If the exponent difference between product and addend is larger than the necessary range of alignment shifting, the extra digits other than the necessary shifting range do not affect the correctness of the final result. Additionally, the decimal point is shifted n digits to right to guarantee that the final result is an integer.

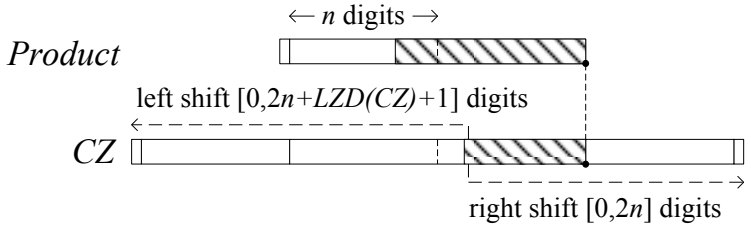


Figure 6.25: Left and right shifting range of the pre-alignment.

If the product is zero, the result should be numerically equal to the addend. However the preferred exponent which is defined in the standard should be achieved. If $EP \geq EC$, the absolute value of the result is exactly equal to the addend which has a less exponent. If $EP < EC$, the significand of Z has to be shifted to left to reduce the exponent of the addend. The final result will be normalized to get the possible maximum number of significant digits or the possible minimum exponent which is close to EP. If the addend is zero, the precision and preferred exponent of the final result will be guaranteed by the digits in product and the post-alignment algorithm regardless of the shifting direction and shifting digits of the zero addend. In the first case, the final result can be directly figured out, and in the latter two cases, the computing process follows the pre-alignment rule analyzed in the previous paragraph. The post-alignment algorithm to guarantee the rounding position and the preferred exponent is described in next section. The shifting range to align two operands are shown in Fig. 6.25.

In the proposed architecture, the pre-alignment algorithm is divided into four cases, which are 1) left shifting with overflow, 2) left shifting without overflow, 3) right shifting without overflow, and 4) right shifting with overflow. In the case 1), the exponent of the added is larger than the exponent of the product, and the difference is larger than the maximum left shifting amount. In this case, the addend is shifted $2n+1+\mathrm{LZD}(CZ)$ digits to left. The OV signal is therefore set to indicate a left shifting overflow occurs. In the case 2), the exponent

Algorithm 6.4.1: Pre-alignment algorithm

Data: EX, EY, EZ, CZ.

Result: Left and right shift amount $Lsa1, Rsa1$.

Overflow signal OV.

if $(2n + 1 + LZD(CZ) < EZ - EP)$ **then**

  $Lsa1 = 2n + 1 + LZD(CZ)$;

  $OV =$ "10";

else if $(0 \leq EZ - EP \leq 2n + 1 + LZD(CZ))$ **then**

  $Lsa1 = EZ - EP$;

  $OV =$ "00";

else if $(0 < EP - EZ \leq 2n)$ **then**

  $Rsa1 = EP - EZ$;

  $OV =$ "00";

else if $(2n < EP - EZ)$ **then**

  $Rsa1 = 2n$;

  $OV =$ "01";

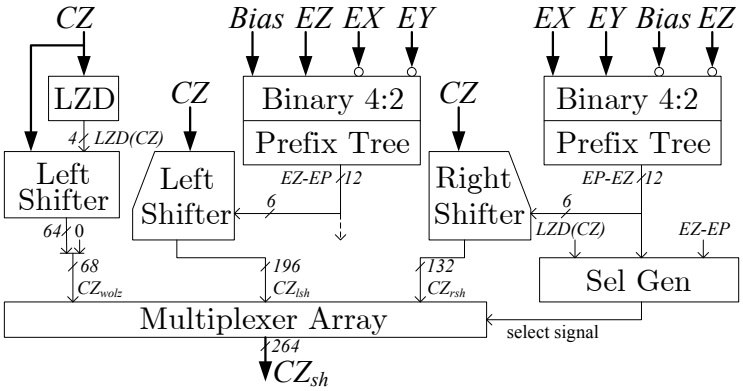
end

where $LZD()$ means the leading zero detection function, and $EP = EX + EY$;

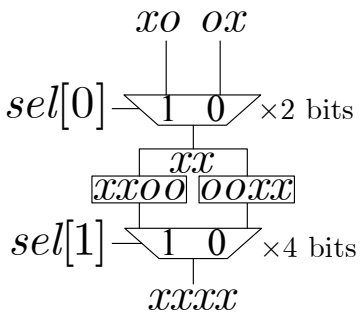
of the added is larger than the exponent of the product, but the difference is smaller than the maximum left shift amount. Hence, the difference on the exponent is set to the left shifting amount ($Lsa1$). In the latter two cases, the exponent of the product is larger than the exponent of the addend, thus, right shifting is performed. The mathematical description of the pre-alignment algorithm is given in Algorithm 6.4.1.

The hardware implementation of the proposed pre-alignment unit is depicted in Fig. 6.26(a). The left and right shifting amount $Lsa1$ and $Rsa1$ are calculated simultaneously by two binary prefix tree adders. Since the maximum shift amount to right or left are constant, only lower bits of the results from two carry propagating adders are fed into the shifters. To reduce the timing delay, the number of leading zeros in the addend LZD(CZ) is not considered to obtain the $Lsa1$ before the left shifter. Instead, the addend without the leading zeros

(CZ_{wolz}) is created by a separate shifter, and selected out as the most significant digits if left overflow occurs (i.e. OV = "10"). To select the correct shifted addend from the shifters, a selection signal generator and a multiplexors array are applied aside and after the shifters. The selection signal generator decides the real shifting direction, and selects out the corrected shifted addend from the results of the three shifters. The selection signal can be easily figured out by the sign of $EZ - EP$ and the overflow signal OV.



(a) Hardware implementation of the pre-alignment



(b) Hardware implementation of a simplified left shifter in the pre-alignment

Figure 6.26: Architecture of the pre-alignment

Since the widths of the input and output of the two shifters in Fig. 6.26(a) are different, it is possible to elaborate the structure and reduce the hardware cost of the shifter. In Fig. 6.26(b), a simplified model of the proposed left shifter is shown to shift one bit input x to left. Since the lower bits of result are obtained earlier than the higher bits in the binary adder, the multiplexors for shifting less digits are placed on the top of the shifter. The right shifter has a symmetrical structure. In contrast to the original shifter which has the same width on both input and output, the refined shifter saves about 37% of the multiplexors.

Table 6.10: Selection algorithm of the shifted addend

$\{Sign1$[a]$;OV\}$	$CZ_{sh}.S3$[b]	$CZ_{sh}.S2$	$CZ_{sh}.S1$	$CZ_{sh}.S0$
000	$CZ_{lsh}.S3$	$CZ_{lsh}.S2$	$CZ_{lsh}.S1$	0
001	x	x	x	x
010	CZ_{wolz}	0	0	0
011	x	x	x	x
100	0	0	$CZ_{rsh}.S1$	$CZ_{rsh}.S0$
101	0	0	0	0
110	x	x	x	x
111	x	x	x	x

[a] $Sign1$=The sign of $(EZ - EP)$

[b] $S.S3 = S\{65 : 49\}; S.S2 = S\{48 : 33\}; S.S1 = S\{32 : 17\}; S.S0 = S\{16 : 0\}$

After the pre-alignment, the layout of the aligned addend and product and the shifted decimal point are shown in Fig. 6.27. The exponent after pre-alignment $Exp1$ (i.e. the exponent of the result after the carry free adder) is adjusted by subtracting 16. There are some signals which are used out of the critical path can be calculated in pre-alignment unit as well. The equations of these signals are given below:

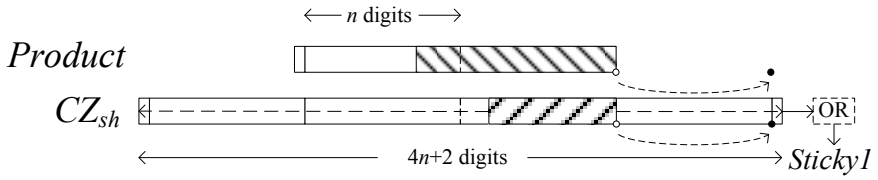


Figure 6.27: Layout of the aligned product and addend

$$EOP = SX \oplus SY \oplus SZ \tag{6.25}$$

```
if (OV = "10")
    Exp1 = EC − LZD(CZ) − 49;
else                                                    (6.26)
    Exp1 = EP − 16;
endif
```

```
    if (RSHOR(CZ) = 0)
        Sticky1 = "00";
    else
        if(EOP = 1)
            Sticky1 = "11";                             (6.27)
        else
            Sticky1 = "01";
        endif
    endif
```

where RSHOR() means the bit-by-bit OR of all right shifted digits out of the CZ_{rsh}.

In equation (6.26), if left shifting overflow occurs (i.e. OV = "10"), the real left shifting amount is only $2n + 1 + \text{LZD}(CZ)$ digits. In this case, $Exp1$ is adjusted by the real left shifting amount and the shifting of the decimal point.

6.4.2 Post-Alignment and Sticky Bits Generation

Post-Alignment Shifting Amount Decision

The result from the adder may have a large number of significant digits which exceed the required precision. Thus a post-alignment unit is applied to decide a proper exponent and truncate the significand to fit the required precision. Moreover, if the result is inexact, enough information has to be kept to decide the increment to the least significant digit in the following rounding unit. In the decimal floating-point standard, both of the input operands and output result are not normalized. Therefore, the post-alignment processing in the decimal floating-point is more complicated than the normalization processing in the

Algorithm 6.4.2: Analysis of the shifting direction and range to achieve preferred exponent

if $(EP \geq EC)$ **then**

`/* right shift addend */`

$Exp1 = EP - 16;$

$Expp = EC;$

$DIFF_{pre} = EC - EP + 16;$

$DIFF_{pre} = -DIFF_{abs} + 16;$

$DIFF_{pre} \leq 16;$

else

`/* left shift addend */`

if $(OV = 0)$ **then**

$Exp1 = EP - 16;$

$Expp = EP;$

$DIFF_{pre} = EP - EP + 16;$

$DIFF_{pre} = 16;$

else

$Exp1 = EC - LZD(CZ) - 49;$

$Expp = EP;$

$DIFF_{pre} = EP - EC + LZD(CZ) + 49;$

$DIFF_{pre} = -DIFF_{abs} + LZD(CZ) + 49;$

$DIFF_{pre} \leq 16;$

end

end

$DIFF_{pre} = Expp - Exp1;$

$DIFF_{abs} = ABS(EP - EC);$

$Expp = MAX(-398, MIN(EP, EC));$

$ABS()$ means the absolute value function.

binary floating-point. The most difficult problem is to decide if the preferred exponent can be achieved or not. In a conventional method, the leading zero anticipation algorithm needs to detect the place of the most significant one, and detect the decimal cancelation in parallel. For example, if the significand is "$1\overline{99...9}234...$", the sequence of $\overline{9}$ after the leading 1 will have to be canceled in the final result, and the width of the significand is reduced accordingly. In the proposed number system, the digit 9 or $\overline{9}$ are not existing. Therefore, the cancelation only causes one-digit error on the basis of the conventional leading one detection algorithm. For example, "$1\overline{8}...123...$" will be converted to "02...123...".

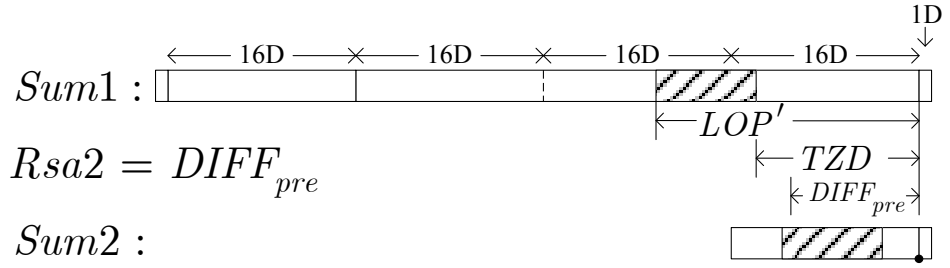
Algorithm 6.4.3: Post-alignment algorithm

$LOP' = LOP + 1^{\overline{needcorrect}}$;
if $(LOP' - TZD \leq 16$ *and* $TZD - DIFF_{pre} \geq 0)$ **then**
  if $(LOP' - DIFF_{pre} \leq 16)$ **then**
    `/* case 1 */`
    $Rsa2 = DIFF_{pre}$;
  else if $(LOP' - DIFF_{pre} > 16)$ **then**
    `/* case 2 */`
    $Rsa2 = LOP' - 16$;
  end
else if $(LOP' - TZD \leq 16$ *and* $TZD - DIFF_{pre} < 0)$ **then**
  `/* case 3 */`
  $Rsa2 = TZD$;
else if $(LOP' - TZD > 16$ *or* $DIFF_{pre} < 0)$ **then**
  `/* case 4 or 5 */`
  $Rsa2 = LOP' - 16$;
end

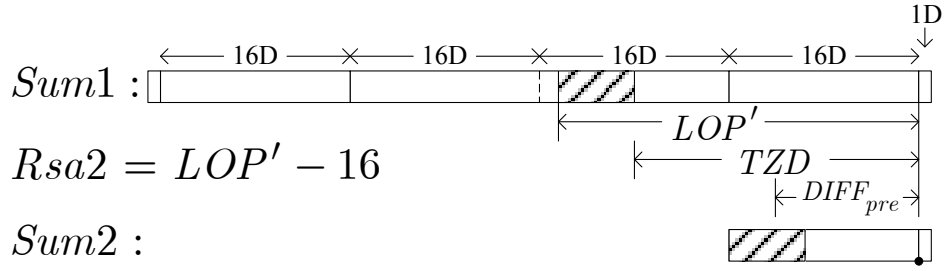
To figure out the cases of the shifting in post-alignment, the exponent of the temporary result of the addition and the preferred exponent are analyzed in Algorithm 6.4.2. First of all, two parameters are created. $DIFF_{pre}$ is defined as the difference between the preferred exponent and the temporary exponent. If $DIFF_{pre}$ is larger than zero, it means the necessary

right shifting digits to achieve the preferred exponent. $DIFF_{abs}$ is defined as the absolute value of the difference between the exponents of the product and the addend. In the first case, the addend is right shifted and $DIFF_{pre} \leq 16$. If $0 \leq DIFF_{pre} \leq 16$, the number of right shifting digits in post-alignment depends on the significant digits between leading and trailing zeros. If $DIFF_{pre} < 0$, the temporary result has to be shifted to left to achieve the preferred exponent. But after moving the decimal point, significant digits are always enough to guarantee the required precision. Therefore, in this case the preferred cannot be achieved, and the number of right shifting digits depends on the most significant non-zero digit. In the second case, where $DIFF_{pre} = 16$, it is similar to the first case. In the third case, $LZD(CZ)+$ 49 means the possible maximum left shifting digits in the proposed pre-alignment. Since left overflow happens in this case, $DIFF_{abs}$ is always larger than $LZD(CZ) + 33$ according to Algorithm 6.4.1. Thus $DIFF_{pre}$ is less than 16, and the analysis of shifting is similar to the first case.

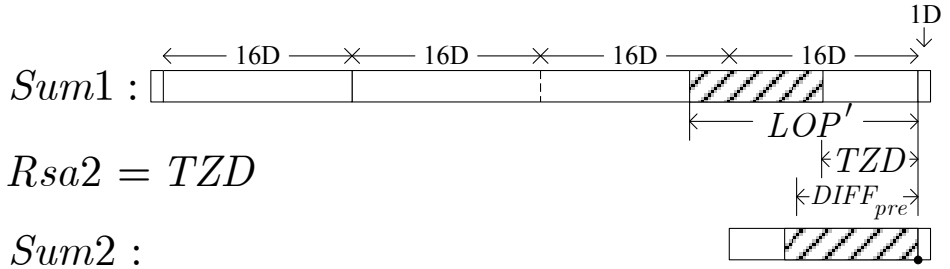
In Figures 6.28(a-e), the post-alignment algorithm is illustrated into five cases, and the mathematical description is given in Algorithm 6.4.3. In the first two cases, the result can be exactly represented in 16 digits. Therefore, the preferred exponent might be reached. In case 1, the number of significant digits, which is equal to the difference between the leading one position (LOP) and the number of trailing zeros (TZD), is less than 16 digits, and the difference between the temporary exponent and the expecting exponent ($DIFF_{pre}$) is smaller than the number of trailing zeros in the redundant result. Note that the LOP' means the real position of leading non-zero digit, which is obtained by correcting LOP. Additionally, after shifting the result of the addition to right by $DIFF_{pre}$ digits, the exponent reaches the preferred one, and the result still keeps all the significant digits. In case 2, after shifting $DIFF_{pre}$, not all the significant digits can be restored, and more digits need to be shifted. In this case, only the most significant 16 digits are retained. In case 3, the $DIFF_{pre}$ is greater than the number of trailing zeros. Thus, the maximum right shifting amount only can be TZD to keep all the significant digits. The preferred exponent cannot be reached in this case, and the adjusted exponent ($Exp2$) is less than and closest to the preferred exponent. In the previous three cases, the result is exact. In case 4, the significant digits are larger than 16 digits. Hence, the preferred exponent cannot be reached and the final result is



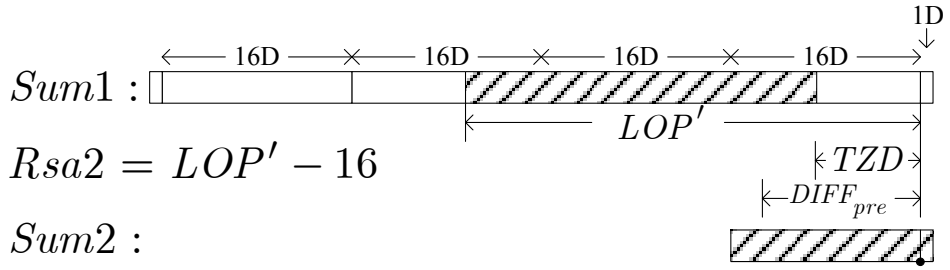
(a) Case 1



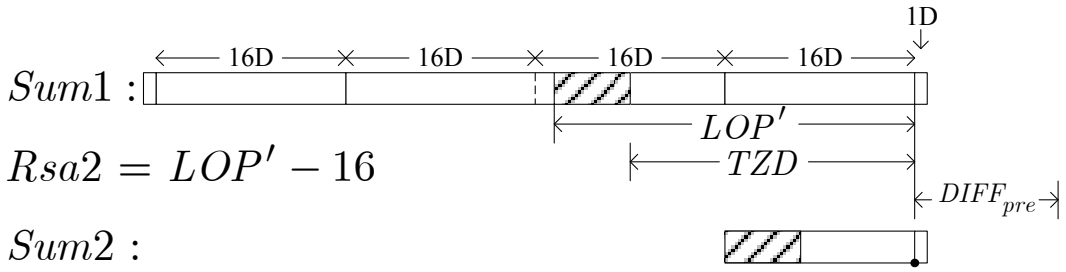
(b) Case 2



(c) Case 3



(d) Case 4



(e) Case 5

Figure 6.28: Post-alignment shift amount decision

inexact. In case 5, if the preferred exponent is smaller than the temporary exponent, the preferred exponent cannot be reached, since the LOP has 16 digits at minimum. Therefore, left shifting is not possible in the proposed architecture. Consequently, any digits out of the required precision are moved out.

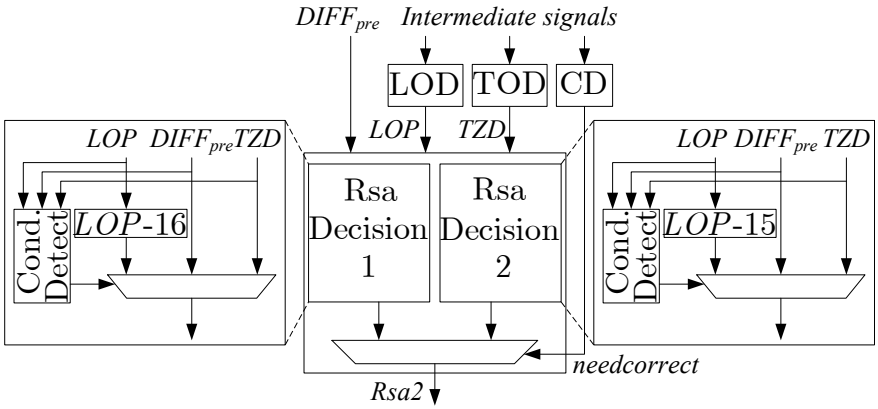


Figure 6.29: Detailed structure of the post-alignment shift amount calculation

The hardware implementation is shown in Fig. 6.29. The data path is divided into two branches, which are selected by detecting the one digit error of the leading one position in $Sum1$. The left path covers the five cases if the possible one-digit error doesn't exist. Otherwise, the right path is selected. There are three blocks upon the shifting amount decision unit. The leading zero detector (LOD), which generates the position of the leading non-zero digit minus one ($LOP' - 1$), is similar to the detector applied in binary designs. The trailing zero detector (TZD) similarly creates the number of the trailing zeros. The correction detector (CD) is introduced in the next section.

Leading One Position Correction

To detect the possible one digit error on the leading one position in $Sum1$, two cases have to be recognized. As shown in Table 6.11, if the pattern of the $Sum1$ is $z^k 1 z^l n(x)$ or $z^k \overline{1} z^l p(x)$, the leading one or leading minus one will be converted to zero in the rounding unit. The position of the leading non-zero digit is therefore reduced by one. To detect these two patterns, a binary tree structure is applied for both positive and negative $Sum1$. The principle of the correction detector is similar to the algorithm described in [98]. The difference is that the result of the radix-10 signed-digit subtraction is more complicated than the one in the

radix-2 signed-digit subtraction proposed in [98]. In Table 6.12 and equations (6.28 and 6.29), the algorithm and logic of the basic node on the detection tree for both positive and negative $Sum1$ are given. Similar to the logic proposed in [98], x is represented by setting all the output signals to zero. A simplified hardware structure for positive $Sum1$ is shown in Fig. 6.30. In the last level, only the correction signal y is needed, and the final correction signal is ORed by two correction signals for both $Sum1 > 0$ and $Sum1 < 0$. A leading zero anticipation algorithm for binary redundant encodings can be found in [104]..

Table 6.11: Scenarios of one digit error on leading one position

Sign of $Sum1$	$Sum1$ String pattern	No. of LZ	Example
$Sum1 > 0$	$z^k1z^lp(x)$	k	0...012...
	$z^k1z^ln(x)$	$k+1$	0...01$\overline{2}$...
	$z^kp^+(x)$	k	0...02$\overline{2}$...
$Sum1 < 0$	$z^k\overline{1}z^ln(x)$	k	0...0$\overline{12}$...
	$z^k\overline{1}z^lp(x)$	$k+1$	0...0$\overline{1}$2...
	$z^kn^-(x)$	k	0...0$\overline{22}$...

$z : (s = 0)$; $p : (s > 0)$; $n : (s < 0)$; $p^+ : (s > 1)$; $n^- : (s < 1)$;

$k >= 0, l >= 0$; x: don't care

$$Sum1 > 0 => \begin{cases} p^+ = p^{+l} \cdot z^r + z^l \cdot p^{+r} \\ po = z^l \cdot po^r + po^l \cdot z^r \\ z = z^l \cdot z^r \\ n = n^l + z^l \cdot n^r \\ y = y^l + z^l \cdot y^r + po^l \cdot n^r \end{cases} \tag{6.28}$$

Table 6.12: Node functions for the positive and negative detection trees

$Sum1 > 0$		right branch					
		p^+	po	z	n	x	y
left branch	p^+	x	x	p^+	x	x	x
	po	x	x	po	y	x	x
	z	p^+	po	z	n	x	y
	n	n	n	n	n	n	n
	x	x	x	x	x	x	x
	y	y	y	y	y	y	y

$Sum1 < 0$		right branch					
		n^-	no	z	p	x	y
left branch	n^-	x	x	n^-	x	x	x
	no	x	x	no	y	x	x
	z	n^-	no	z	p	x	y
	p	p	p	p	p	p	p
	x	x	x	x	x	x	x
	y	y	y	y	y	y	y

po : $(s = 1)$; no : $(s = -1)$; y: need correction

$$Sum1 < 0 => \begin{cases} n^- = n^{-l} \cdot z^r + z^l \cdot n^{-r} \\ no = z^l \cdot no^r + no^l \cdot z^r \\ z = z^l \cdot z^r \\ p = p^l + z^l \cdot p^r \\ y = y^l + z^l \cdot y^r + no^l \cdot p^r \end{cases} \tag{6.29}$$

where s^l means the signal s is from the left branch, and s^r means the signal s is from the right branch; " $\cdot$ " means logic and, " $+$ " means logic or.

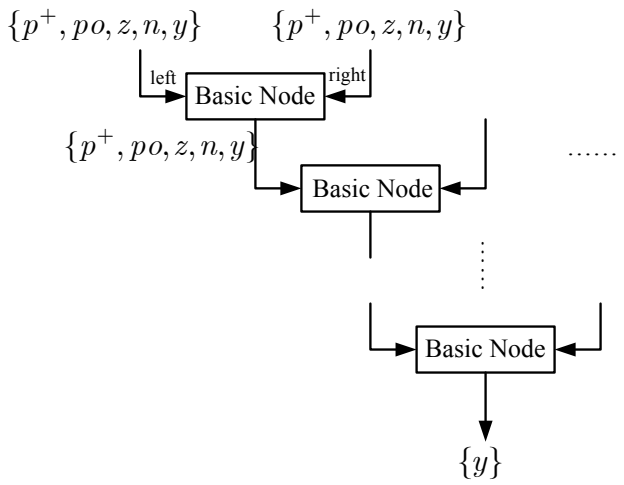


Figure 6.30: Hardware structure of the correction detection unit

To decide the post-alignment shifting amount $Rsa2$, there are some intermediate signals that have to be generated first. As shown in Fig. 6.29, the intermediate signals are generated and fed into three detection units. Afterwards, the corresponding variables are obtained, and the shifting amount is therefore decided. All the intermediate signals have been already introduced in Table 6.11 and Table 6.12. These seven signals, z, p, n, p^+, p^-, po and no are directly generated from the $Sum1$.

Sticky Bits Generation

Since the digit-set of the result $Sum1$ is redundant in $[-8, 7]$, if right shifting is applied in post-alignment, the shifted out digits can be positive, zero or negative. For example, after

right shifting, the shifted out digits after the rounding digit of the string "123...90...56.5$\overline{4}$32..." is "$\overline{4}$32...". The rounding digit 5 after the point is therefore reduced by one due to the negative sticky digit. Consequently, the rounding process is more complicated than the one in the conventional architecture. To correctly round the result, a signed sticky with two bits has to be detected.

If the signal p_i is set to 1, the value from the i^{th} digit to right digits is larger than zero or positive, and if z_i is set to 1, the value from the i^{th} digit to right digits equals zero. If both signals are 0, the value is negative. For example, the p and z signals are set to "0011..." and "0000..." for string "0$\overline{3}$21...". The detection algorithm is similar to the carry propagation process, in which a positive (negative) digit will be propagated to the left by another positive (negative) digit or a zero digit and killed by a negative (positive) digit. A prefix tree structure is applied to generate the sticky signals. The logic of a basic cell on the prefix tree is given in equation (6.30). Once the signals p and z are ready, the correct sticky digit can be obtained by right shifting it with $Rsa2$ digits. Therefore, the right shifter for obtaining sticky only contains 2 bits at the output.

$$\begin{aligned} p &= p^l + z^l \cdot p^r \\ z &= z^l \cdot z^r \end{aligned} \tag{6.30}$$

6.4.3 Rounding

In the proposed DFMA, the redundant result which is shifted in the post-alignment unit is sent to the rounding block to obtain the final rounded result in the conventional BCD encoding. Since the digit of the result $Sum2$ can be positive or negative, two major problems are emerged in the proposed system. First, the digits shifted out of the rounding digit can be positive or negative. In some rounding cases, this might affect the value on the rounding digit and hence the rounded result. For example, "*ssss...ssss*51234..." and "*ssss...ssss*5$\overline{1}$234..." have same digits on the most significant 16 digits (*ssss...ssss*) and the rounding digit (5) and different rest of digits on the right side (1234... and $\overline{1}$234...). In the Ties-to-Away mode, the first result will be rounded to "*ssss...ssss* + 1", and the latter one will be rounded to "*ssss...ssss*". Second, the final result can be a negative number which needs to be inverted

first. In this case, all the consideration of the rounding algorithm is based on the negation of the current result. For example, if "$ssss...ssss51234...$" is negative, it is negated to "$\overline{ssss...ssss51234}...$", and rounded it to "$\overline{ssss...ssss}-1$" in the Ties-to-Away rounding mode. However, the simplicity of the proposed number system should be noticed. In the proposed digit-set, there is no positive carry propagation at all. The only consideration is the negative carry propagation or borrow propagation. The final result is divided into two parts (i.e. the most significant 15 digits and the least significant 1 digit). In the most significant 15 digits, the only processes are the negation of the negative result and the negative carry propagation for digit-set conversion. On the other hand, the least significant digit can be added by "± 1" or retained according to the rounding increment.

The rounding algorithm is therefore divided into two steps, obtaining rounding increment and converting the result with the consideration of the rounding increment. An algorithm which generates the increment accordingly is given in Tables 6.13-6.15. Suppose a positive final result with a negative intermediate sum "$\overline{1}000...000\overline{3}.51234...$" creates an increment "$+1$" in Ties-to-Away mode. But since the intermediate sum is less than zero, it is negated to "$1000...0003.\overline{51234}...$". Moreover the increment is negated to "-1". The final result is therefore rounded and converted to "1000...0002". The complicated computation is encapsulated in the following conversion algorithm. The signal $SignF$ means the sign of the final result, and the logic is optimized by "$Sign2 \oplus (SX \oplus SY)$".

The top level structure of the proposed rounder is given in Fig. 6.31. Since the digit-set of the post-aligned sum is in $[-8, 7]$, after adding the increment from the rounding digit, no positive carry will be generated from $Sum2_{lsd}$. Instead, a negative carry might be generated (e.g. "$1200...000\overline{3}$" or "1200...0000" with increment -1). Therefore, the propagation bits and generation bits of the negative carry for the most 15 significant digits are generated first by a prefix tree structure. In the meantime, the possible negative carry (NC_{lsd}) from $Sum2_{lsd}$ is generated as shown in equation (6.31). Once the negative carry C for all the 16 digits of CR is obtained, the digit-set conversion with absolute value conversion algorithm is performed as given in the equations (6.32 and 6.33). The correction value $Cor2$ is simply obtained by the nine's complement conversion algorithm with borrow consideration. Since the LSD of CR will be adjusted by the rounding increment, the correction value for the LSD is different

Table 6.13: Rounding increment generation algorithm of "TiesToAway" and "TowardPositive" modes

Ties to Away					
$Sign2 = 0$			$Sign2 = 1$		
RD[a]	SD[b]	inc	RD	SD	inc
$[6, 7]$	x[c]	+1	$[6, 7]$	x	-1
5	0 or 1	+1	5	1	-1
5	-1	0	5	0 or -1	0
$[0, 4]$	x	0	$[0, 4]$	x	0
$[-4, -1]$	x	0	$[-4, -1]$	x	0
-5	0 or 1	0	-5	1	0
-5	-1	-1	-5	0 or -1	+1
$[-8, -6]$	x	-1	$[-8, -6]$	x	+1
Toward Positive					
$Sign2 = 0, SignF = 0$			$Sign2 = 0, SignF = 1$		
RD	SD	inc	RD	SD	inc
$[1, 7]$	x	+1	$[1, 7]$	x	0
0	1	+1	0	1 or 0	0
0	0 or -1	0	0	-1	-1
$[-8, -1]$	x	0	$[-8, -1]$	x	-1
$Sign2 = 1, SignF = 0$			$Sign2 = 1, SignF = 1$		
$[1, 7]$	x	0	$[1, 7]$	x	-1
0	1 or 0	0	0	1	-1
0	-1	+1	0	0 or -1	0
$[-8, -1]$	x	+1	$[-8, -1]$	x	0

[a] RD=Rounding Digit

[b] SD=Sticky Digit

[c] x=don't care

Table 6.14: Rounding increment generation algorithm of "TiesToEven" and "TowardNegative" modes

Ties to Even							
$Sign2 = 0$				$Sign2 = 1$			
RD	SD	LE[a]	inc	RD	SD	LE	inc
$[6, 7]$	x	x	$+1$	$[6, 7]$	x	x	-1
5	1	x	$+1$	5	1	x	-1
5	0	0	$+1$	5	0	0	-1
5	0	1	0	5	0	1	0
5	-1	x	0	5	-1	x	0
$[0, 4]$	x	x	0	$[0, 4]$	x	x	0
$[-4, -1]$	x	x	0	$[-4, -1]$	x	x	0
-5	1	x	0	-5	1	x	0
-5	0	0	-1	-5	0	0	$+1$
-5	0	1	0	-5	0	1	0
-5	-1	x	-1	-5	-1	x	$+1$
$[-8, -6]$	x	x	-1	$[-8, -6]$	x	x	$+1$

Toward Negative					
$Sign2 = 0, SignF = 0$			$Sign2 = 0, SignF = 1$		
RD	SD	inc	RD	SD	inc
$[1, 7]$	x	0	$[1, 7]$	x	$+1$
0	1 or 0	0	0	1	$+1$
0	-1	-1	0	0 or -1	0
$[-8, -1]$	x	-1	$[-8, -1]$	x	0
$Sign2 = 1, SignF = 0$			$Sign2 = 1, SignF = 1$		
$[1, 7]$	x	-1	$[1, 7]$	x	0
0	1	-1	0	1 or 0	0
0	0 or -1	0	0	-1	$+1$
$[-8, -1]$	x	0	$[-8, -1]$	x	$+1$

[a] LE=LSD is Even

Table 6.15: Rounding increment generation algorithm of "TowardZero" mode

Toward Zero					
$Sign2 = 0, SignF = 0$			$Sign2 = 0, SignF = 1$		
RD	SD	inc	RD	SD	inc
$[1, 7]$	x	0	$[1, 7]$	x	0
0	1 or 0	0	0	1 or 0	0
0	-1	-1	0	-1	-1
$[-8, -1]$	x	-1	$[-8, -1]$	x	-1
$Sign2 = 1, SignF = 0$			$Sign2 = 1, SignF = 1$		
$[1, 7]$	x	-1	$[1, 7]$	x	-1
0	1	-1	0	1	-1
0	0 or -1	0	0	0 or -1	0
$[-8, -1]$	x	0	$[-8, -1]$	x	0

than the other 15 digits. If the intermediate result $Sum2$ is less than zero, the XOR gates are necessary to negate the digit. Note that, in the digit-set conversion algorithm, the least significant bit of the negative carry signal C is equal to the sign of the $Sum2$. Therefore, if $Sign2 = 1$, the negative carry might be propagated through $Sum2\{1\}$. Subsequently, the negative carry for higher 15 digits can be obtained by the equation: $NC_{i:0} = g_{i:0}\&(p_{i:0}|NC_{lsd})$.

To clarify the rounding and conversion algorithm, the example provided in Fig. 6.23 is considered. Since the leading positive one in $Sum1$ will be corrected, it is not shifted in the 17-digit $Sum2$, and the sign of the $Sum1$ is positive. In the rounder, the 15-bit propagation "000000000100000" and generation "111111001000101" signals for the negative carry of the most 15 significant digits are first created. In the Ties-to-Away rounding mode, no increment is generated from the rounding digit $\overline{1}$. Subsequently, the LSD "$\overline{3}$" creates a negative carry to the higher digits. The 15-bit propagation and generation signals, together with the negative carry from LSD and the positive $Sign2$, create a final negative carry "11111100100010110". Afterwards the correction signal "$99999a0fa00faf9a$" is further created by the equations

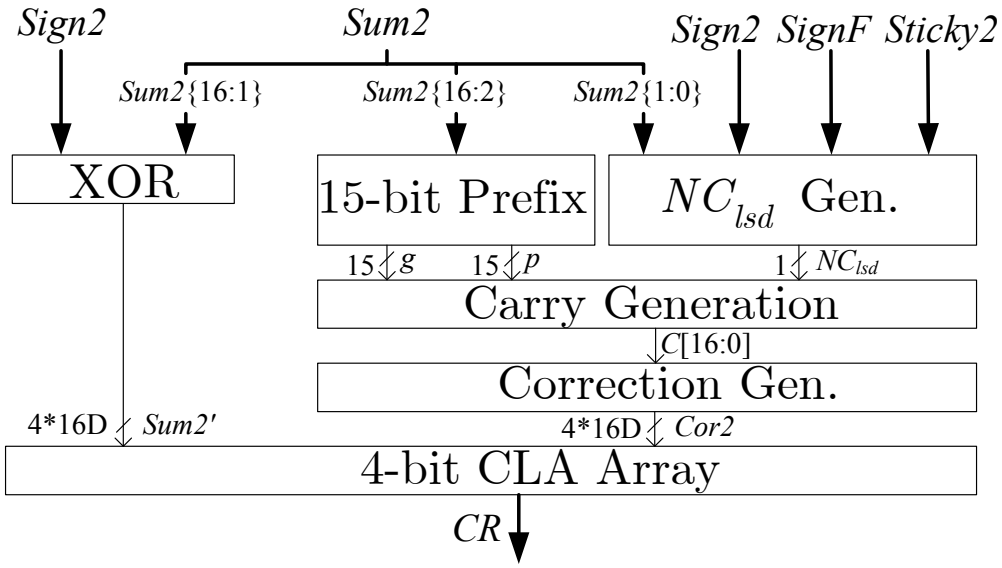


Figure 6.31: Architecture of the rounder

(6.32 and 6.33). After all, the final rounded significand is obtained by adding the bit-inverted $Sum2' =$ "$\overline{3456563}2\bar{3}043\bar{1}1\overline{23}$" with the correction signal.

$$
\begin{aligned}
NC_{lsd}^{+1} &= \begin{cases} 1 & \text{if } Sum2\{1\} < -1 \text{ or} \\ & (Sum2\{1\} = -1 \& Sign2 = 1) \\ 0 & \text{otherwise} \end{cases} \\
NC_{lsd}^{0} &= \begin{cases} 1 & \text{if } Sum2\{1\} < 0 \text{ or} \\ & (Sum2\{1\} = 0 \& Sign2 = 1) \\ 0 & \text{otherwise} \end{cases} \\
NC_{lsd}^{-1} &= \begin{cases} 1 & \text{if } Sum2\{1\} < 1 \text{ or} \\ & (Sum2\{1\} = 1 \& Sign2 = 1) \\ 0 & \text{otherwise} \end{cases} \\
NC_{lsd} &= \begin{cases} NC_{lsd}^{+1} & \text{if } RD_{inc} = +1 \\ NC_{lsd}^{0} & \text{if } RD_{inc} = 0 \\ NC_{lsd}^{-1} & \text{if } RD_{inc} = -1 \end{cases}
\end{aligned}
\tag{6.31}
$$

$$Cor2\{0\} = \begin{cases} 1 & \text{if } C_{1:0} = 00 \ \& \ RD_{inc} = 1, \\ 10 & \text{if } C_{1:0} = 01 \ \& \ RD_{inc} = 1, \\ 11 & \text{if } C_{1:0} = 10 \ \& \ RD_{inc} = 1, \\ 0 & \text{if } C_{1:0} = 11 \ \& \ RD_{inc} = 1, \\ 0 & \text{if } C_{1:0} = 00 \ \& \ RD_{inc} = 0, \\ 11 & \text{if } C_{1:0} = 01 \ \& \ RD_{inc} = 0, \\ 10 & \text{if } C_{1:0} = 10 \ \& \ RD_{inc} = 0, \\ 1 & \text{if } C_{1:0} = 11 \ \& \ RD_{inc} = 0, \\ -1 & \text{if } C_{1:0} = 00 \ \& \ RD_{inc} = -1, \\ 12 & \text{if } C_{1:0} = 01 \ \& \ RD_{inc} = -1, \\ 9 & \text{if } C_{1:0} = 10 \ \& \ RD_{inc} = -1, \\ 2 & \text{if } C_{1:0} = 11 \ \& \ RD_{inc} = -1. \end{cases} \tag{6.32}$$

where $C = \{NC[15:0], Sign2\}$

$$Cor2\{15:1\} = \begin{cases} 0 & \text{if } C_{i+1:i} = 00 \ \& \ C_0 = 0, \\ -1 & \text{if } C_{i+1:i} = 01 \ \& \ C_0 = 0, \\ 10 & \text{if } C_{i+1:i} = 10 \ \& \ C_0 = 0, \\ 9 & \text{if } C_{i+1:i} = 11 \ \& \ C_0 = 0, \\ 10 & \text{if } C_{i+1:i} = 00 \ \& \ C_0 = 1, \\ 11 & \text{if } C_{i+1:i} = 01 \ \& \ C_0 = 1, \\ 0 & \text{if } C_{i+1:i} = 10 \ \& \ C_0 = 1, \\ 1 & \text{if } C_{i+1:i} = 11 \ \& \ C_0 = 1. \end{cases} \tag{6.33}$$

Chapter 7

Comparison and Discussion

In this chapter, all the proposed designs are firstly evaluated by synthesizing the Verilog model with Synopsys tools. Furthermore, the differences on performance between the proposed designs and previous corresponding designs are illustrated and analyzed. The organization of rest sections are addition (section 7.1), multiplication (section 7.2), and DFMA (section 7.3).

7.1 Decimal Fixed-point Addition

A model of the proposed decimal SD adder is implemented in VHDL. The exhaustive test to ensure the correctness is performed. Subsequently, the proposed design was synthesized by Synopsys Design Compiler in STM 90 nm CMOS technology with normal case parameters (1.2V, 25℃).

To compare with other designs, the designs in [51], [56], [52] and [59] were also implemented in the same technology. The implementation results, including timing delay, hardware area, power consumption, area delay product (ADP) ratio and power delay product (PDP) ratio are listed in Table 7.1. On the timing delay, the proposed design has at least a 34% improvement. On the performance in terms of the ADP and PDP, our design could have more than 32% and 76% improvement compared with the referenced designs, respectively.

For further evaluation, the hardware area and power consumption under different timing constraints of the designs in [51], [56], [52], [59] and the proposed one are listed in Fig. 7.1 and Fig. 7.2. The less area and less timing delay compared with all other works could be obtained simultaneously once the timing constraint is larger than 0.3 ns. Since currently the decimal computation is mostly used on high performance server [57], we focus on computation speed rather than hardware cost which could be improved in the future.

Table 7.1: Synthesized results and comparison of 16-digit adders

	Digit Set	Delay (ns)	Ratio	Area (μm^2)	Ratio	ADP Ratio	Power (mW)	Ratio	PDP Ratio	FW-Conv (ΔG)	BW-Conv (ΔG)
[51]	$[-9, 9]$	0.49	1.69	35561	2.76	4.66	57.94	3.55	5.99	0	N/A
[56]	$[-9, 9]$	0.51	1.76	22078	1.71	3.01	35.67	2.18	3.84	0	N/A
[52]	$[-8, 9]$	0.39	1.34	12654	0.98	1.32	21.39	1.31	1.76	1	$2n + 8$
[59]	$[-7, 7]$	0.45	1.55	12781	0.99	1.54	20.01	1.22	1.90	9	$2n + 10$
Proposed	$[-9, 9]$	0.29	1	12898	1	1	16.34	1	1	0	$2 \log n + 10$

Since our design works on the digit set $[-9, 9]$ and the operands are encoded in two's complement, no extra forward converter which converts the BCD inputs to the proposed digit set is needed at all. In [52], the authors use the digit set $[-8, 9]$, so there is an OR gate in the front converter mentioned in their paper. In [59], a combinational logic to generate the correction signal and a 4-bit adder are proposed to convert BCD to RBCD encoding with $9\Delta G$ (i.e., level of gates) delay.

For the backward converter which converts the digit set in the proposed design to the conventional BCD encoding, in [52] and [59], the authors proposed two algorithms which process in linear timing delay proportional to the digit width of the input. Furthermore, to generate the absolute value, the aforementioned designs need more logics to check the sign of the result and invert the result digit by digit which are not counted in Table 7.1. The proposed converter in this thesis could generate the absolute value of the result in BCD encoding with a timing delay logarithmically proportional to the digit width.

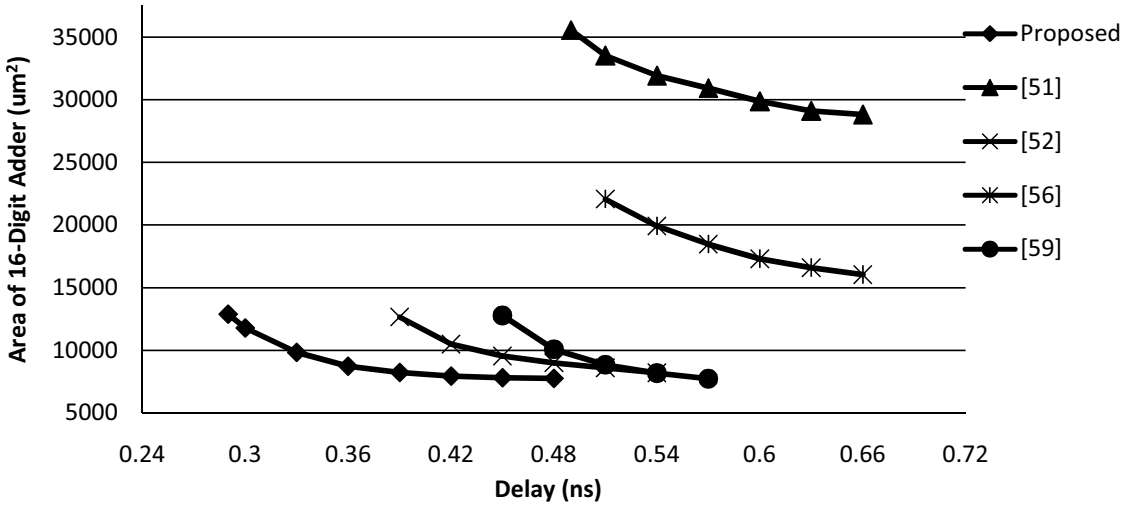


Figure 7.1: Area-Delay Comparison

7.2 Decimal Fixed-point Multiplication

To compare the proposed multiplication algorithm with other designs, a delay model is firstly created in terms of fanout-of-4 inverter's delay on the estimated critical path. Thus the effects from fanout gates and the gate scaling are ignored in the theoretical comparison.

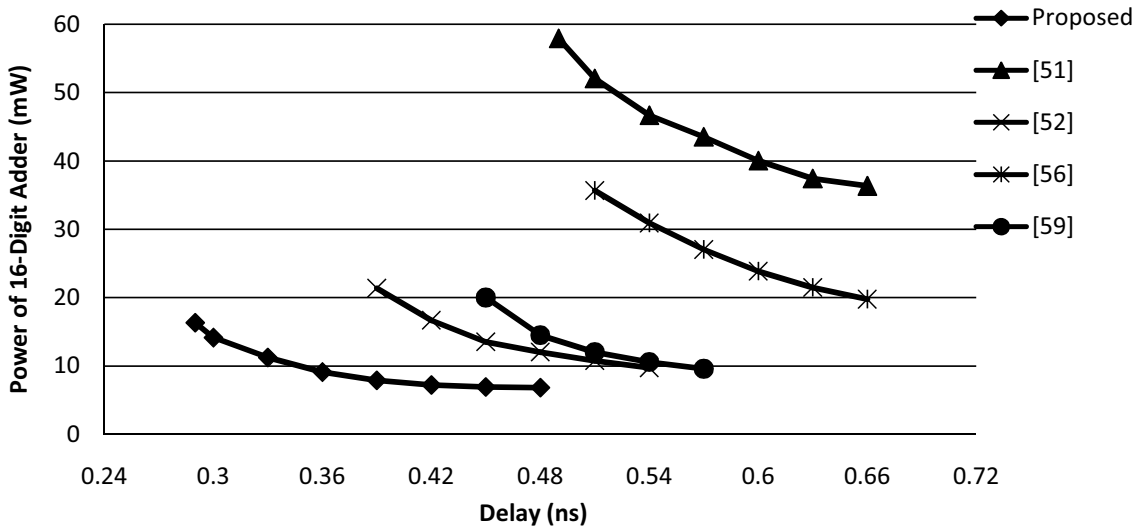


Figure 7.2: Power-Delay Comparison

To obtain a more accurate comparison, a Verilog-HDL model of the proposed 16×16-digit multiplier is synthesized with STM 90 nm standard cell normal case library (1.0V, 25℃). For fairly comparing with previous works implemented in different technologies, the fanout-of-4 inverter's delay and NAND2 gate's area are applied to measure the performance of different designs in different technologies. Since the values of 1 unit of these two metrics change as the technology, these units provide a comparison among different designs on an identical reference. A discussion on the differences of performance between our proposed architecture and other designs is given afterwards.

7.2.1 Parallel Multiplication

Performance Evaluation

In Table 7.2, the numbers of logic gates (i.e., NAND2 gate or ΔG) for different stages of the parallel 16×16-digit multipliers from other designs are listed. We assume that an AND2/OR2 gate equals to one NAND2 gate, and an XOR gate equals to two NAND2 gates. The PPG unit in Table 7.2 is used to generate the partial products in the format which can be directly processed by the PPR unit in the next stage. For example, the decimal carry save adder, to reduce the multiples from double-BCD format (i.e., double-four-bit) to BCD-CS format (i.e., one-four-bit) applied in the sequential design in [63] and the parallel design in [65],

is counted into the PPG stage. Additionally, to fairly analyze the efficiency of the PPR methods, we suppose that the outputs of the PPR unit are two numbers in arbitrary formats (e.g., double-BCD or BCD-CS format). Thus, the fourth level of the ODDS adder in [70] and the final simplified CLA shown in Fig. 6.12 are assumed as the adder setup unit in the final stage. Finally, for the three sequential multipliers in the bottom of Table 7.2, only the ratio (e.g., marked by an asterisk) between the ΔG involved in iterative cycles and the proposed design is provided, since other non-iterative cycles can be pipelined without reducing the overall efficiency of the multiplier. As shown in Table 7.2, some algorithms may be faster than our proposed design on PPG or PPR, but by considering the trade-off among three multiplication stages, our design can perform the best.

To obtain a more accurate performance on not only timing delay but also hardware cost, a hardware model for a 16×16-digit multiplier is implemented by Verilog-HDL and synthesized using Synopsys Design Compiler and STM 90 nm CMOS standard cells library which has $45ps$ as the delay of an inverter with fanout of 4 inverters and $4.4um^2$ as the area of the smallest NAND2 gate. 500,000 random cases and 100 manually created boundary cases are verified in the Verilog-HDL model to guarantee the correctness. The delay in picosecond of each module on the critical path is shown in Table 7.4. Furthermore, the delay-area values which are measured under Design Compiler within the range from $1.94ns$ and 49900 NAND2 to $2.65ns$ and 36655 NAND2 are shown in Fig. 7.3. The delay-area values of other parallel designs are also provided. The latest designs of the architectures of Radix-10 and Radix-5 in [68] and the architecture in [70] are implemented and evaluated with our library and synthesis environment.

Comparison and Discussion

In Table 7.3, the state-of-the-art decimal multipliers for 16-digit operands are listed in terms of timing delay, hardware area, throughput, and latency. In [65], the design is synthesized using the STM 90 nm library which is the same library as used in our design. The latency provided by the authors is $2.65ns$, which equals to about 58.9 FO4. In [66], the authors improve the design in [65] and reduce the latency to $2.51ns$ (55.8 FO4) by an elaborated PPR tree and a binary to decimal converter. Both of these designs have the area of 68000

Table 7.2: Delay analysis of 16 × 16-digit decimal fixed-point multipliers

Architecture		PPG (ΔG)	Ratio	PPR (ΔG)	Ratio	Setup + Final Adder (ΔG)	Ratio	Total (ΔG)	Ratio
Parallel	[65]	20	1.67	60	1.28	17	1.06	97	1.29
	[74]	37	3.08	54	1.15	25	1.56	116	1.55
	[67]	9	0.75	57	1.21	19	1.19	85	1.13
	[70]	11	0.92	46	0.98	29	1.81	86	1.15
	Radix-10 [68]	39	3.25	42	0.89	23	1.44	104	1.39
	Radix-5 [68]	11	0.92	53	1.13	23	1.44	87	1.16
	Proposed	12	1	47	1	16	1	75	1
Sequential	[42]	11	-	13×17	-	43	-	275	2.95*
	[64]	13	-	31×17	-	-	-	-	7.03*
	[63]	20	-	20×17	-	17	-	377	4.76*

* Ratio=$\Delta_{\text{PPR}}/\Delta_{\text{proposed total}}$

Table 7.3: Performance comparison of 16 × 16-digit decimal fixed-point multipliers

Architecture		# Cycles	Cycle Time	Latency		Throughput	Area	
			(FO4)	(FO4)	Ratio	Mult./Cycle	(NAND2)	Ratio
Parallel	Bin. Radix-4 [65]	1	-	31.1	0.72	1	34000	0.68
	[65]	1	-	58.9	1.37	1	68000	1.36
	[66]	1	-	55.8	1.29	1	68000	1.36
	[74]	1	-	54.4	1.26	1	60500	1.21
	[67]	1	-	53.5	1.24	1	79600	1.60
	[70]	1	-	48.1	1.12	1	49500	0.99
	Radix-10 [68]	1	-	48.4	1.12	1	44400	0.89
	Radix-5 [68]	1	-	47.8	1.11	1	50900	1.02
	Proposed	1	-	43.1	1	1	49900	1
Sequential	[42]	24	12.7	305	5.00*	1/17	31500	0.63
	[64]	20	16	320	6.31*	1/17	16000	0.32
	[63]	20	14.7	294	5.80*	1/17	18550	0.37

* Ratio=$(FO4_{Delay}/Throughput)/FO4_{proposed}$

Table 7.4: Critical path of the proposed 16 × 16-digit multiplier

Gen. of mult.	Sel.+Inv.	PPR	GP gen.+Prefix Tree+Sel.
160ps	140ps	1230ps	410ps

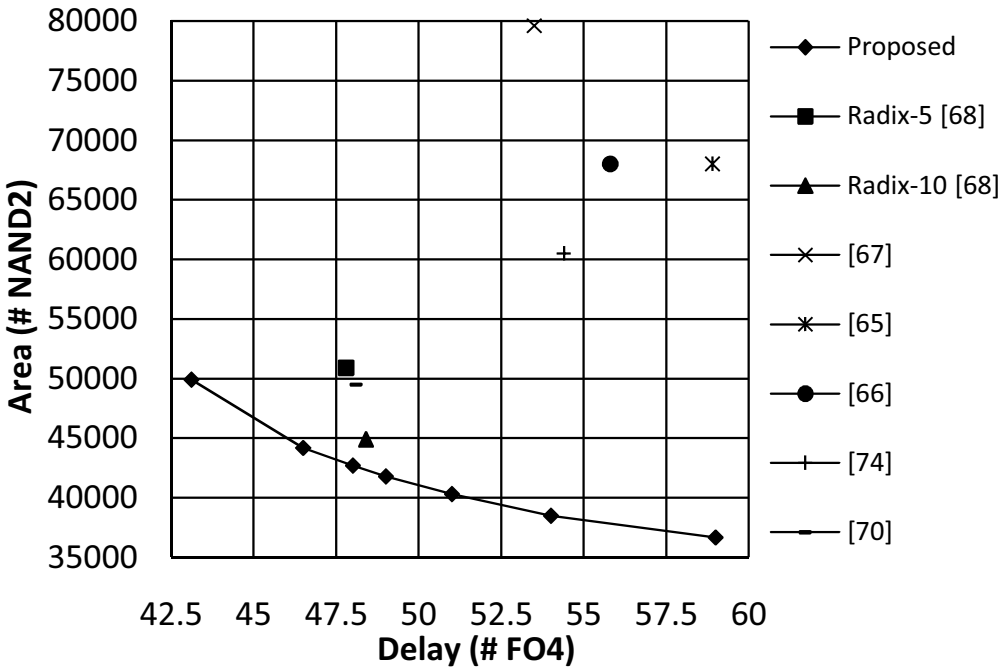


Figure 7.3: Delay-area space of the decimal multipliers

NAND2 gate. Our PPG algorithm avoids the decimal CSA in the PPG unit applied in those designs. Furthermore, the PPG which consists of six levels of BCD-FAs in [65] involves six levels of carry propagations in 4-bit width which lower the performance of the multiplier. These two radix-10 combinational multipliers cost at least 29% more timing delay and 36% bigger area than our proposed design.

In [74], the authors propose a parallel decimal floating-point multiplier by applying the fixed-point design with radix-10 architecture proposed in [76]. Such a parallel decimal multiplier applies new decimal encodings (i.e., BCD-4221 and BCD-5211) to simplify the design of the PPR tree. In the proposed radix-10 design in [76], a carry propagation through all bits in an operand is involved in the PPG stage. Besides, the proposed 17:2 reduction tree with binary CSAs and encoding converters is slower than our proposed PPR unit with two levels of SD adders. Overall, our proposed algorithm reduces about 26% timing delay and 21% area compared to the radix-10 design applied in [74].

In [67], the authors propose a method to represent $8X$ and $9X$ in two digits to avoid the long path in PPG. Consequently, the delay of the PPG is reduced significantly. To reduce the partial products, the authors present an architecture within 6-level simplified BCD-FA. Additionally, after the PPR unit, a narrower result is obtained. However, the level of prefix tree applied in the final addition cannot be reduced, since the reduction on the result of PPR is not over half of the width. The BCD-FA used in the PPG in [65] is replaced by a simplified BCD half adder. Nevertheless, the digit-level reduction tree based BCD-FA shows the disadvantage of the relatively large delay and big area as described for the design proposed in [65]. The synthesized design in [67] under TSMC 130 nm standard cells library costs about 53.5 FO4 and 79600 NAND2, respectively. Although the PPG unit which has no XOR gate and a simple selection circuit is faster than our proposed PPG, due to the slower PPR and final addition in [67], our multiplier could gain about 24% less delay with 60% less hardware cost overall.

In the SD multiplier proposed in [70], the efficiency of the PPG and PPR units are guaranteed as in our proposed design. However, the double-BCD format partial product array takes off the advantage of the overall performance on timing delay in [70]. Furthermore, the final overloaded decimal digit set adder with the following traditional digit converter is slower than the simplified 4-bit CLA and the converter proposed in our design. Finally, the synthesis result shows that our design takes 12% less delay with almost the same area cost.

In [68], the authors improve the design they proposed in [76]. The PPR trees are optimized for both radix-10 and radix-5 architectures in [76]. Thus as shown in Table 7.2, the number of gates for the 17:2 reduction tree in radix-10 architecture is faster than our proposed PPR. However, the 32:2 reduction tree for radix-5 architecture is still slower than our design, since about twice operands are processed in such a structure. Moreover, the carry propagation which affects the overall performance in the PPG of the radix-10 architecture cannot be avoided. In the radix-5 architecture, the partial products are generated by a couple of recoders within a small delay. Additionally, the unconventional encodings avoid the complicated decimal correction in most of other works. Thus, the proposed PPR tree could be arranged as the binary CSA tree (i.e., Wallace-like structure based on binary CSAs and encoding converters). However, our design balances the delay of PPG and PPR and applies a simpler

final conversion compared to the designs in [68]. Overall, our design has about 11% less delay with 2% less area compared to the fastest state-of-the-art design (radix-5) in [68], and has about 12% less delay with 11% more area compared to the radix-10 architecture in [68].

The sequential designs of the fixed-point decimal multiplication are also listed in Table 7.7. The latency ratio with asterisk is calculated according to the FO4 spent on iterative cycles. Such sequential designs show the advantage on the area cost and disadvantage on latency and throughput as expected.

Generally speaking, the output format of a PPG algorithm can be a single-BCD (e.g., the radix-10 architecture in [68]), a single-BCD with identical carry for each partial product (e.g., the proposed method), a BCD-CS (e.g., the method applied in [65]), or a double-BCD (e.g., the algorithms used in [67, 70], and the radix-5 architecture in [68]). In general, the less bits in the output of a PPG, the more complexities in the PPG, and the less complexities in the PPR. For example, the single-BCD result of the PPG in the radix-10 architecture in [68] provides the chance to apply the simplest PPR unit, but it cannot avoid the carry propagation in the PPG. On the other hand, the double-BCD result of the simplest PPG in [67] involves a complicated PPR unit. Our proposed PPG method generates the partial product which has the bit-width close to the single-BCD format without the carry propagation. Thus, the complexity of the PPR is potentially reduced. Moreover, since only simple combinational logic is applied to convert the digit-set in the proposed PPG, the BCD-FA used in the PPG of [65] is eliminated. The PPR algorithm highly depends on the encoding of the result of the PPG. Besides, in the PPR, the less input width the better, and the less bit-level carry propagation the better. Our proposed PPR design based on the multi-operand SD addition which involves two bit-level carry propagation is a bit more complicated than the design in [68] with the same input width (i.e., $n + 1$ digits), and is simpler than the design in [68] with the double-sized input width and the designs based on BCD-FA in [65, 67]. The carry propagation in the final addition cannot be avoided in any method, since the result of the PPR is in the redundant format. However, the efficiency of the final addition or conversion can be affected by the complexity of the setup logic and the prefix tree. The proposed conversion method involves a 4-bit carry propagation to generate the propagation and generation bit for each digit, but by applying the hybrid carry prefix tree, the logic on the critical path is

minimized. After all, although the proposed method is not the simplest on some stages of a multiplication, the overall delay of the proposed multiplication is minimized by considering the trade-off of the complexity in each stage.

7.2.2 Sequential Multiplication

The proposed multiplier, according to Fig. 6.20, consists of three main parts namely PPG, PPA and Conversion each of which consumes 1, $n+1$ and 2 cycles, where n is the digit width of the input. This concludes that the entire single multiplication can be performed in $n+4$ cycles with the initiation interval of $n+1$ cycles.

The cycle time, thus the clock frequency, determined by the critical delay path of the PPA, is equal to the latency of the multi-operand adder shown in Fig. 6.18. The details of the critical delay path are tabulated in Table 7.5.

Table 7.5: The critical delay path of the proposed multiplier (ns)

(4:2) Compressor	4-bit CLA	Recoder	Register	Total
0.17	0.13	0.15	0.18	0.63

The area consumption of the proposed 16-digit multiplier is evaluated as the sum of the area cost of various constituent parts tabulated in Table 7.6.

Table 7.6: Area consumption of the proposed 16-digit multiplier

	Area (um^2)
PPG	6900
PPA	20100
Conversion	4500
Registers	7680
Misc	220
Total	39400

Comparison and Discussion

The multiplier described in [63] requires $n + 4$ cycles per multiplication where the cycle time of is equal to the latency of a BCD (4:2)-compressor plus registers. According to the evaluation in [68], the cycle time and the area of this design is 16 FO4 and 16000 NAND2 gates, respectively, for a 16-digit multiplication.

In [42], the multiplier using the overloaded decimal representation calls for a special decimal carry-free adder which brings about a critical delay path of a (4:1) multiplexer, a +6 increment block, a binary full-adder plus registers. This concludes to the latency of 12.7 FO4 where the number of required cycles is $n + 8$. The area of this multiplier for 16-digit operands is reported as 31500 NAND2 gates.

Another multiplier proposed by Erle in [64] takes the advantage of the decimal signed-digit adder which is introduced in [50] for the iterative portion of the PPA. Thus the latency of the redundant adder plus registers (i.e., 14.7 FO4) determines the cycle time. The number of required cycles is the same as [63] (i.e., $n + 4$) and the area cost is reported as 18550 NAND2 gates for a 16-digit multiplication.

In accordance with the above discussions, Table 7.7 illustrates the details of the evaluation results and compares the proposed design with others in terms of latency and area. Moreover, the simulation results of the proposed multiplier based on delay constraints are depicted in Fig. 7.4. It is shown that the proposed design consumes lower area in comparison with the previous works. The evaluation and comparison results reveal the undisputed area advantage of the proposed sequential decimal multiplier over the previous sequential designs.

Table 7.7: Comparison of the 16-digit multipliers

	Cycle time (FO4)	Ratio	No. of cycles	Total Latency (FO4)	Ratio	Area (NAND2)	Ratio
[63]	16	1.14	20	320	1.14	16000	1.79
[42]	12.7	0.91	24	305	1.09	31500	3.52
[64]	14.7	1.05	20	294	1.05	18550	2.07
Proposed	14	1	20	280	1	8960	1

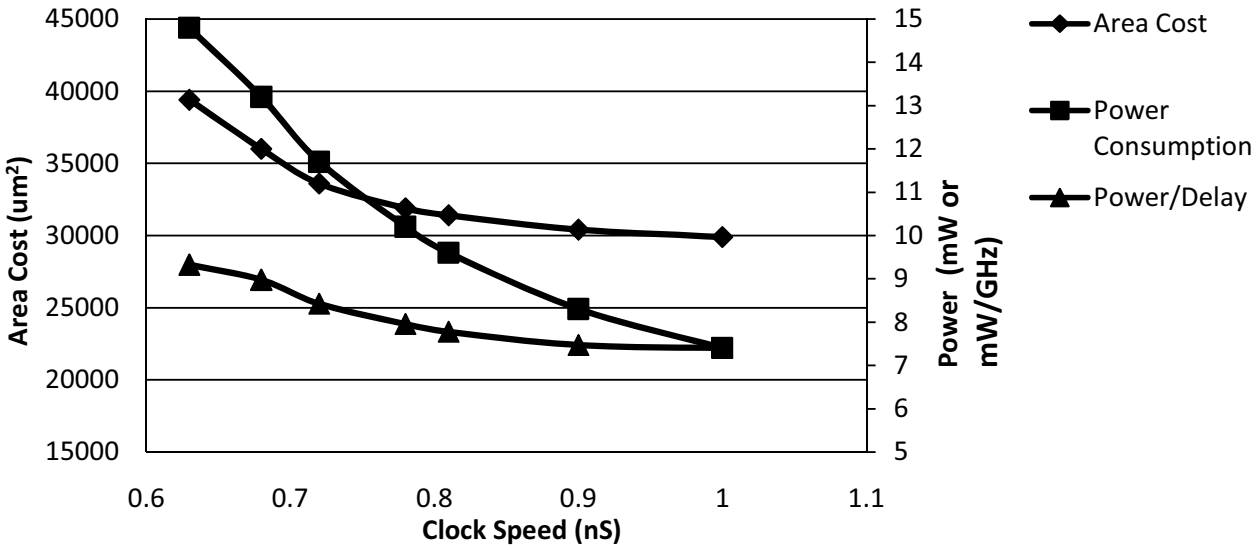


Figure 7.4: Evaluation of speed, area, power consumption of the proposed sequential multiplier

7.3 Decimal Floating-point FMA

To analyze the performance of the proposed architecture, a Verilog-HDL model is created and verified by a test package with 425599 vectors [101] and 50K random vectors generated by Python decimal library. Furthermore, the Verilog model is synthesized by Synopsys Design Compiler with the normal case of the STM 90 nm standard cell library [100] (1.0V, 25℃) which has $45ps$ as the delay of an inverter with fanout of 4 inverters and $4.4um^2$ as the area of the smallest NAND2 gate.

7.3.1 Performance Evaluation

In this section, only the synthesis result of the combinational logic of the proposed architecture, which does not contain the registers at the input and output, is provided. In Table 7.8, the delay and area of the entire design which contains six major blocks are shown. If only the combinational configuration is considered, the pre-alignment is not on the critical path. Additionally, the exception processing unit includes the DPD/BCD conversions at the front and end of the design and the post processing unit which handles the exceptions and creates the flag signals.

Table 7.8: Delay and area partition of the proposed architecture

Component	Delay (ns)	Ratio	Area (um^2)	Ratio
Multiplier Array	1.75[a]	44.6%	211362	66.9%
Pre-alignment	1.75	-	19341	6.1%
Adder	0.38[a]	9.7%	38095	12.1%
Post-alignment	0.8[a]	20.4%	33092	10.5%
Rounding	0.5[a]	12.8%	6002	1.9%
Exception Proc.	0.49[a]	12.5%	7933	2.5%
Total	3.92	100%	315825	100%

[a] The units on the critical path.

7.3.2 Comparison and Discussion

To explain the advantages of the proposed architecture, two previously published designs are compared in details in this section. In [95], first, the multiplier involves a final partial product accumulation which is a decimal quaternary tree adder. In our design, the redundant product obtained from the partial reduction of the multiplier is directly used by the following units. Second, since only addend is shifted in our design, the swapping unit to exchange the operands is not necessary, and the pre-alignment shifters are totally moved out from the critical path which passes through the multiplier. However, since both left and right shiftings are performed in [95], the data path is therefore restricted in $2n$ digits which may reduce the area in DFP128 format. Third, the "pre-correction" of the decimal adder, two prefix networks, one 4-bit binary adder, and some combinational logics are involved in the decimal leading zero anticipator which is simultaneously performed with the adder in [95]. On the other hand, in our design the carry free adder is applied before the leading zero anticipator. However less units (i.e., one prefix network, one 4-bit binary adder, and some combinational

logics in the "Transfer digit generation" and "Intermediate signals generation") are applied to figure out the number of leading and trailing zeros in our design. The rounding unit is not analyzed since the detailed design in [95] is not provided.

In [102], although the top level architecture is similar to our design, the complexity of the circuits in sub-modules is different. First, the multiplier which does not include the final partial product accumulation has a similar complexity to our multiplier. However, the multiplexer in the selection unit and two encoding converters in the decimal 4221-BCD CSA take more delay than the correction signal generator in our decimal carry free adder. Second, in the LZA of [102], four steps anticipation algorithm is proposed. However, in our LZA design, the input signals are directly generated from the only one redundant result with 4-bit two's complement on each digit. Additionally, in our design, only binary leading zero detector is performed on critical path to obtain the possible position of the non-zero leading digit. The pattern is only used for correcting the possible one digit error at the end of shifting amount generator for post-alignment. Moreover, the post-alignment shifter in our design is simply a right shifter with optimization to reduce the delay on critical path and hardware area. Finally, the combined addition and rounding unit in [102] employs a binary compound adder with pre-/post-corrections which are bitwise constant adders with multiplexors, whereafter the final result is selected by the rounding increments. On the other hand, the final rounding and conversion unit in our design applies three generation units with only small constant delay and one bitwise binary CLA.

In Table 7.9, a comparison on the synthesis results of three combinational designs and the performance of corresponding software libraries evaluated in [99] are provided. The actual delay of the designs in [102] and [95] under $65nm$ technology are $5.4ns$ and $4.6ns$ which are slower than that of our design under $90nm$ technology. Since previous works are synthesized under different standard cell libraries, the delay and area are unified by $35ps$ of FO4's delay and $1.44um^2$ of NAND2's area in 65nm technology. Under the same metric, our design takes about 66% of timing delay and 83% of hardware area of the previous fastest design proposed in [95]. Additionally, the power estimation of our design is about $114mW$. Note that the number of cycles of the software libraries depends on the processor and compiler.

Table 7.9: Performance comparison

Design	Delay ($FO4$)	Ratio	Area ($NAND2$)	Ratio
[102]	154.3	1.77	107708	1.50
[95]	131.4	1.51	86061	1.20
Proposed	87.1	1	71778	1
decDouble[99]	785[a]	-	-	-
idfpl64[99]	879[a]	-	-	-
decNum[99]	1683[a]	-	-	-

[a] The performance of software libraries is measured by the number of cycles.

7.3.3 Pipeline Configuration

The proposed DFMA can also be regularly configured to perform efficiently. In Fig. 7.5, a possible configuration is shown. In this case, the multiplier array can be divided into 3 cycles. If addition is performed, the multiplier array can be bypassed by the multiplexor at the end of the third cycle in the multiplier array. The following three units can be partitioned accordingly. The minimum cycle time is therefore decided by the timing delay of the pre-alignment unit and pipeline registers. Consequently, the decimal floating-point addition may be finished in 4 cycles at 1.1GHz (i.e., 0.9ns per cycle). On the other hand, if the decimal floating-point multiplication is performed, all the components on the critical path have to be enabled by setting Z to 0. Hence, the DFP multiplication may be finished in 6 cycles at 1.1GHz.

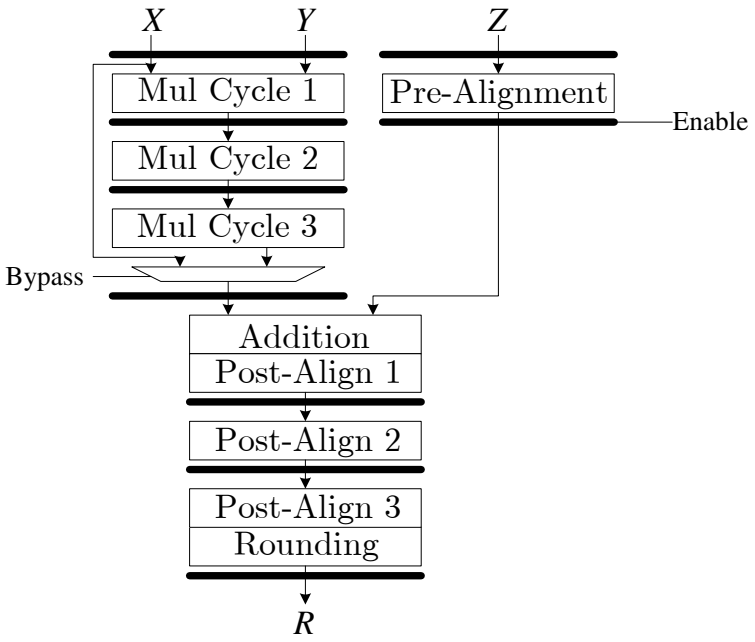


Figure 7.5: A regular pipeline configuration of the proposed architecture

Part IV

Conclusion

Chapter 8

Summary and Future Research

8.1 Summary and Conclusion

In this thesis, the architectures and algorithms to perform decimal fixed-point addition and both parallel and sequential multiplications are first proposed. Afterwards, a new decimal floating-point FMA architecture is described in detail. The study's motivations are recalled before the research and theoretical work appear summarized in the conclusion. In section 1.2, the demands of high performance decimal floating-point arithmetic are discussed. Further, the decimal floating-point processing with hardwired fused multiply-add function is proposed as the major work of this research. Decimal processing with unconventional number systems is also considered and seen to be the competitive technique for performance improvement.

During this research, the previous designs of decimal fixed-point carry free addition and of parallel/sequential multiplications have both been studied. On the basis of previous techniques, our own ideas to further improve the performance of the decimal fixed-point addition and multiplication were proposed.

In the proposed addition, a new nonspeculative decimal carry free adder, which calculates the operands in digit set $[-9, 9]$ with two's complement binary encoding, was discussed. This design determines the transfer digit directly on the input operands instead of on the position sum, the ordinary process used in the conventional carry free signed digit addition algorithm. The digit range of the operands to minimize the cost of exception handling was also analyzed. Furthermore, to improve the speed and reduce the area of the adder, a new algorithm to calculate the result digit without the temporary result was proposed. The synthesized results demonstrate the superiority of the proposed design in terms of the area delay product as well as those of the power delay product. Overall, about 25% of

the delay is reduced in comparison to the fastest state-of-the-art decimal redundant adder. Furthermore, a digit set conversion algorithm that directly converts the absolute value of the signed digit result to the conventional BCD encoding was introduced in detail to solely apply the proposed carry free addition (i.e., without the subsequent processing unit). The new conversion algorithm, which has only one propagation with logarithmical timing delay, is more suitable for high precision computation.

In the proposed multiplications, a new technique to implement the parallel decimal multiplication is first introduced. Unlike other designs, in the proposed algorithm, the multiples (i.e., from $-5X$ to $5X$) are represented in a redundant digit-set $[-8, 8]$. Thus, the signed digit partial products could be generated without the carry propagation in $3X$. To reduce the partial products into one signed digit result, a partial product reduction unit based on the multi-operand signed digit addition was discussed. Moreover, all of the components inside the multi-operand signed digit adder, except for two combinational recoders, could be reused in binary designs. The combinational recoders are currently implemented with logic gates. However, the customized circuits can be applied to further improve the performance. Moreover, the proposed hybrid prefix network displayed the advantages of squeezing more delay from the critical path in the final digit-set conversion for standalone application. Overall, the synthesis result under STM $90nm$ technology showed that the proposed parallel multiplier could achieve about 11% less delay with 2% less hardware cost even when compared to the fastest state-of-the-art parallel decimal multiplier. In the proposed sequential multiplication, we exploited the signed digit multiples generation algorithm for $1X$, $2X$, and $4X$. Furthermore, the partial product generation algorithm (which uses only these three easy multiples) was proposed by introducing redundancy into the second operand in the multiplication (i.e., Y). Following this step, a partial product accumulation architecture, including a series of multi-operand carry free adders, was given to iteratively sum up the partial product in every iteration. At the same time, the lower half digits of the product are converted simultaneously. After the last iteration of the partial product accumulation, the higher half digits of the product are generated by a parallel conversion algorithm with prefix network. Finally, the evaluation of the proposed design illustrated that our design achieves about 52% of less area and 0.5% of less latency compared to the fastest state-of-the-art design.

Meanwhile, three decimal floating-point fused multiply-add designs have been published. After finishing the fixed-point addition and multiplication, this work discusses such FMA designs. Subsequently, a new technique to improve the performance of the decimal floating-point fused multiply-add is proposed. The fixed-point adder and parallel multiplier were therefore reused and modified in the new DFMA. Applying the specific number system required that the digit-set conversion inside the proposed DFMA be minimized as much as possible. Therefore, only two stages–partial product generation and partial product reduction–are retained in the parallel multiplier. Moreover, the pre-alignment could be further moved out of the critical path by shifting only the addend. The modification of our proposed decimal carry free adder also allows different digit sets on the operands and the result. In the post-alignment unit, the rounding position was decided by detecting the number of leading/trailing zeros and the possible cancellation with simple logics due to the application of the specific number system. Since the digits $\pm(radix - 1)$ do not exist in the proposed digit set, only one digit error may happen in the leading zero detection of the accumulation result. Therefore, the shifting amount decision is relatively simple in the post-alignment unit. Finally, the rounder combined the absolute value conversion, digit-set conversion, and the rounding operation in one carry propagation process. The synthesis result of the Verilog model shows that about 33.7% of delay and 16.6% of area were reduced in comparison to results from the previous fastest designs.

So far, the advantage of unconventional number systems in decimal arithmetic has been exhibited in previous chapters. With the specific redundant number system and the careful hardware design, both processing speed and area efficiency (i.e., hardware cost) of decimal fixed-point addition and multiplication were improved. Furthermore, both the proposed fixed-point functions and the specific number system were applied in order to design a new decimal floating-point fused multiply-add with a better performance. The decimal floating-point arithmetic was therefore enriched by the proposed designs and techniques. In the thesis, the proposed ideas (e.g., two steps non-speculative adder, multiplies without carry generation, hybrid carry propagation, easy leading zero anticipation, and etc.) can be also applied or extended in other non-binary computing systems in order to improve the performance of such systems which are built up with binary devices.

8.2 Future Research

The decimal floating-point fused multiply-add itself is discussed in this thesis. However, the application of such a hardwired design may be a topic for future work. For example, the functional division, square root, reciprocal, and reciprocal square root operations which exploit a series of fused multiply-add operations with Newton's or similar methods could benefit from the proposed DFMA. Both software and hardware solutions are applicable.

In order to improve the area efficiency, the parallel multiplier can be replaced by the sequential design. However, the latency and especially the throughput are going to worsen from this step. Furthermore, if the sequential multiplier is applied, the carry free adder that is currently used in the DFMA may no longer be necessary. The architecture and corresponding algorithms of the DFMA should be changed. The pros and cons of the DFMA architectures with parallel and sequential multipliers suggest another future research topic.

To perform the standalone decimal floating-point addition and multiplication efficiently, the hardware of the proposed DFMA could be optimized. The techniques that have been applied in existing binary design could be exploited in the future.

Alternatively, the similar concept of the unconventional number system applied to improve the performance of the DFMA could be considered for other functions (e.g., sequential division, square root, reciprocal, and even reciprocal square root).

References

[1] J. M. Muller, "On the definition of ulp (x)", URL: http://ljk.imag.fr/membres/Carine.Lucas/TPScilab/JMMuller/ulp-toms.pdf, Last access: May 17, 2013.

[2] H. H. Goldstine and A. Goldstine, "The electronic numerical integrator and computer (ENIAC)", *IEEE Annals of the History of Computing*, vol. 18, no. 1, pp. 10-16, 1996.

[3] G. Gray, "UNIVAC I instruction set", *Unisys History Newsletter*, vol. 5, no. 3, 2001.

[4] M. F. Cowlishaw, "The 'telco' benchmark", URL: http://speleotrove.com/decimal/telco.html, Last access: May 17, 2013.

[5] M. F. Cowlishaw, "Decimal floating-point: Algorism for computers", in *16th IEEE Symposium on Computer Arithmetic*, pp. 104-111, Jun. 2003.

[6] M. F. Cowlishaw, E. M. Schwarz, R. M. Smith, and C. F. Webb, "A Decimal Floating-Point Specification", in *15th IEEE Symposium on Computer Arithmetic*, pp. 147-154, Jun. 2001.

[7] 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 Journal of Research and Development*, vol. 51, no. 1/2, pp. 217-228, Jan.-Mar. 2007.

[8] C. F. Webb, "IBM z10: The next-generation mainframe microprocessor", *IEEE Micro*, vol. 28, no. 2, pp. 19-29, Mar.-Apr. 2008.

[9] E. M. Schwarz, J. Kapernick, and M. Cowlishaw, "Decimal floating-point support on the IBM z10 processor", *IBM Journal of Research and Development*, vol. 53, no. 1, pp. 4:1-4:10, Jan. 2009.

[10] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture", *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 639-662, 2007.

[11] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti, "Design of the POWER6 microprocessor", in *IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 96-97, Feb. 2007.

[12] L. Eisen, J. W. Ward, 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 Journal of Research and Development*, vol. 51, no. 6, pp. 663-684, Nov. 2007.

[13] M. Cornea, "Intel Decimal Floating-Point Math Library", URL: http://software.intel.com/en-us/articles/intel-decimal-floating-point-math-library/, Last access: May 17, 2013.

[14] "Class BigDecimal", URL: http://download.oracle.com/javase/1,5.0/docs/api/java/math/BigDecimal.html, Last access: May 17, 2013.

[15] "Decimal fixed point and floating point arithmetic", URL: http://docs.python.org/library/decimal.html, Last access: May 17, 2013.

[16] "The decNumber Library", URL: http://speleotrove.com/decimal/decnumber.html, Last access: May 17, 2013.

[17] IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-2008, 2008.

[18] L.-K. Wang, C. Tsen, M. J. Schulte, and D. Jhalani, "Benchmarks and Performance Analysis of Decimal Floating-Point Applications", in *25th International Conference on Computer Design*, pp. 164-170, Oct. 2007.

[19] M. Anderson, C. Tsen, L.-K. Wang, K. Compton, and M. J. Schulte, "Performance analysis of decimal floating-point libraries and its impact on decimal hardware and software solutions", in *26th International Conference on Computer Design*, pp. 465-471, Oct. 2009.

[20] 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*, pp. 56-68, Jun. 2007.

[21] L.-K. Wang and M. Schulte, "A decimal floating-point adder with decoded operands and a decimal leading-zero anticipator", in *19th IEEE Symposium on Computer Arithmetic*, pp. 125-134, Jun. 2009.

[22] B. J. Hickmann, A. Krioukov, M. J. Schulte, and M. A. Erle, "A parallel IEEE P754 decimal floating-point multiplier", in *25th International Conference on Computer Design*, Oct. 2007, pp. 296-303.

[23] 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, Jun. 2007.

[24] D. Chen, L. Han, Y. Choi, and S. Ko, "Improved Decimal Floating-Point Logarithmic Converter Based on Selection by Rounding", *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 607-621, May 2012.

[25] A.Vazquez, J. Villalba, and E. Antelo, "Computation of Decimal Transcendental Functions Using the CORDIC Algorithm", in *19th IEEE Symposium on Computer Arithmetic*, pp. 179-186, June 2009.

[26] R. K. Montoye, E. Hokenek, S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit", *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59-70, Jan. 1990.

[27] P. W. Markstein, "Computation of elementary functions on the IBM RISC System/6000 processor", *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 111-119, Jan. 1990.

[28] M. Cornea, J. Harrison, and P. Tang, "Intel Itanium floating-point architecture", *International Symposium On Computer Architecture*, Article 3, 2003.

[29] S. Anderson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo, "RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide", *IBM Corporation, International Technical Support Organization, First edition*, Oct. 1998.

[30] R.M. Jessani, M. Putrino, "Comparison of single- and dual-pass multiply-add fused floating-point units", *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 927-937, Sep. 1998.

[31] P.-M. Seidel, "Multiple path IEEE floating-point fused multiply-add", in *2003 IEEE International Symposium on Micro-NanoMechatronics and Human Science*, vol. 3, pp. 1359-1362, Dec. 2003.

[32] T. Lang and J. D. Bruguera, "Floating-Point Fused Multiply-Add with Reduced Latency", in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 145-150, 2002.

[33] T. Lang and J. D. Bruguera, "Floating-Point Multiply-Add-Fused with Reduced Latency", *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 988-1003, Aug. 2004.

[34] J. D. Bruguera and T. Lang, "Floating-point fused multiply-add: reduced latency for floating-point addition", in *17th IEEE Symposium on Computer Arithmetic*, pp. 42-51, June 2005.

[35] E. Quinnell, E.E. Swartzlander, C. Lemonds, "Floating-Point Fused Multiply-Add Architectures", in *41th Asilomar Conference on Signals, Systems and Computers*, pp. 331-337, Nov. 2007.

[36] E. Quinnell, E.E. Swartzlander, C. Lemonds, "Bridge Floating-Point Fused Multiply-Add Design", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 12, pp. 1727-1731, Dec. 2008.

[37] L. Huang, L. Shen, K. Dai, and Z. Wang, "A New Architecture For Multiple-Precision Floating-Point Multiply-Add Fused Unit Design", in *18th IEEE Symposium on Computer Arithmetic*, pp. 69-76, Jun. 2007.

[38] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, "Low Cost Binary128 Floating-Point FMA Unit Design with SIMD Support", *IEEE Transactions on Computers*, Apr. 2011.

[39] J.-M. Muller et al., *Handbook of Floating-point Arithmetic*, ISBN 978-0-8176-4704-9, Boston : Birkhuser, 2010.

[40] P.-M. Seidel, G. Even, "On the design of fast IEEE floating-point adders", in *15th IEEE Symposium on Computer Arithmetic*, pp. 184-194, Jun. 2001.

[41] P. K. Monsson, *Combined Binary and Decimal Floating-point Unit*, Master Thesis, Dept. of Information and Mathematical Modeling, Technical University of Denmark, Aug. 2008.

[42] R. D. Kenney, M. J. Schulte, and M. A. Erle, "A high-frequency decimal multiplier", in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 26-29, Oct. 2004.

[43] H. He, Z. Li, and Y. Sun, "Multiply-add fused float point unit with on-fly denormalized number processing", in *48th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 1466-1468, Aug. 2005.

[44] W. Kahan, *Check Whether Floating-Point Division Is Correctly Rounded*, monograph, Dept. of Computer Science, University of California, Berkeley, 1956.

[45] F. G. Gustavson, J. E. Moreira, and R. F. Enenkel, "The fused multiply-add instruction leads to algorithms for extended-precision floating point: applications to java and high-performance computing", in *1999 conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[46] R.C. Agarwal, F.G. Gustavson, and M.S. Schmookler, "Series approximation methods for divide and square root in the Power3$^{\text{TM}}$ processor", in *14th IEEE Symposium on Computer Arithmetic*, pp. 116-123, 1999.

[47] I. Koren, *Computer Arithmetic Algorithms, 2nd Edition*, ISBN 9781568811604, A. K. Peters, 2002.

[48] M. Ercegovac and T. Lang, *Digital Arithmetic*, ISBN 1558607986, Elsevier Science (USA), 2004.

[49] B. Parhami, *Computer Arithmetic - Algorithms and Hardware designs*, ISBN 0195125835, Oxford University Press, 2004.

[50] A. Svoboda, "Decimal adder with signed digit arithmetic", *IEEE Transactions on Computers*, C-18(3), pp. 212-215, 1969.

[51] H. Nikmehr, B. Phillips and C.C. Lim, "A decimal carry-free adder", in *SPIE conference on Smart Materials, Nano-, Micro-Smart Systems,* pp. 786-797, 2004.

[52] A. Kaivani and G. Jaberipur, "Fully redundant decimal addition and subtraction using stored-unibit encoding", *Integration, the VLSI journal*, pp. 34-41, 2010.

[53] H. Fahmy and M.J. Flynn, "The case for a redundant format in floating-point aritmetic", in *16th IEEE Symposium on Computer Arithmetic,* pp. 95-102, June 2003.

[54] G. Jaberipur and M. Ghodsi, "High Radix Signed Digit Number Systems: Representation Paradigms", *Scientia Iranica* , 10(4), pp. 383-391, 2003.

[55] G. Jaberipur and S. Gorgin, "A Nonspeculative Maximally Redundant Signed Digit Adder", *The 13th international CSI Computer Conference*, pp. 235-242, 2008.

[56] John Moskal, Erdal Oruklu and Jafar Saniie, "Design and Synthesis of a Carry-Free Signed-Digit Decimal Adder", *IEEE International Symposium on Circuits and Systems*, pp. 1089-1092, 2007.

[57] L-K. Wang, M. A. Erle, C. Tsen, E. M. Schwarz, M. J. Schulte, "A survey of hardware designs for decimal arithmetic", *IBM Journal of Research and Development*, vol. 54, no. 2, Mar. 2010.

[58] 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, Mar. 2009.

[59] B. Shirazi, D. Yun, C. N. Zhang, "RBCD: redundant binary coded decimal adder", in *IEE Proceedings*, vol. 136, no. 2, March 1989.

[60] F. Y. Busaba et al., "The IBM z900 Decimal Arithmetic Unit", in *Conference Record of the Thirty-Fifth Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1335-1339, 2001.

[61] E. M. Schwarz, J. S. Kapernick, and M. F Cowlishaw, "Decimal floating-point support on the IBM System z10 processor", *IBM Journal of Research and Development*, vol. 53, no. 1, pp. 4:1-4:10, Apr. 2010.

[62] M. Cornea et al., "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, 2009.

[63] M. A. Erle and M. J. Schulte, "Decimal Multiplication Via Carry-Save Addition", in *IEEE International Conference on Application Specific systems, Architectures, and Processors*, pp. 348-358, Jun. 2003.

[64] M. A. Erle, E. M. Schwarz, and M. J. Schulte, "Decimal multiplication with efficient partial product generation", in *17th IEEE Symposium on Computer Arithmetic*, pp. 21-28, 2005.

[65] T. Lang and A. Nannarelli, "A Radix-10 Combinational Multiplier", in *40th Asilomar Conference on Signals, Systems and Computers*, pp. 313-317, Oct. 2006.

[66] L. Dadda and A. Nannarelli, "A Variant of a Radix-10 Combinational Multiplier", in *IEEE International Symposium in Circuits and Systems (ISCAS 2008)*, pp. 3370-3373, May 2008.

[67] 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.

[68] 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, May 2010.

[69] I. D. Castellanos and J. E. Stine, "Decimal partial product generation architectures", in *51st Midwest Symposium on Circuits and Systems*, pp. 962-965, Aug. 2008.

[70] S. Gorgin and G. Jaberipur, "A fully redundant decimal adder and its application in parallel decimal multipliers", *Microelectronics Journal*, vol. 40, no. 10, Oct. 2009.

[71] L. Dadda, "Multioperand Parallel Decimal Adder: a mixed Binary and BCD Approach", *IEEE Transactions on Computers*, vol. 56, pp. 1320-1328, Oct. 2007.

[72] I. D. Castellanos and J. E. Stine, "Compressor Trees for Decimal Partial Product Reduction", in *18th ACM Great Lakes Symposium on VLSI*, pp. 107-110, May 2008.

[73] 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*, pp. 25-27, 2007.

[74] M. A. Erle, B. J. Hickmann, and M. A. Schulte, "Decimal Floating-Point Multiplication", *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 902-916, Jul. 2009.

[75] A. Vázquez and E. Antelo, "Conditional Speculative Decimal Addition", in *7th Conference on Real Numbers and Computers (RNC 7)*, pp. 47-57, Jul. 2006.

[76] A. Vázquez, E. Antelo, and P. Montuschi, "A New Family of High-Performance Parallel Decimal Multipliers", in *18th IEEE Symposium on Computer Arithmetic*, pp. 195-204, June 2007.

[77] C. H. Chang, J. Gu, and M. Zhang, "Ultra low-voltage low-power CMOS 4-2 and 5-2 compressors for fast arithmetic circuits", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 10, pp. 1985-1997, 2004.

[78] G. Jaberipur and B. Parhami, "Constant-time addition with hybrid-redundant numbers: Theory and implementations", *Integration, the VLSI journal*, vol. 41, pp. 49-64, 2008.

[79] T. Aoki et al., "Signed-weight arithmetic and its application to a field-programmable digital filter architecture", *IEICE Transactions on Electronics*, vol. E82-C, no.9, pp. 1687-1698, 1999.

[80] P. Kornerup, "Reviewing 4-to-2 Adders for Multi-Operand Addition", *Journal of VLSI Signal Processing*, vol. 40, pp. 143-152, 2005.

[81] Decimal IP, SilMinds, URL:
http://www.silminds.com/decimal-products, Last access: May 17, 2013.

[82] GNU C compiler library URL:
http://gcc.gnu.org/onlinedocs/gcc/Decimal-Float.html, Last access: May 17, 2013.

[83] J. Thompson, M. J. Schulte, and N. Karra, "A 64-bit decimal floating-point adder", in *IEEE Computer society Annual Symposium on VLSI*, pp. 297-298, Feb. 2004.

[84] A. Vazquez and E. Antelo, "A high-performance significand BCD adder with IEEE 754-2008 decimal rounding", in *19th IEEE Symposium on Computer Arithmetic*, pp. 135-144, Jun. 2009.

[85] S. Gorgin and G. Jaberipur, "Fully redundant decimal arithmetic", in *19th IEEE Symposium on Computer Arithmetic*, pp. 145-152, Jun. 2009.

[86] 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.

[87] L.-K. Wang and M. J. Schulte, "A decimal floating-point divider using Newton-Raphson iteration", *Journal of VLSI Signal Processing Systems,* vol. 49, no. 1, pp. 3-18, Oct. 2007.

[88] L.-K. Wang and M. J. Schulte, "Decimal Floating-Point Square Root Using Newton-Raphson Iteration", in *16th IEEE International Conference of Application-Specific Systems, Architectures and Processors*, 2005.

[89] A. Vázquez, E. Antelo, and P. Montuschi,"A radix-10 SRT divider based on alternative BCD codings", in *IEEE International Conference on Computer Design*, pp. 280-287, Oct. 2007.

[90] R. C. Agarwal, F. G. Gustavson, and M. S. Schmookler, "Series approximation methods for divide and square root in the Power3™ processor", in *14th IEEE Symposium on Computer Arithmetic*, pp. 116-123, 1999.

[91] R. M. Jessani and M. Putrino, "Comparison of single- and dual-pass multiply-add fused floating-point units", *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 927-937, Sep. 1998.

[92] J. D. Bruguera and T. Lang, "Floating-point fused multiply-add: reduced latency for floating-point addition", in *17th IEEE Symposium on Computer Arithmetic*, pp. 42-51, June 2005.

[93] R. Samy, H. A. H. Fahmy, R. Raafat, A. Mohamed, T. ElDeeb, and Y. Farouk, "A decimal floating-point fused-multiply-add unit", in *53rd IEEE International Midwest Symposium on Circuits and Systems*, pp. 529-532, Aug. 2010.

[94] R. Raafat, A. M. Abdel-Maheed, R. Samy, T. ElDeeb, Y. Farouk, M. Elkhouly, and H. A. H. Fahmy, "A decimal fully parallel and pipelined floating point multiplier", in *42 Asilomar Conference on Signals, Systems, and Computers, Asilomar*, Oct. 2008.

[95] A. Akkas and M. J. Schulte, "A decimal floating-point fused multiply-add unit with a novel decimal leading-zero anticipator", in *22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, Sep. 2011.

[96] L. Han and S. Ko, "High Speed Parallel Decimal Multiplication with Redundant Internal Encodings", *IEEE Transactions on Computers*, vol. 62, no. 5, pp. 956-968, May 2013.

[97] L. Han, D. Chen, K. A. Wahid, and S. Ko, "Nonspeculative decimal signed digit adder", in *2011 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1053-1056, May 2011.

[98] J. D. Bruguera and T. Lang, "Leading-one prediction with concurrent position correction", *IEEE Transactions on Computers*, vol. 48, no. 10, Oct. 1999.

[99] M. Cowlishaw, "Decimal library Performance v1.12", URL: http://speleotrove.com/decimal/decperf.pdf, Last access: May 17, 2013.

[100] STMicroelectronics, 90nm CMOS Design Platform, 2007.

[101] A. S. Ahmed and H. A. H. Fahmy, "2010_07_d64_fma.zip", URL: http://eece.cu.edu.eg/∼hfahmy/arith_debug/#vectors, Last access: May 17, 2013.

[102] A. EITantawy, *Decimal floating point arithmetic unit based on a fused multiply add module*, MS.c. dissertation, Electronics and Electrical Communications Engineering Department of Cairo University, 2011.

[103] A. Vázquez, *High-performance decimal floating-point units*, Ph.D. dissertation, Electronics and Computer Engineering Department of University of Santiago de Compostela, 2009.

[104] H. A. H. Fahmy, *A Redundant Digit Floating Point System*, Ph.D. dissertation, Stanford University, June 2003.