Design And Implementation of a Radix-100 Division Unit

A Thesis Submitted
to the College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Electrical and Computer Engineering
University of Saskatchewan

by

Zhuo Wang

Saskatoon, Saskatchewan, Canada

© Copyright Zhuo Wang, August, 2012. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, it is agreed that the Libraries of this University may make it freely available for inspection. Permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professors who supervised this thesis work or, in their absence, by the Head of the Department of Electrical and Computer Engineering or the Dean of the College of Graduate Studies and Research at the University of Saskatchewan. Any copying, publication, or use of this thesis, or parts thereof, for financial gain without the written permission of the author is strictly prohibited. Proper recognition shall be given to the author and to the University of Saskatchewan in any scholarly use which may be made of any material in this thesis.

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

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

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. Seok-Bum Ko, for all his help. He offered me his outstanding knowledge, experience, advices, and patience without which the thesis would not have been possible. In addition, Dr. Ko provided many advices for my life and career, which helped knowing my future much better.

Also, I appreciate the generous help from Liu Han who works in the same lab with me. It is him who helped me avoiding many detours. In addition, I would like to thank all the friends working in 2C60 for all their friendship and help.

I would also like to thank Dr. Ron Bolton, Dr. Anh van Dinh, Dr. Aryan Saadat Mehr in the Department of Electrical and Computer Engineering for reading my thesis and providing comments. And I would like to thank Dr. Li Chen for his help and advices during my graduate studies.

Finally, I would like to thank family for their understanding, constant support and encourage throughout my life.

Abstract

Nowadays, DFP (Decimal Floating-point) is widely used in financial fields such as tax calculation, currency conversion and other areas where precise arithmetic is needed. Binary arithmetic, although widely used in current ALU (Arithmetic Logic Unit)s, has some limitations when performing correct decimal arithmetic. Consequently, DFU has drawn more and more attention in recent years. Due to the increasing demands for DFUs, IEEE 754-2008 formally defines three decimal DFU formats for both industry and research areas.

To perform DFP arithmetic, hardware implemented DFUs are the trend in industry. IBM announced Power6, which fully supports IEEE 754-2008 standard, in the year of 2007. But that microprocessor is mainly designed for high-end computers. More effort should be made on the spread of DFUs.

In this thesis, a hardware based radix-100 divider is designed and implemented. Instead of using popular SRT (Sweeney, Robertson, and Tocher) division algorithm, selection by truncation algorithm is utilized. As a high-radix decimal divider, radix-100 divider can generate two quotient digits in each iteration. This is the major advantage of high-radix decimal divider compared to the decimal dividers. Besides, a compensation method is utilized to reduce the cycle time and the time consumed on the "multiples selection" module. Decimal carry-save adders and decimal carry-propagate adders are reused to reduce the overall area.

The radix-100 divider is proven to be faster (3%) than the current decimal dividers, although the ratio is not outstanding. Meanwhile, the radix-100 divider consumes a larger area than the decimal dividers. It is expected to be a good start for the radix-100 divider. By applying more techniques in the future, the performance (latency) of the radix-100 divider is very likely to be much better than the decimal dividers.

Table of Contents

Permission to Use i

Acknowledgments ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

List of Abbreviations ix

1 Introduction 1

1.1 Why Decimal? 2

1.2 The Evaluation of Decimal Arithmetic 4

1.3 IEEE 754-2008 Standard 5

1.3.1 Decimal Formats and Encoding 6

1.3.2 Decimal Rounding 7

1.3.3 Exception Handling 8

1.4 Radix-100 Division 10

1.5 Binary Division Algorithms 11

1.5.1 Binary Non-restoring Method 12

1.5.2 Binary SRT Method 13

1.5.3 High-radix Division 13

1.6 Contribution 16

2 Previous Work 18

2.1 Typical SRT-based Decimal Divider 18

2.2 Non-restoring based Decimal Divider 22

2.3 Decimal Divider with Different Encodings 25

2.4 Radix-16 Dividers 28

3 Algorithm 31

3.1 Decimal Floating-point Division 31

3.2 Decimal Representations 33

3.3 Proposed Algorithm 35

3.3.1 Pre-scaling Parameters 37

3.3.2 Pre-scaling 40

3.3.3 Quotient Selection 42

4 Architecture 45

4.1 Major Blocks 45

4.1.1 10's Complement 45

4.1.2 Doubler & Quintupler 46

4.1.3 BCD DCSA 47

4.2 Pre-scaling 48

4.3 Iteration 51

4.3.1 Digit Recognition 51

4.3.2 Multiples Selection 54
4.3.3 Addition 57
4.3.4 Reuse Consideration 58
4.3.5 On-the-fly Conversion 60
4.3.6 Rounding 61
4.4 Operation Sequence 63
5 Implementation and Comparison 65
5.1 Synthesis Results 65
5.2 Comparison 67
5.2.1 Pre-calculation 67
5.2.2 Quotient Selection 68
5.2.3 Addition 69
5.2.4 Area 69
5.3 Conclusion 70
6 Summary 71
7 Future Work 74
References 76

List of Tables

1.1 Database statistic results 2

1.2 Example of a binary implemented decimal tax calculation 3

1.3 Decimal formats defined in IEEE 754-2008 6

2.1 Different encodings and their efficiency 25

3.1 Pre-scaling parameter comparison 38

3.2 Parameter exceptions 39

3.3 Generation of D_m 40

3.4 Parameter's range 40

3.5 Needs of compensations 43

4.1 9's complement table 46

4.2 Algorithm for the quintupler 47

4.3 Prediction of sign 53

4.4 $q_H D'$ and the corresponding compensation selection 54

4.5 $q_{L1} D'$, $q_{L2} D'$ and the corresponding compensation selection 55

4.6 Addends 57

4.7 Rounding rules 62

5.1 Critical Path 66

5.2 Results Comparison 67

List of Figures

1.1 P-D plot 16

2.1 Overlapping quotient selection 28

2.2 Cascaded radix-16 divider 30

3.1 Architecture of IEEE 754-2008 divider 32

4.1 Architecture of the pre-scaling module 49

4.2 Architecture of the iteration module 51

List of Abbreviations

ALU	Arithmetic Logic Unit
BCD	Binary Coded Decimal
BID	Binary Integer Decimal
CPA	Carry-propagate adder
CSA	Carry-save adder
DCSA	Decimal Carry-save adder
DCPA	Decimal Carry-propagate adder
DFU	Decimal Floating-point Unit
DFP	Decimal Floating-point
DPD	Densely Packed Decimal
LSB	Least Significant Bit
LSD	Least Significant Digit
MSB	Most Significant Bit
MSD	Most Significant Digit
SRT	Sweeney, Robertson, and Tocher division algorithm
FO4	Fan-out of 4
MUX	Multiplexer

1. Introduction

Decimal arithmetic is the most natural arithmetic system in human calculations. In fact, during the early days of the computer era, many computers, such as ENIAC [1] and IBM 650 [2] used decimal arithmetic. However, the fundamental units of memory and flip-flops are naturally binary elements and the representation of decimal digit, which takes at least 4 binary bits, consumes much more storage than the binary based system [3]. What's more, the binary system was much simpler and faster than the decimal system at that time [3]. Consequently, binary systems dominate the current world and few computer systems include the decimal arithmetic unit.

Until now, full IEEE 754-2008 supportive DFUs were limited on some certain high-end microprocessors [4], for example, IBM POWER6 [5], IBM mainframe Z9 (with assistance) [6], and IBM mainframe Z10 [7], while in most other microprocessors, decimal operations were based on software libraries or integer decimal arithmetic units (IBM Z900 [8]).

Due to the booming commercial needs, decimal arithmetic has drawn more and more attention. Hardware implemented DFU is a trend and it is possible that, just like the popularization of the binary floating-point units, the DFU will be a standard block in low-end microprocessors in a few years [4].

To make the DFU more widely used in the future, many companies and researchers tried their best to improve the performance of the DFU. However, efforts focusing on the four basic operations (addition, subtraction, multiplication and division) are not the same. Division got the least amount of attention. However, the importance of division should not be underestimated [9]. Consequently, in this thesis, we focus on

the decimal division field and try to apply high-radix to the decimal field.

In Section 1.1, the reasons why decimal arithmetic is important are introduced. The history and evaluation of decimal arithmetic will be covered in Section 1.2. IEEE 754-2008 standard is explained in Section 1.3 followed by a brief introduction of the proposed divider. Finally, some binary division methods are included in Section 1.5 since high-radix division is also based on those fundamental methods.

1.1 Why Decimal?

Although binary arithmetic is the major workforce in ALU, decimal calculation can never be avoided. First of all, it is proven that decimal numbers are quite widely used [10]. A database owned by 51 major organizations, which includes Banking, Airline, Financial Analysis, Insurance, Inventory control, Management reporting, Marketing services and so on, was analyzed. There are totally 456,420 columns of data used to get the statistic information, and the results are shown in Table 1.1.

Table 1.1 Database statistic results

Data Type	$No.of columns$	$Percent$
Decimal	251,038	55.0
Integer	78,842	17.3
SmallInt	120,464	26.4
Float	6,180	1.4

From table 1.1, it is notable that 55% of the total analyzed data are decimal. What's more, another 43.7% are integers which are normally held in the form of decimal numbers.

Another essential reason for using the decimal system is that we have been used to the decimal arithmetic ever since the numbers were invented by our ancestors. Although binary can represent every decimal integer number, it cannot represent most of decimal fractions. The value 0.1, for example, results in an infinitely repeating binary number. Other than that, if we use binary to represent a certain decimal

fraction, a notable difference might happen. For instance, the calculation of the total amount of money (including 5% tax) which should be charged on a phone call cost $0.70 will result in different bills in decimal and binary systems. The calculation and comparison is shown in Table 1.2.

Table 1.2 Example of a binary implemented decimal tax calculation

Number system	Cost	Including Tax	Calculation result	Rounding
Decimal	0.7	1.05	0.735	0.74
Binary	0.6999999	1.05	0.734999999	0.73

Notice that this difference is not because of rounding. In fact, the cause of this issue is the lack of binary representations [4], so that 0.7 cannot be represented properly in binary. This one cent difference will be accumulated to quite a large amount of money if this kind of calculation is used by a big telecom company. Consequently, binary arithmetic cannot be used in many commercial fields such as banking, accounting, tax calculation, insurance, currency conversion and so on. These areas almost touch everyone's life so no fault can be tolerated.

Another issue caused by binary arithmetic is the removal of trailing fraction zeros. For example, there is no way for the binary system to distinguish 2.4 and 2.40 since the current binary format doesn't support the trailing fraction zeros. However, decimal system can easily keep those trailing fraction zeros. Trailing fraction zeros may seem meaningless, but they are essential because of the following reasons [11]:

1. Users expect the trailing fraction zeros after some certain arithmetic operations such as addition and subtraction.

2. Sometimes currency calculation needs a full width of digits. For example, in European regulations, the exchange rates have to be represented in 6 digits, even when there are trailing fraction zeros. For instance, 1 EURO should equal to 340.750 Greek drachmas (Only for instance, the current ratio may be different).

3. Unit depends on the LSD (least significant digit) of a sequence of numbers. If

the trailing fraction zeros are removed, the unit will be changed. For example, there is a difference between "using a 1.2 meter steel" and "using a 1.200 meter steel" to construct a certain part of a house.

Overall, the trailing fraction zeros are essential in the calculation, which is another reason for using decimal arithmetic. From these three aspects, it is certainly needed to figure out a way to deal with decimal calculations. In fact, decimal arithmetic is not a brand-new topic. There are several ways of performing decimal arithmetic. They are introduced in Section 1.2.

1.2 The Evaluation of Decimal Arithmetic

Efforts have been made to deal with decimal arithmetic. Before decimal systems were implemented in hardware, there were two ways to calculate decimal numbers [12]. The easier way is to convert decimal numbers to binary numbers, then use the built-in binary arithmetic unit to finish the operation before converting the results back to decimal numbers. However, the major technique behind this method is still binary arithmetic, so the issues caused by binary arithmetic are still there. Another way is to keep the decimal numbers in their original format while using software libraries to do the decimal calculation. This method, which includes Sun BigDecimal for Java 5 and IBM decNumber library for ANSI C and C++, is widely used. But the performance of those softwares are unacceptable. In some applications, decimal calculations can take almost 90% of the total workload [11]. In fact, it is proven that the software based operation is 100 to 1000 times slower than the DFU implemented in hardware [12].

Due to the common complaints of the performance of software-based decimal arithmetic, many companies have tried to implement hardware based decimal systems. The IBM z900 is one of the earliest mainframe microprocessors that includes decimal arithmetic unit [8]. However, this type of decimal arithmetic is limited on decimal integers. Despite the fact that this decimal arithmetic unit can improve the performance of decimal fixed-point calculations, it requires manual scaling which is error-prone and hard to use [10].

The ideal way of doing decimal arithmetic is to execute decimal operations on exact arithmetic. The exact arithmetic will hold every single bit generated by the arithmetic unit. For example, multiplication will probably need double length of each operand. However, the commercial calculations have become more and more complex so it is impossible to let the computer deal with the exact arithmetic since the memory storing those commercial numbers will run out eventually [10].

Another restriction of decimal fixed-point and integer arithmetic is "rounding". Most of the fixed-point arithmetic operations do not require rounding. However, rounding is required by many financial and other applications, so fixed-point and integer arithmetics are not capable of dealing with calculations happened in those areas [4].

The decimal floating-point can be used to avoid any inaccuracy seen in the binary floating-point, and it can extend the decimal fixed-point and the integer arithmetic [4]. Therefore, decimal floating-point arithmetic should dominate the industry in the future. IBM has shown strong interests in the decimal floating-point area. In the year of 2007, IBM announced the POWER6 microprocessor, which brought the microprocessor to the decimal floating-point era. It fully supports IEEE 754-2008 standard and includes a DFU in each core [5].

1.3 IEEE 754-2008 Standard

Without a world-accepted format, it would be hard for the decimal floating-point arithmetic to be widely used. Before the establishment of IEEE 854-1987 (withdrawn by IEEE in 2008), there were no standards for floating-point numbers so each application had its own format. However, IEEE 854-1987 was an radix independent floating-point standard, it is mainly designed for scientific and engineering uses instead of commercial uses [4]. Consequently, this standard cannot be adapted to commercial needs. On the other hand, another standard, IEEE 754-1985, is a binary floating-point standard so it cannot be used for decimal arithmetic. Under these circumstances, IEEE published the IEEE 754-2008 standard (previously known as

IEEE 754r), which is derived from IEEE 854-1987 and IEEE 754-1985, in August 2008. IEEE 754-2008 includes almost all of the standards defined in its predecessors [13].

IEEE 754-2008 mainly defines several standards for both binary and decimal floating-point data, including formats, encodings, rounding rules, operations and some exception handling [14]. In the following subsections, the rules of the decimal arithmetic are introduced.

1.3.1 Decimal Formats and Encoding

Basic format

The standard defines three decimal formats, which are decimal 32, decimal 64 and decimal 128, as seen in Table 1.3.

Table 1.3 Decimal formats defined in IEEE 754-2008

Decimal format	*Digits*	E_{max}	E_{min}
Decimal 32	7	+96	-95
Decimal 64	16	+384	-383
Decimal 32	34	+6144	-6143

The representation of decimal floating-point number is similar to the binary floating-point number except that its base is 10, as shown in the following formula:

$$(-1)^s \times C \times 10^q \tag{1.1}$$

Where s is the sign, which can be zero or one, C is the significant value, or coefficient, and q is the exponent. There are two restrictions on C and q [13]:

1. C must be an unsigned integer between 0 and $10 \times p - 1$, where p is the number of digits in each decimal format (e.g., if p=7 then C will not be larger than 9999999). Notice that in the standard, C is not required to be normalized. Although normalized C can save some storage and bring advantage to the calculations in binary floating-point formats, it makes no sense in the decimal field. What's more, as we discussed in Section 1.1, it is preferred to keep the non-normalized decimal numbers.

2. q must be an integer satisfying this equation: $1 - E_{max} \leq q + p - 1 \leq E_{max}$. For example, q ranges from -101 to 90 in decimal 32 format.

Extended precision formats

In some certain cases, the formats provided in the standard might be insufficient. In the extended precision formats, the precision or the range can be extended.

There are two kinds of encoding defined in the standard. DPD, which is short for Densely Packed Decimal, is proposed by IBM. It can fit three decimal digits into 10 bits [15]. Compared with BCD, which use 4 bits to represent each digit, the DPD introduces a significant reduction in terms of storage and bandwidth. However, since DPD is quite a compact format, it is not convenient to use DPD directly in the arithmetic implementations [4]. So some compact representations such as BCD, is easier to handle. Due to the low cost (two or three gates delay [16]) in conversion between DPD and BCD, the DPD stored BCD operation is possible and utilized in POWER 6 [17].

Another encoding method is BID (Binary Integer Decimal) proposed by Intel. This format is designed mainly for software implementation [4] to avoid the penalties caused by the conversion between DPD and BCD [18]. Both encoding cases apply to all decimal formats.

1.3.2 Decimal Rounding

As discussed in Section 1.2, rounding is one of the reasons why decimal floating-point units will replace the decimal integer or fixed-point units. Operations like multiplication will consume a significant amount of storage. Under the IEEE 754-2008 standard, the decimal operations have to be followed by a rounding step.

roundTiesToEven

Round the operation result to the nearest value. If the halfway condition happens, it is rounded to the nearest number with an even least significant bit.

roundTiesToAway

Round the operation result to the nearest value. If the halfway condition happens, it is rounded to the number with larger magnitude.

roundTowardsPositive

Also named as rounding up or ceiling. In this rounding scheme, the operation result is rounded to the positive infinite. The closest number (greater than the exact result) which satisfies the decimal format should be used as the rounding result.

roundTowardsNegative

Also named as rounding down or floor. In this rounding scheme, the operation result is rounded to the negative infinite. The closest number (lower than the exact result) which satisfies the decimal format should be used as the rounding result.

roundTowardsZero

Also named as truncation. In this rounding scheme, the operation result is truncated. The closest number (lower in magnitude) which satisfies the decimal format should be used as the rounding result.

Among the five rounding schemes mentioned above, roundTiesToEven is recommended since the result is rounded up and down alternately. This feature will avoid keeping rounding up or down which may result in a positive or negative bias [19].

Also, there are three other rounding modes, which, though not defined in the IEEE standard, are used occasionally in DFU implementations [4]. They are roundingTiesZero, roundAwayZero and roundToVariablePrecision.

1.3.3 Exception Handling

The standard specifies five different kinds of exceptions. These exceptions happens when the result is not the expected floating-point number. To deal with these exceptions, the normal logic which is used to deal with meaningful decimal floating-

point numbers will be bypassed and the default nonstop exception handling will set a status flag to indicate which exception has been triggered. The occurrence of these exceptions always has something to do with the following numbers: ± 0, $\pm\infty$, and not-a-number (NaN) [20]. Some of these numbers have different encoding modes than those mentioned above, but they are not within our interests.

Invalid Operation

This happens when the result of a certain operation is not defined. This might be caused by some invalid operands, for instance, computation with NaN, square-root of negative numbers and so on. The default result is a qNaN with other information.

Division by zero

As can be explained by the title, when the divisor is 0 in a division operation, this exception will be triggered. The result will be set as a signed ∞.

Overflow

It is quite possible that after some arithmetic operations, the result exceeds the standard's maximum supportive number. If this happens, there are several possible results, which are plus or minus infinity or the maximum representable positive number in the corresponding format. This depends on the rounding mode.

Underflow

Underflow happens when the magnitude of a result is below the smallest representable number in the corresponding format. In this case, the result can be zero, or a subnormal number, or the positive or negative minimum number in the current decimal format.

Inexact

This status flag will be raised if the correct rounded results is different from the infinite precision. Then, the default result would be the rounded one or the overflow result.

Notice that these five operations have nothing to do with the main logic, which is used to calculate normal numbers. They are performed in some side logics which will not influence the overall performance. Consequently, they will not be touched in the algorithm discussed in this thesis.

1.4 Radix-100 Division

Ever since DFU became a popular topic in industry as well as the research area, decimal floating-point adders and multipliers have drawn most of the attention. The reason is that division is believed to be a rare operation hence less effort has been made [21]. However, it is true that among the four basic operations (addition, subtraction, multiplication and division), division, although is a infrequently used operation, is the most complex and time-consuming calculation. Proven in [9], underestimating the division implementation will result in a degradation of system performance. This fact reveals the incentive to design high-speed division algorithms so as to enhance the performance of arithmetic processors.

Concluded and compared in [21], division algorithms can be divided into five classes, which are "digit recurrence, functional iteration, very high radix, table lookup, and variable latency" [21]. However, it doesn't mean each DFU can only follow one of these methods, instead, multiple methods are used in each DFU. For instance, a functional iteration based division unit can use a look-up table to get the approximate initial reciprocal, then use the functional iteration to get the quotients, then use the variable latency methods at the end of the calculation [21].

In terms of the popular methods used in the area of decimal division, the SRT algorithm (a digit recurrence method which is named after Sweendy, Roberton, and Tocher [19]) is the most widely used technique. This is supported by some published papers in recent years [22] [23] [24] [25] [26]. It is also proved that SRT would be applied to decimal field [22]. Other implementations are [27], which is based on Newton-Raphson algorithm (A typical functional iteration method), and [17], which utilizes selection by truncation method. The latest radix-10 divider using SRT has

reached a great performance [26].

However, there is a limitation of improvement inside the radix-10 algorithm. As a way of improving the performance of decimal dividers, high-radix decimal division algorithms can be considered. This is utilized widely in the area of binary division [28] [29] [30] [31]. We also focus on high-radix decimal division method (radix-100) in this thesis to try to ameliorate the slow decimal division operation.

However, the well-known SRT algorithm is not suitable for the radix-100 division. The main reason is that the number of potential quotients is tremendous. SRT needs the same number of comparisons as the number of quotients before getting the final quotient. This is unrealistic for such a high-radix division. In the binary field, overlapping [29] and cascading [30] methods are developed for high-radix implementation. However, neither method has a notable advantage in terms of area and latency in the usage of radix-100 division.

Theoretically, higher radix would consume higher area to reach a good performance. As concluded from several recent papers, reduction of overall latency is the major aspect in the research area. A non-restoring method combined with selection by truncation method [17] has drawn our attention. Although performing moderately in the field of decimal division, it has an inherent advantage which is a simple quotient digit selection. By adopting that algorithm in radix-100 divider, a two-digit quotient can be selected by simple combinational logics at the same time. Although selection by truncation takes more time in the pre-scaling step compared with SRT, iteration cycles are reduced by half compared to radix-10 dividers, which can save a great amount of time in the end.

Details of the algorithm will be explained in Chapter 3.

1.5 Binary Division Algorithms

Although decimal division may seem like a different field to the binary division field, the basic methods used in the decimal field are derived from the binary field.

Here, we only concisely introduce the non-restoring and the SRT methods whose details can be found in the book [19].

1.5.1 Binary Non-restoring Method

The basic idea of the non-restoring method is that the remainder can be negative while the quotient obtained in each iteration might be incorrect. These incorrect quotient digits can be modified by next quotient digits. Assuming R_i is the partial remainder obtained after the ith iteration. The basic formula in binary division is shown in equation (1.2).

$$R_i = 2 \times R_{i-1} - q_i \times D \tag{1.2}$$

The selection of the q_i follows

$$q_i = \begin{cases} 1 & \text{if } 2 \times R_{i-1} \geq 0 \\ -1 & \text{if } 2 \times R_{i-1} \leq 0 \end{cases} \tag{1.3}$$

One of the reasons why the non-restoring method is faster than the restoring method is that it compares the partial remainder with 0 instead of D. In hardware, the logic performing non-zero checking is much simpler than that used to do comparison between two data.

In the restoring algorithm, if $2 \times R_{i-1} - D \leq 0$, the subtraction will be cancelled and the remainder would be $2 \times R_{i-1}$. In the next iteration, the remainder is left shifted by one bit and D is subtracted from the remainder again. After these two iterations, the partial remainder should be $4 \times R_{i-1} - D$. In the non-restoring method, the D is subtracted regardless of the sign of the partial remainder. In the next iteration, since the previous partial remainder is negative, D should be added. Consequently, the result is $2 \times (2 \times R_{i-1} - D) + D = 4 \times R_{i-1} - D$. By performing this kind of correction, the result of the non-restoring method will be the same with that of restoring algorithm.

1.5.2 Binary SRT Method

Notice that the non-restoring division needs to perform addition or subtraction in every iteration, which can be sped up by using SRT division [19]. The main improvement of SRT is the introduction of quotient 0 and the definition of boundary. The selection of q_i can be changed to equation (1.4)

$$q_i = \begin{cases} 1 & \text{if } 2 \times R_{r-1} \geq D \\ 0 & \text{if } -D \leq 2 \times R_{r-1} \leq D \\ -1 & \text{if } 2 \times R_{r-1} \leq -D \end{cases} \tag{1.4}$$

The calculation of remainder still follows equation (1.2).

This requires the comparison between the partial remainder and the divisor, which, as discussed before, is time-consuming in hardware implementation. Consequently, if the divisor can be normalized to a certain range, such as $\frac{1}{2} \leq |D| \leq 1$, the partial remainder can be compared with either $\frac{1}{2}$ or $-\frac{1}{2}$ instead of $|D|$, which is obviously easier to implement. Notice that $\frac{1}{2}$ is 0.1 in binary.

Now, the selection of q_i becomes equation (1.5). This kind of division algorithm is called SRT.

$$q_i = \begin{cases} 1 & \text{if } 2 \times R_{r-1} \geq \frac{1}{2} \\ 0 & \text{if } -\frac{1}{2} \leq 2 \times R_{r-1} \leq \frac{1}{2} \\ -1 & \text{if } 2 \times R_{r-1} \leq -\frac{1}{2} \end{cases} \tag{1.5}$$

Notice that as long as we follow equation (1.5), the new partial remainder should always be smaller than $|\frac{1}{2}|$, which is smaller than or equal to $|D|$. Only if that convergence requirement is satisfied can SRT work properly. Imagine if one partial remainder was larger than $|D|$, the next partial remainder $2 \times R_i - D$ would still be larger than $|D|$. This would continue so the division would never end.

1.5.3 High-radix Division

As discussed in Section 1.4, high-radix division is an efficient way to improve the performance of division. No matter in decimal or binary field, many tricks and

techniques have been used to speed up the division. However, most of their basic division algorithms are SRT based high-radix division. This method is also considered to be applied to the radix-100 divider. Although it seems that SRT is not suitable for the radix-100 division, it is still worth discussing the basic ideas of the high-radix SRT binary division algorithm.

Assuming radix β ($\beta = 2^m$) is utilized to perform the high-radix division, then the number of the total steps generating the required number of quotients n is reduced to $\frac{n}{m}$. The equation of this high-radix division would be in equation

$$R_i = \beta \times R_{i-1} - q_i \times D \tag{1.6}$$

Where the q_i is no longer $[-1, 1]$. Instead, q_i can be any value ranging from $-\alpha$ to α, where α should meet the equation [19].

$$\lceil \frac{1}{2}(\beta - 1) \rceil \leq \alpha \leq (\beta - 1) \tag{1.7}$$

As discussed before, the convergence requirement should be $|R_i| \leq |D|$. This requirement should be changed. Assuming k is the factor which represents the influence of the α in high-radix division algorithm. So the following equation is obtained.

$$|R_i| \leq k \times |D| \tag{1.8}$$

After solving that inequality with equation (1.6), an important equation (1.9) can be obtained.

$$k \leq \frac{\alpha}{(\beta - 1)} \tag{1.9}$$

Larger α (larger redundancy) always means easier SRT operation since the restrictions on the partial remainder are much wider and the overlap regions are larger than those in the dividers with smaller redundant number system. On the other hand, larger α means more quotient candidates, more $q \times Ds$ and more complex quotient selection algorithm. Consequently, it is hard to say what kind of redundancy is the best since each algorithm has a different situation.

The overlap region is an essential concept in the high-radix SRT division algorithm. For example, if β is 8 and α is 5, then k should be $\frac{5}{7}$ and according to equation (1.6) and (1.8), the following equation is obtained.

$$-\frac{5}{7} + q \leq \frac{8 \times R_{i-1}}{D} \leq \frac{5}{7} + q$$

When $q = 1$, the inequality is transformed to $\frac{2}{7} \leq \frac{8 \times R_{i-1}}{D} \leq \frac{12}{7}$. When $q = 2$, $\frac{9}{7} \leq \frac{8 \times R_{i-1}}{D} \leq \frac{19}{7}$ can be obtained. Obviously, if $\frac{8 \times R_{i-1}}{D}$ falls into the region $[\frac{9}{7}, \frac{12}{7}]$, the quotient can be either 1 or 2, which both satisfy the convergence requirement.

Normally, we need to choose a boundary within a overlap region. Any partial remainder located in the overlap region should be compared with the boundary. Notice that the comparison is normally performed by subtraction followed by a sign detection module so one may need to choose the best boundary (with fewer digits) which is easier for implementation. Since larger redundancy results in larger overlap regions, it is easier to find better boundaries when the redundancy is large enough.

P-D plot (Figure 1.1) is the basic method of choosing the boundaries. First of all, equation (1.6) should be transformed to

$$\beta \times R_{i-1} = q_i \times D + R_i \tag{1.10}$$

Replacing R_i in equation (1.10) with R_i in equation (1.8), the range of $\beta \times R_{i-1}$ can be obtained as shown in equation (1.11).

$$(-k + q) \times D \leq \beta \times R_{i-1} \leq (k + q) \times D \tag{1.11}$$

$\beta \times R_{i-1}$ is the shifted partial remainder. Working with several divisor Ds, the P-D plot can be obtained and illustrated in Figure 1.1.

Nevertheless, it is impossible to define a boundary for each divisor in the hardware implementation. An alternate way is to store some boundaries in a look-up table for each section of divisor [23]. Here, the details of choosing the sections and calculating the best boundaries will not be introduced, since they are not important in our design. Refer to [19] for more details.

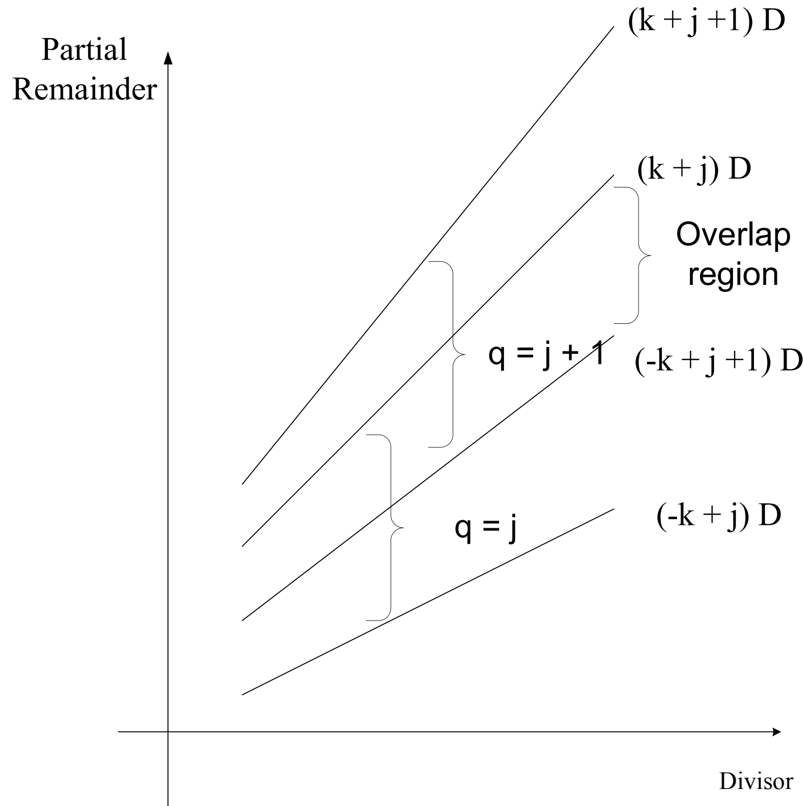

Figure 1.1 P-D plot

Overall speaking, three major parts should be considered if one implements a divider by using the high-radix SRT division. They are redundancy, divisor sections, and the boundary of each section. The algorithm discussed in this section is just the basic algorithm of the high-radix SRT division. In the real implementation, no one will use the whole partial remainder to do the comparison since the number of bits/digits in the partial remainder will result in a large amount of delay. Consequently, the estimated partial remainder will be used and the error will be eliminated during the selection of boundaries. This technique will be explained in the next chapter.

1.6 Contribution

The major contributions of this thesis are:

1. It proposes a novel high-radix decimal division algorithm. The radix-100 divider can produce two quotient digits in each iteration, which can reduce the number of

iteration cycles. Consequently, the overall latency can be reduced.

2. It avoids the popular SRT division algorithm which would result in a huge area and a long latency. Instead, the radix-100 divider utilizes selection by truncation algorithm. Compared to SRT algorithm, selection by truncation method has a fast quotient selection step which is one of the major parts implemented in the critical path.

3. It combines the partial reminder in the form of "carry-sum" and the selection by truncation method to achieve a fast quotient selection and an easy partial remainder calculation.

4. It utilizes the compensation method to reduce the area and latency consumed to generate and select multiples of divisor.

2. Previous Work

In this chapter, some recent typical works from related areas are included. Most of them concern decimal dividers. As already mentioned, the proposed radix-100 division algorithm is derived from the decimal division algorithms. Meanwhile, all the dividers focus on the calculation of the coefficient part as shown in equation (1.1). Before the introduction of those algorithms, the basic equation of decimal division is provided in equation (2.1), which is derived from equation (1.6). R_i is the partial reminder generated from the *ith* iteration while q_i is the quotient digits selected in the *ith* iteration. D stands for the divisor.

$$R_i = 10 \times R_{i-1} - q_i \times D \tag{2.1}$$

2.1 Typical SRT-based Decimal Divider

Tomas Lang and Alberto Nannarelli proposed a decimal division algorithm based on SRT in the year 2007 [23]. To deal with the two common complicated issues in decimal division (the selection function and the generation of the divisor multiples) the authors came out with a solution, which is to split the quotient digit into two parts as shown in equation (2.2)

$$q_i = 5 \times q_{Hi} + q_{Li} \tag{2.2}$$

Where $q_{Hi} \in [-1, 0, 1]$ and $q_{Li} \in [-\alpha, ..., \alpha]$. By replacing the R_{i-1} in equation (2.1) with equation (2.2), equation (2.3) is obtained.

$$R_i = (10 \times R_{i-1} - 5 \times q_{Hi+1} \times D) - q_{Li+1} \times D = V_i - q_{Li+1} \times D \tag{2.3}$$

Note that $\alpha = 2$ is chosen in equation (2.3). Although the details concerning

why $\alpha = 2$ is chosen are not available in [23], it is found that all of the following possibilities are reasons: first of all, only if $\alpha \geq 2$ will every number within $[-5, 5]$ be represented by q_H and q_L. Second, since the number of multiples of divisor is one of many important aspects which can influence the performance significantly, $\alpha = 2$ is the best choice. Another reason is that the doubler is quite easy to implement. All of these features contribute to the selection of $\alpha = 2$.

Since the q_i ranges from -7 to 7, $k = \frac{7}{9}$ is obtained according to equation (1.9). The next major step is to calculate the boundaries. Here, instead of using the whole partial remainder to select the quotient, only a few of the most significant digits (MSD) from the partial remainder are used. The use of truncated partial remainder will obviously introduce errors, as shown in equation (2.4)

$$L_k(D_{j+1}) \leq m_k j \leq U_{k-1}(D_j) - 1.12 \times 10^{-t} \tag{2.4}$$

Obviously, 1.12×10^{-t} is the error introduced by truncation and the t is the number of digits in the partial remainder used to select the quotient. Such a technique is used by almost all the SRT-based dividers [32] [25] [24] [33] [34] [35], since no one can afford the comparisons between compact partial remainder and boundaries. Meanwhile, L_k and U_k are the boundaries of quotient k on the P-D plot, and $[D_i, D_{i+1}]$ is a section of the divisor.

According to the convergence requirement shown in equation (1.8), two boundaries are set for R_i and V_i. The equations for the L_ks and U_ks corresponding to R_i and V_i respectively, are also obtained and shown in (2.5):

$$\begin{aligned} U_{kV} &= (5k + 25/9) \times D \\ L_{kV} &= (5k - 25/9) \times D \\ U_{kR} &= (k + 7/9) \times D \\ L_{kR} &= (k - 7/9) \times D \end{aligned} \tag{2.5}$$

The selection of "t" will influence the performance. The smaller the "t" is, the faster the quotient selection (SRT partial remainder comparison) will be. However,

there is a restriction on "t" as shown in equation (2.6)

$$U_{k-1}(D_i) - L_k(D_{i+1}) \geq 1.12 \times 10^{-t} \tag{2.6}$$

After equation (2.6) is solved, t = 2 is selected which means three MSDs from the partial remainder are then used to select the quotient. The boundaries employed in the divider are selected by the authors. The largest subinterval of D following equation (2.4) are obtained. Also, notice that an important symmetrical property of the P-D plot is shown in equation (2.7), so not all boundaries need to be selected individually.

$$m_{-k+1} = -m_k \tag{2.7}$$

In terms of the architecture, the authors utilized two techniques to reduce the cycle time. The first one is to employ carry-save subtraction and sign detection for the comparisons. The carry-save subtraction (addition) is also widely used in the proposed radix-100 divider. On the other hand, it would be unacceptable if the q_L is selected after the calculation of V_i, since this will introduce more delay. Therefore, the selection of q_H and q_L are overlapped; this technique is used in a radix-16 divider as well. Since the partial remainder is in the form of sum and carry, the selection of q_L is now shown in equation (2.8)

$$(10R_S)_{trun} + (10R_C)_{trun} - u \times (5D)_{trun} - m_{Lk} \tag{2.8}$$

in which the $-u \times (5D)_{trun} - m_{Lk}$ can be pre-computed with $u = [-1, 1]$ and $k = [-1, 2]$ (eight values in total). By performing such overlapping, the q_H and q_L can be selected almost at the same time with the sacrifice of area.

Another smart retiming is the use of radix-2 to implement the quotient selection part. Radix-2 is faster than decimal operation. By adding the compensation digits generated from the decimal subtraction part, the binary part can perform separately and generate correct results.

As a typical SRT-based decimal division algorithm, [23] was analyzed carefully. The following aspects contribute to the proposal of the radix-100 divider:

1. Is the SRT algorithm suitable for the radix-100 divider? Similar to the digit-set used in Tomas Lang's algorithm, the radix-100 divider's quotient can be decomposed into two parts, q_H and q_L. However, in Tomas's decimal divider, the authors overlapped the selection of quotients into one level of CSA followed by a sign detection module, which reduces the latency but results in 14 CSAs and sign-detection blocks. In terms of the radix-100 divider, if a similar overlapping method is used, there should be at least 100 CSAs and sign-detection blocks, which will result in huge MUXes and more delay, not to mention the area caused by those CSAs and other blocks. By contrast, if two levels of quotient selection are used instead of overlapping, the radix-100 divider is nothing but a cascaded decimal divider which lacks novelty.

2. Is carry-save subtraction (addition) helpful? In [23], three operands are added together. The CSAs (or DCSAs) work much better than any other addition logics. In the proposed radix-100 divider, more operands are involved in the operation so that decimal CSAs would become essential. This is one of the reasons for decimal CSAs being widely used in the proposed radix-100 divider.

3. Does the binary quotient selection technique also work on the radix-100 divider? The binary solution works fine with the SRT division under the condition of proper retiming. But the reasons that SRT is not suitable for the radix-100 divider are just discussed, leaving the use of the binary system unconsidered.

4. Compensation. The compensation digits passed from the decimal side to the binary side help to maintain a correct binary quotient selection. In the radix-100 divider, a carry generated from the decimal prefix-tree works in ways similar to the compensation digits.

5. Although the quotient's digit-set is $[-7, 7]$, the partial remainder's digit-set is $[0, 9]$. This partial-remainder's digit-set is straightforward so it is also used in the radix-100 divider. However, some recent work done on the radix-100 divider shows that employing signed-digit number system may result in a better area and performance. This consideration will be included in the future work.

2.2 Non-restoring based Decimal Divider

Eric M.Schwarz and Steven R.Carlough published the "Power6 Decimal Divide" in 2007 [17]. Non-restoring division algorithm is the basic decimal division algorithm present in Power6, so it is worthwhile to read this paper carefully. This algorithm is based on high-frequency BCD hardware.

Instead of using SRT, Power6 utilizes the traditional non-restoring method which follows steps: quotient selection, multiples of divisor, creation of the partial remainder, and the final quotients accumulation. Normally, the critical path goes through the first three steps, so some techniques should be used to reduce the cycle time.

Here, the pre-scaling and selection by truncation method is utilized. The basic idea of pre-scaling is to convert the divisor close to 1, so that the quotient selection will be easier, as shown in equation (2.9)

$$\begin{aligned} q_i &= (R_{i-1})_{trun} \\ R_i &= 10 \times R_{i-1} - q_i \times D \end{aligned} \tag{2.9}$$

In the above equations, $(R_{i-1})_{trun}$ is the first digit of the partial remainder. This type of quotient selection is called selection by truncation. Although there will be errors in certain quotient digits, the non-restoring method can correct the errorous quotient digits by applying their following quotient digits. But this kind of correction can handle only one-unit's error.

After some calculations, it is proven that the divisor should be pre-scaled to equation (2.10) so that the the convergence requirement $|R_i| \leq |D|$ can be met.

$$1 \leq D' \leq 1/9 \tag{2.10}$$

The pre-scaling is done through a two-cycle BCD adder adding multiples generated from a BCD doubler and quintupler. After completing analysis of the quotient-selection part, it is found that another issue is the generation of multiples of the scaled divisor. The $1D'$ is the scaled divisor while the $2D'$ and $5D'$ can be obtained

through the BCD doubler and quintupler. In the implementation of Power6, the $3D'$ and $4D'$ are calculated in advance and saved in registers before the start of division. However, there is no time for the calculation of multiples ranging from $6D'$ to $9D'$. Power6 therefore introduces a compensation method by generating an alternate partial remainder (partial remainder B) in the previous iteration. Partial remainder B is employed to compensate the missing of $6D'$ to $9D'$. The calculation of the partial remainder B is shown in equation (2.11).

$$R_{iB} = 10 \times R_{i-1} - (q_i + / - 1) \times D' \tag{2.11}$$

For instance, if the current selected quotient is 6, the partial remainder B will be used as the current partial remainder and will replace the R_{i-1} in (2.1). This will result in equation (2.12)

$$R_i = 10 \times R_{i-1B} - 6 \times D' \tag{2.12}$$

$$= 10 \times (R_{i-1} + 1 \times D') - 6 \times D' \tag{2.13}$$

$$= 10 \times R_{i-1} + 4 \times D' \tag{2.14}$$

Therefore, the $4D'$ can be selected to perform the calculation in that case. Also, to determine whether the partial remainder B should perform plus or minus one, the second digit of the current partial remainder should be checked. Since the divisor is pre-scaled to less than 1.1, when the current selected quotient falls in the range of 6 to 9, it is possible that the next partial remainder could be negative and the abstract quotient would then be larger than or equal to 5. This circumstance needs a partial remainder B based on minus one.

Since Power6 is a commercial product, the paper does not describe in detail the architecture of the proposed decimal algorithm. For instance, the details of the adders performing the calculation of the partial remainder are missing. To cater to the overall clock frequency, the divider is decomposed into several pipeline stages.

The Power6 needs three cycles (non-redundant partial remainder) to perform each iteration, but with the help of pipeline, the latency consumed on performing a sequence of data will be shorter. However, since most of the decimal division dividers

do not care about the real ALU implementation, most of the proposed decimal algorithms do not take the pipeline stages into consideration as the proposed radix-100 divider does.

Although [17] only introduces the overall algorithm and does not touch upon many details, it provides several good ideas which can be considered in the design of radix-100 divider:

1. The non-restoring method: As discussed in the first section, the decimal SRT method is not a good choice for the radix-100 divider. The major reason is the quotient selection. But with the non-restoring method and the selection by truncation, the quotient selection is much simpler than that of the SRT division. Besides, there is no big difference between the latency of the quotient selection in the radix-100 divider and that in a decimal divider.

2. The adders used in the Power6 are uncertain. However, as concluded from Tomas Lang's algorithm, decimal carry-save addition is a good choice for multiple addends. Consequently, if there is a way to combine the selection by truncation method in Power6 and decimal CSA addition, the latency would be acceptable.

3. The generation of multiples of divisor. As described in [17], the BCD doubler and quintupler are easy to be implemented and fast in terms of latency. The same modules may be useful in the radix-100 divider. Besides, the quotients selected in the radix-100 divider can be split into two parts, which are q_H and q_L. Both of them can be treated as decimal numbers, so multiples of the divisor can be shared by the two sub-quotients.

4. BCD operation. BCD represented decimal digits are used as the basic format in this decimal divider. It is quite likely that BCD representation is suitable for the radix-100 divider as well.

As a mature design already implemented in the Power6 microprocessor, the Power6 decimal divider should be considered as a standard reference for future decimal

or higher-radix dividers. Consequently, the proposed radix-100 divider utilizes the same basic algorithm, which is the non-restoring division algorithm with selection by truncation.

2.3 Decimal Divider with Different Encodings

Another typical decimal divider was proposed in the year of 2007 by Alvaro Vazquez Alvarez [24]. Again, the decimal SRT method is utilized with some new techniques. The basic equation is (2.1) and the basic format of the partial remainder is non-redundant. However, the author utilized some different encodings in different parts of the calculation as discussed later below.

The BCD encoding, although widely used, is considered as an inefficient encoding since 8421 encoding can represent 16 different digits while the BCD utilizes only 10 of them. In the proposed radix-100 divider, BCD is the basic encoding format since BCD is mature and the modules calculating BCD arithmetics are widely used. Besides, after analyzing the encodings proposed by Vazquez, one can discover that the benefits they can introduce to the radix-100 divider are limited. Several encodings and their efficiencies are concluded in Table 2.1.

Table 2.1 Different encodings and their efficiency

Encoding	8421	3321	4221	5211	4321	5221
Efficiency	10/16	1	1	1	10/11	10/11
Maximum representable numbers	16	10	10	10	11	11

In this table, the efficiency (Used numbers/Maximum representable numbers) of each encoding is concluded. It is obviously seen from the table that the encoding 3321, 4221, 5211 have the best efficiency. As proven in Vazquez's PhD thesis, the encoding 4221 and 5211 can lead to fast carry-save adders. This quality is especially shown by the 5211 carry-save addition, which takes three 5211 addends and produces one 5211 sum and one 4221 carry. But a decoder should be added to transform the carry to 5211, which needs 8 levels of gates. On comparison to the decimal BCD

carry-save adder (DCSA) which also takes 8 levels of gates, although the DCSA's real implementation is slightly slower than the 5211 DCSA, the benefit of using 5211 coding style is not remarkable.

The digit-set for this divider is $[-5, 5]$. Consequently, the partial remainder should be smaller than or equal to $5/9 \times D$. As in Tomas Lang's decimal division algorithm, the estimated partial remainder and divisor are used, and the formula (2.15) is for $k \geq 0$ and (2.16) is for $l \leq 0$:

$$L_k(D_{j+1}) \leq m_k j \leq U_{k-1}(D_j) - \delta\epsilon_w + h(m_k) \tag{2.15}$$

$$L_k(D_j) \leq m_k j \leq U_{k-1}(D_{j+1}) - \delta\epsilon_w + h(m_k) \tag{2.16}$$

Here, instead of using a number of digits to represent the length of an estimated partial remainder and divisor, Vazquez decided to use the number of bits since in this case, the number of bits might be smaller than the number of digits, which will result in a lower delay. In addition to the previous two restrictions, the estimated partial remainder should follow equation (2.17)

$$-\delta\epsilon_w - 10 \times \frac{5}{9} \times D \leq (10 \times R_i)_{est} \leq 10 \times \frac{5}{9} \times D \tag{2.17}$$

The author analyzed three decimal encodings, which are BCD, 4221 and 5211. It is proven that 5211 uses one less fractional bit for the SRT comparison. Besides, since the 5211 carry-save adder (with 4221 carry output) has the same delay as that of the binary CSA, it is employed on the estimation of the partial remainder and the divisor as well. On the other hand, since the decimal 5421 based carry-propagate adder has the similar latency as the BCD carry-propagate adder and the conversion between the 5421 and 5211 is simple enough, the calculation of the partial remainder is done in a decimal 5421 adder.

Instead of using a look-up table to store all the boundaries, here in [24], the multiples of the estimated divisor will be calculated first; some rounding and truncation techniques will be used on those multiples to generate the real boundaries. Since that

topic is not important to the radix-100 divider, the details are not described here. Refer to Alvaro Vazquez Alvarez's thesis for detail information.

In terms of the retiming and architecture, [24] works in patterns similar to Tomas Lang's architecture. The quotient selection (selected quotient will be used in the next iteration) and the calculation of the current partial remainder are performed in parallel. The critical path goes through the quotient selection part.

The major novel aspect of [24] is the different encodings. The major features of different encodings used in [24] are concluded below.

1. 5211: 9's complement is the reversion of the original format. Since some multiples of divisor are involved in the radix-100 divider, this feature is useful. In addition, as introduced above, the 5211 carry-save adder (with 4221 carry output) is fast.

2. 4221: Right shift "a" in 4221 format will result in "a/2" in 5211 format. What's more, there is a 4221 carry-save adder already proposed by Vazquez.

3. 5421: Left shift "a" in 5421 format will result in "2a" in 8421 format. Left shift "a" in BCD format will result the "5a" in 5421 format.

Features 2 and 3 can be used to generate multiples of divisor. Besides, it seems that 5211 has some benefits for the iteration. First of all, the quotient selection step is considered. If the traditional BCD carry-save format is used to represent the partial remainder, the carry digit has only one bit, which will simplify the implementation of selection by truncation. If the partial remainder is in the form of 5211 and 4221, a 5211 and 4221 carry propagate adder should be designed. Although a 4-bit addition should be enough, it still takes more time. Consequently, the quotient selection step will be longer than that using the BCD format if the 5211 and 4221 partial remainder is used, but it is still acceptable if the addition part can reduce the time significantly.

In terms of the addition, a partial remainder in the form of sum-carry and at least two multiples of divisor are the basic addends which cannot be avoided. Consequently,

the addition cannot be done in one level of carry-save adder, and a decoder should be used since the second level of addition needs inputs with the same decimal encoding. Other than that, further analysis shows that decoders should be used in multiple places, which will add more latency. Therefore, BCD is decided to be used.

2.4 Radix-16 Dividers

Two typical high-radix (radix-16) dividers are briefly introduced in this section. Both of them are derived from the radix-4 SRT division. Before going further, note that there are two equations, (2.18) and (2.19), which are utilized in both designs.

$$R_i = 16 \times r_{i-1} - q_i \times D \tag{2.18}$$

$$q_i = 4 \times q_{Hi} + q_{Li} \tag{2.19}$$

The first radix-16 divider is proposed in [29]. Both of the q_{Hi} and the q_{Li} fall into $[-2, 2]$. Similar to Tomas Lang's quotient selection method, overlapping is chosen again in this radix-16 divider. As shown in equation (2.20) and Figure 2.1

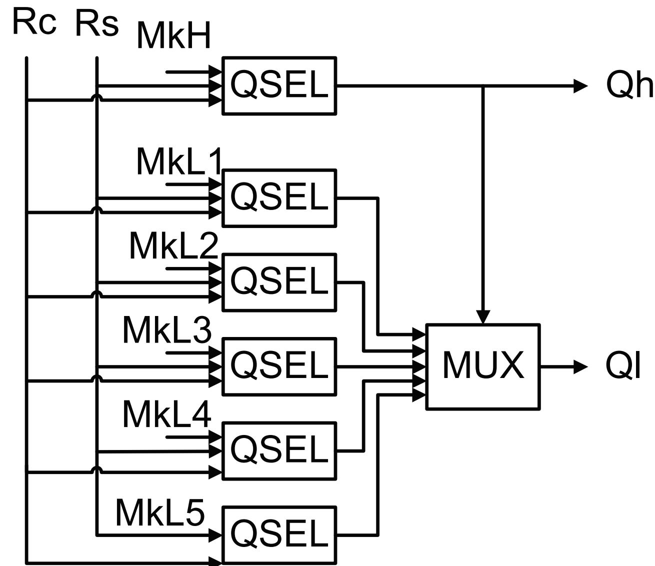

Figure 2.1 Overlapping quotient selection

$$q_{Li} = \begin{cases} SEL_L((R_i)_{est} + (2D)_{est}, D_{est}) & \text{if } q_{Hi} = -2 \\ SEL_L((R_i)_{est} + D_{est}, D_{est}) & \text{if } q_{Hi} = -1 \\ SEL_L((R_i)_{est}, D_{est}) & \text{if } q_{Hi} = 0 \\ SEL_L((R_i)_{est} - D_{est}, D_{est}) & \text{if } q_{Hi} = +1 \\ SEL_L((R_i)_{est} - (2D)_{est}, D_{est}) & \text{if } q_{Hi} = +2 \end{cases} \quad (2.20)$$

However, as discussed in Chapter 2.1, the overlapping method is not applicable for the radix-100 division, since the quotients' digit-set is much wider than the radix-16 division. The consumed area and timing would be much more than those of the radix-16 division.

Another high-radix divider appears in the Intel Core2 Penryn Processor family [30]. Also, since the Core2 is an commercial product, the paper [30] does not introduce many details regarding its radix-16 divider. However, it is obvious that the design is based on a cascaded SRT division method. With the new "digit-redundant represented partial remainder" (similar to the carry-save format) and the "implicit bias bits concept" (similar to the use of estimated partial-remainder and divisor), the proposed radix-16 divider uses only a few bits for the selection of quotients; thus it is proven to be an efficient and fast way of doing the division compared to the original radix-4 divider [30], which uses a carry-propagate adder in each iteration. As shown in Figure 2.2, there are two levels of radix-4 SRT divider in the radix-16 data flow, so this method can be considered as a cascaded low-radix division algorithm.

Nevertheless, notice that even though the new radix-16 is much better than the original radix-4 divider, according to the comparison results obtained by [29], the performance improvement of the cascaded radix-16 divider is not outstanding compared to a SRT based radix-4 divider. The brief architecture of the cascaded radix-16 divider is illustrated in Figure 2.2.

Another paper [26] published in 2011 combines the features in [23] and [24] to reach better performance. Since there are no new techniques utilized in that paper, details are not provided. Its performance will be given in Chapter 5.

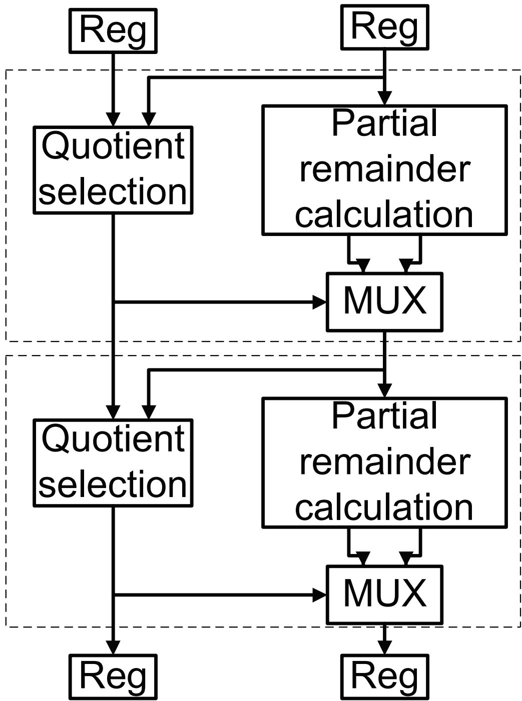

Figure 2.2 Cascaded radix-16 divider

Generally speaking, after the analysis of the typical papers included in this chapter and some other papers, the radix-100 divider utilizes the non-restoring with selection by truncation method similar to the Power6 decimal divider. In addition, the basic number system would be BCD.

3. Algorithm

To begin with, the details of the radix-100 divider need be introduced. The radix-100 algorithm is described in this chapter, followed by the description of its architecture. Implementation and comparison will be discussed in Chapter 5.

3.1 Decimal Floating-point Division

As described in Chapter 1, the IEEE 754-2008 defines three basic formats for decimal floating-point numbers, which are decimal 32, decimal 64 and decimal 128. The proposed radix-100 divider is implemented on decimal 64 standard. The reason for this choice is that most of decimal dividers are based on decimal 64, and decimal 128 dividers can be implemented by modifying the decimal 64 easily.

The architecture of the overall decimal floating-point divider is shown in Figure 3.1. As introduced in Chapter 1, the floating-point number has three parts, which are sign, coefficient and exponent. Assuming dividend FX and divisor FD are enrolled in the division, and they are in form (3.1)

$$\begin{aligned} FX &= (-1)^{S_X} \times X \times 10^{E_X - bias} \\ FD &= (-1)^{S_D} \times D \times 10^{E_D - bias} \end{aligned} \tag{3.1}$$

The X and D are the coefficients representing the fractional digits, which means there is a virtual point before the left-most digit of the coefficient. But notice that the IEEE 754-2008 does not require the removal of the leading zeros of the decimal coefficient [22], which means that there might be some zeros locating in the left side of the coefficient. To make sure that the result has the maximum number of digits

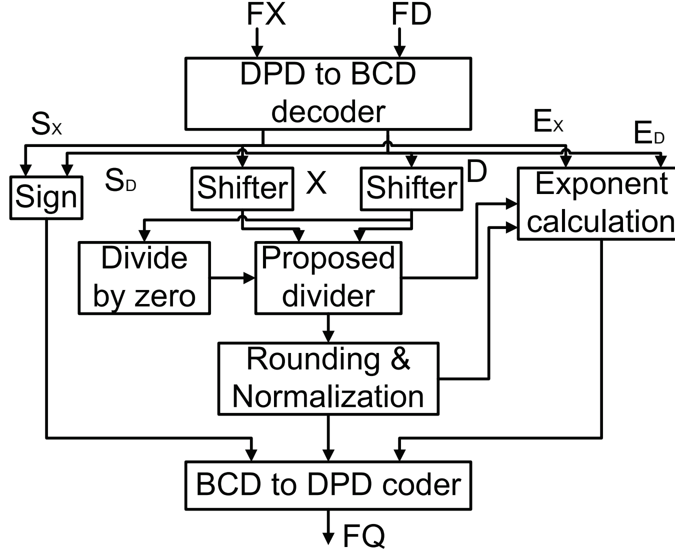

Figure 3.1 Architecture of IEEE 754-2008 divider

and to ensure the convergence, the leading zeros should be removed through a shifter. Obviously this shifter will influence the exponent. Assuming there are Lz_X and Lz_D leading zeros in FX and FD respectively, the exponent obtained after shifting would be (3.2) after shifting.

$$E_Q = E_X - E_D + Lz_D - Lz_X \tag{3.2}$$

Since the pre-scaling parameter should also be applied to the dividend, the exponent will be influenced again.

The calculation of the coefficient will be discussed in Section (3.3). It is also possible that the divisor equals to zero, a condition which the proposed fixed-point divider is not designed to handle. This possibility is one of the exceptions mentioned in Chapter 1. If that happens, the module "divide by zero" will assert a signal to point out an exception. This module is not part of the fixed-point radix-100 divider block, that can produce two digits in each cycle. In the end, to perform the final rounding and normalization (optional), 18 digits will be produced. A module named "rounding and normalization" is in charge of this operation.

In terms of the sign, it is assumed that the coefficients provided to the divider are

positive, so the coefficient of the result is also positive. The sign will be generated through an XOR gate, as shown in equation (3.3)

$$S_Q = S_X \oplus S_D \tag{3.3}$$

The final result will be (3.4)

$$FQ = (-1)^{S_Q} \times Q \times 10^{E_Q - bias} \tag{3.4}$$

Required by the IEEE 754-2008 standard, the decimal numbers should be saved in the form of a DPD. So there is a DPD to BCD decoder at the very beginning of the data flow and a BCD to DPD coder in the end. This is the same as the DFU in [4].

Overall, there are separate modules dealing with coefficient, exponent and sign. But only the modules related to the calculation of the coefficient (proposed divider, rounding and normalization module) are introduced since the other modules will not influence the overall performance. They are not included in other implementations mentioned in the previous chapter.

3.2 Decimal Representations

Before further explanation, the representations of the decimal operands should be introduced well in advance. Normally, there are three important representations as follows:

1. The dividend, divisor, and the quotient: these three operands are defined by IEEE 754-2008, so there are not very many choices. One can use the DPD directly but because it is a highly compact encoding, DPD is not a good choice. So, the BCD is normally used to represent those three operands.

2. The partial remainder R_i: As shown in Chapter 2, there are several different forms of partial remainder. It can be represented in a non-redundant format which normally needs a carry-propagate adder. After a proper retiming, the carry-propagate

adder can be used as seen in [4] which uses 5421 encoding. Moreover, some references use carry-save format [36] [26] and various other techiniques [22]. Besides, in terms of each single digit of the partial remainder, there can be different ranges as long as equation (1.7) is met. In this thesis, however, the digits of the partial remainder are chosen from $[0, 9]$ while the negative numbers are represented by 10's complement, just as Tomas Lang did in his decimal divider. This choice suits our selection by truncation algorithm quite well since no recoding is needed.

3. The quotient digit q_i: Similar to the partial remainder, the quotient digit can choose any range following equation (1.7). The selection of quotient's range will influence the performance significantly. The larger redundancy means more multiples of divisor. In terms of SRT, it also means more boundaries. But here, the maximum redundancy $[-9, 9]$ is selected. The reason for choosing this redundancy is that it is the most nature range for the selection by truncation method. By using this range, the quotients can be determined easily without performing any decoding. However, a wide range definitely introduces some troubles relating to the generation of multiples. Several tricks and retiming techniques should be used to minimize the latency caused by the generation of multiples.

There are several carry-save adders, which can generate partial remainder in the form of "carry-save". For instance, as described in Chapter 2, some carry-save adders are designed for different encodings other than BCD. Here only the BCD carry-save adders are considered. There are two adders to be considered [4]:

1. Digit carry-save: This kind of carry-save adder can add two full range BCD addends and one carry (each digit equals to 0 or 1) and the results are one full range sum and one carry. The basic arithmetic behind such feature is that the addition of the addends are $|9 + 9 + 1| = |19|$, which can be represented as one carry and one sum whose abstract value is smaller than or equal to 9.

2. Full carry-save: This carry-save adder can add three full range BCD addends. The results is composed of one full range sum and a carry within $[-2, 2]$. Similarly,

the basic idea is that $|9+9+9| = |27|$ can be represented in a larger carry and a sum.

In this thesis, the first DCSA is chosen since the timing latency of the first one is smaller then that of the second one. Although using the second one can reduce some area of the current radix-100 design, sacrificing the area would be preferred if a better latency can be obtained.

3.3 Proposed Algorithm

Assuming that R_i is the partial remainder obtained after ith iteration and q_i is the quotient used in the ith iteration. D is the divisor. The basic equation for the calculation of the radix-100 division is (3.5)

$$R_i = 100 \times R_{i-1} - q_i \times D \tag{3.5}$$

Since the quotient's digit-set is $[-99, 99]$ as shown in the previous section, here the fraction k (in equation (1.9)) can be calculated through equation (3.6) since the α is replaced by 99 and the radix β is 100.

$$k = \frac{99}{100-1} = 1 \tag{3.6}$$

Based on the convergence requirement ($|R_i| \leq k \times |D|$), the radix-100 division should comply with

$$|R_i| \leq |D| \tag{3.7}$$

Similar to [17], the selection by truncation method used on the radix-100 divider needs a pre-scaled divisor which is close to 1. But how close should it be? "Close" can be understood as close and larger than 1, close and smaller than 1, or close and around 1. In this thesis, only the scaled divisors larger than 1 are considered. The reason for this choice is that the partial remainder digits are all positive values ranging from 0 to 9. So in terms of positive partial remainder, a subtraction is needed. Assuming that the scaled divisor is smaller than 1, then the next partial remainder would be

definitely positive and probably larger than one. If it is larger than one, errors will be accumulated, so the division will never end. Consequently, the scaled divisor used in the proposed radix-100 divider should be "close and larger than 1", as represented in equation (3.8)

$$D' \in [1, 1+\epsilon) \tag{3.8}$$

In fact, although the divisor D is used in equation (3.7), it is the scaled divisor D' that is the real divisor involved in the calculation of the proposed radix-100 divider. So, equation (3.7) can be modified as

$$|R_i| \leq |D'| \tag{3.9}$$

This modification is also applied to equation (3.5)

$$R_i = 100 \times R_{i-1} - q_i \times D' \tag{3.10}$$

Replacing the $|R_i|$ in equation (3.9) with the R_i in equation (3.10) and the $|D'|$ with (3.8), an inequality regarding ϵ is obtained

$$|100 \times R_{i-1} - q_i \times D'| \leq min([1, 1+\epsilon)) = 1 \tag{3.11}$$

The determination of ϵ is essential for the whole divider since this value will determine whether the selection by truncation method is possible for the radix-100 (if $1+\epsilon$ is smaller than 1, then it means the divisor cannot be pre-scaled to the range (3.8)). ϵ also determines the complexity of the pre-scaling step. At this point, ϵ will also determine the number of digits in the pre-scaling parameter, which will then determine the whole architecture, as will be introduced later. By replacing the D' in (3.11) with $1+\epsilon$, the range of ϵ can be obtained.

$$\begin{aligned} -1 \leq 100 \times R_{i-1} - q_i \times (1+\epsilon) \leq 1 \\ => -1 \leq 0.X - q_i \times \epsilon \leq 1 \\ => \epsilon \leq \frac{1}{99} \end{aligned} \tag{3.12}$$

Therefore, the range of the D' is $[1, 1.01]$. This is a relatively tight range but it is found that three digits' parameter is enough to pre-scale the divisor into that range, which is still acceptable. The details of the selection of the pre-scaling parameters will be discussed later in this section.

There should be some modifications on the dividend, too. First of all, the dividend should be multiplied by the same pre-scaling parameter used on the divisor, and the result is R_0'. Since the dividend should be treated as the first partial remainder which should follow the equation (3.10), the pre-scaled dividend is shifted as shown in (3.13) where "n" represents the number of integers in the pre-scaled dividend.

$$0.1 \leq \frac{R_0}{n} \leq 1 \tag{3.13}$$

3.3.1 Pre-scaling Parameters

There are several different methods of pre-scaling. As concluded in [37] and [21], to calculate the pre-scaling parameter (similar to the calculation of the reciprocal), one of three methods can be used:

1. Using a look-up table which can provide the pre-scaling parameters directly. This method suits the pre-scaling with lower precision requirements, and the parameters should have a limited number of digits.

2. Using a look-up table to provide a pair of coefficients before performing linear approximation. This is popular in designs based on functional iteration (Newton-Raphson) [27] and some very high-radix division implementations [37], since they can benefit from an accurate initial reciprocal approximation.

3. Using a Newton-Raphson iteration with the initial approximation provided by either one of the two previous methods. This will generate a full-range accurate reciprocal value. This is, in fact, the process to get the reciprocal of a certain number, which is done through a functional iteration based divider.

The first method is obviously the most straightforward one as long as the number of digits in the pre-scaling parameters is acceptable. After the further analysis shown

below, each of the parameters consists only of three decimal digits, which is acceptable. Consequently, similar to the Power6 decimal divider, a look-up table indexed by the divisor directly is utilized to provide the pre-scaling parameters.

"Pa" is used to represent the pre-scaling parameter. The pre-scaling requirement shown in (3.12) should be met by all possible divisors, including two ultra situations:

$$\begin{aligned} 1 \leq (0.D_1D_2D_3...D_n0....0) \times Pa \leq 1 + \frac{1}{99} \\ 1 \leq (0.D_1D_2D_3...D_n9....9) \times Pa \leq 1 + \frac{1}{99} \end{aligned} \tag{3.14}$$

So, the range of Pa for each divisor would be

$$\frac{1}{0.D_1D_2D_3...D_n0....0} \leq Pa \leq \frac{1 + 1/99}{0.D_1D_2D_3...D_n9....9} \tag{3.15}$$

There are two vital things to be determined from (3.15), which are the number of digits in the divisor (the value of "n") serving as the index to the look-up table and the number of digits in the Pa. A MATLAB program is written to list those two important values. The results are available in Table 3.1. Since the range $[1, 1+1/99)$ required by the radix-100 divider is larger than that of the Power6 divisor which uses two digits' pre-scaling parameters, the number of parameter digits would be at least 2. Therefore, the table starts from $n = 2$ to $n = 5$. The third and fourth row of the table are the range of 0.15111 and its next nearest number (based on precision) respectively. They are used as examples of the ranges (only five MSDs) of parameters.

Table 3.1 Pre-scaling parameter comparison

n = 2	n = 3	n = 4	n=5
Invalid	3	3	3
66.667,63.13	66.225,66.454	66.181,66.806	66.177, 66.841
62.5,59.418	65.789,66.02	66.138, 66.761	66.173, 66.837

The rules of the selection of parameters are: the upper boundary in (3.15) should be larger than the lower one, and the number of exactly the same digits in the upper

and lower boundaries plus one is the number of digits in the parameter. For instance, if the boundaries for 0.151 are [66.225,66.454], the parameter can be set to 66.3. Of course, these rules should apply to all the divisors. Concluded from the table, $n = 3$ is the best choice for the radix-100 divider. This means that three MSDs of the divisor are used to index the look-up table, where the three digits' parameters are saved.

There are some exceptional cases when $n = 3$. In these cases, the divisor should be pre-scaled by 4-digit parameters. They are available in Table 3.2.

Table 3.2 Parameter exceptions

Divisor's 3 MSDs	Parameter range	Selected parameter
0.909	11.001,11.1	11.05
0.943	10.604,10.7	10.65
0.952	10.504,10.599	10.55
0.961	10.406,10.5	10.45
0.980	10.204,10.297	10.25
0.990	10.101,10.193	10.15

Although it seems that they have 4-digit parameters, notice that all the exceptions' parameters have a zero digit, which means that there are also three digits involved in the pre-scaling; all we need to do is to shift the addends before doing the multiplication. These exceptions can be avoided by setting $n = 4$, but 4-digits' index is slower than 3-digits' index and the area of the 4-digit indexed look-up table will also be larger.

Even with the 3-digit index, the look-up table would have 900 (10^3) entries, which starts from 0.100 to 0.999. This requires a large amount of area and also latency. By specific techniques though, the size of the look-up table can be reduced. As introduced in the Power6 decimal divider, the BCD doubler and quintupler are easy to implement so that the divisor can go through one level of doubler and quintupler, giving outcome D_m shown by Table 3.3

By utilizing this transformation, one ensures that the look-up table only can

Table 3.3 Generation of D_m

Divisor	[0.10,0.20)	[0.20,0.50)	[0.50,1.00)
Multiple	$5 \times D$	$2 \times D$	D
D_m	[0.50,1.00)	[0.40,1.00)	[0.50,1.00)

contain the parameters corresponding to the divisors within $[0.40, 1.00)$. The total number of entries would be reduced from 900 to 600 (6×10^2). Further investigation shows that the adjacent divisors tend to have the same parameters. Actually, there are only 180 different parameters stored in the look-up table, which may leave the compiler some room to reduce more area.

3.3.2 Pre-scaling

The pre-scaling comes after the selection of the partial remainder. After analyzing the parameters of all possible divisors, the range of each digit in the parameter is shown in Table 3.4:

Table 3.4 Parameter's range

MSD	Middle digit	LSD
1,2	0,9	0,9

Since the parameter is to be multiplied by the divisor, a 3-digit parameter means three multiples of divisor and two levels of addition. According to Table 3.4, the MSD is easy to handle because $2D_m$ can be obtained from one level of doubler. However, multiples such as $3D_m$, $6D_m, 7D_m, 8D_m, 9D_m$ are difficult to generate from simple logics. Inspired by [22],one finds that these multiples which cannot be obtained through one level of doubler and quintupler are split into two parts, as shown in the

following equations:

$$
\begin{aligned}
3D_m &= 2D_m + 1D_m \\
4D_m &= 2D_m + 2D_m \\
6D_m &= 5D_m + 1D_m \\
7D_m &= 5D_m + 2D_m \\
8D_m &= 10D_m - 2D_m \\
9D_m &= 10D_m - 1D_m
\end{aligned}
\tag{3.16}
$$

By doing this transformation, all multiples can be calculated based on D_m (notice that $10D_m$ is obtained by shifting D_m), $2D_m$, and $5D_m$. There are then a total of five sums, a situation which needs three level of addition. The optimization of the addition will be discussed in the architecture chapter.

Many efforts have been made on the reduction of addends. Reducing one addend would mean the removal of one decimal CSA and one level of addition. There is a method which can reduce the number of addends by one while the left four addends are within $[-2D_m, -1D_m, 1D_m, 2D_m, 4D_m, 5D_m, 8D_m]$. For instance, in terms of divisor 0.568, the parameter is 17.7. Instead of using $10D_m + 5D_m + 2D_m + 0.5D_m + 0.2D_m$, $10D_m + 8D_m - 0.2D_m - 0.1D_m$ can be used. However, after going through all possible divisors and their parameters, it is found that there are no regular patterns of this kind of transformation, meaning that the look-up table should save all the transformations which will result in a much larger look-up table than the current one. In addition, the calculation of $4D_m$ and $8D_m$ needs three levels of doubler which are both area and time consuming. Consequently, since using the extra one level of addition does not influence the critical path, the transformation to four addends is not necessary.

Notice that the pre-scaling should be applied to the dividend, too, which means the dividend should be multiplied by the factor generating D_m from D, and the split parameter digits. Otherwise the final quotients would not be the correct ones.

3.3.3 Quotient Selection

In the radix-100 divider, the quotients generated in each iteration can be any value within $[-99, 99]$. It is impossible to generate the multiples of divisor for such a wide range. Similar to some high-radix dividers [29] [23], the quotients will be decomposed into two parts following the equation (3.17)

$$q_i = 10q_{Hi} + q_{Li} \tag{3.17}$$

Here, q_{Hi} and q_{Li} are both in the range from -9 to 9. As described before, our implementation uses selection by truncation method, which means the two quotient digits are the first two digits (excluding sign digit) in the partial remainder. However, compared to the straightforward quotient selection, the generation of multiples of the scaled divisor D' is complex. In [22] [23] [24], the multiples of divisor are pre-calculated and saved before iterations. The possibility of calculating the multiples in parallel with the quotient selection is analyzed, but since there is no way to generate those multiples at the same time in a short latency (for instance, the generation of $3D'$), we decide to pre-calculate them and save them in registers before the selection of the first quotient digit.

The problem is that of which multiples should be saved. It is impossible to save all the multiples from $-9D'$ to $9D'$ since generating all those multiples consumes a great amount of time. There are no direct logics to calculate those multiples other than $2D'$ and $5D'$. To deal with this issue, two possible ways can be considered:

1. Using a similar method as that used in the pre-scaling calculation. The two quotient digits can be split into four parts, which are based on the doubler and quintupler of the scaled divisor D'. By using this method, $2D'$ and $5D'$ can be either pre-calculated or calculated in parallel with the quotient selection step.

2. Using a similar method as that used in the Power6 [17] decimal divider. Using the compensation method to deal with the quotients within the range of $[-9, -6], [6, 9]$ while saving $1D', 3D'$, and $4D'$. The calculation of $2D', 5D'$ can be performed in parallel with the quotient selection.

The advantage of the first method is the easy calculation in the pre-scaling step. No further additions are needed after the doubler and quintupler of the scaled divisor. But the drawback is that there will be five addends, including the partial remainder in the iteration. The carry-save adder used in the iteration can handle only two sums and one carry at a time, so five sums mean at least four decimal CSAs and three levels of addition. Therefore, this method is not a good choice if the second one can provide a better solution.

In terms of the second method, assuming that the two compensation values for the two quotient digits are $Comp_H$ and $Comp_L$ respectively, the value of $Comp_H$ can be $\pm 100D'$ while $Comp_L$ can be $\pm 10D'$. The situations which need compensation values are concluded in Table 3.5. In this table, the values of qH, qL and their corresponding compensation values are given.

Table 3.5 Needs of compensations

q_H	q_L	$Comp_H$	$Comp_L$
1,5	6,9	0	$-10D'$
6,9	6,9	$-100D'$	$-10D'$
6,9	1,5	$-100D'$	0
-1,-5	-6,-9	0	$10D'$
-6,9	-6,-9	$100D'$	$10D'$
-6,-9	-1,-5	$100D'$	0

Since two quotient digits are selected in each iteration, the sign of the two digits is the same. For instance, if the quotients are 78, the real calculation should be done through $R_i - 100D' - 10D' + 30D' + 2D'$. Although it seems that there are still five addends involved in the calculation, it is concluded from Table 3.5 that only 6 possible compensation values are needed, which are $\pm 100D', \pm 10D', \pm 110D'$. The first four compensation values $\pm 100D', \pm 10D'$ can be obtained through shifting $\pm D'$ while $\pm 110D'$ can be calculated in the pre-calculation step. Therefore, there will be only four sums to be added together in the iteration, which needs only three DCSAs

and two levels of addition.

The compensation method is chosen to perform the iterations. However, it can deal with only the quotients whose abstract value is larger than 5. The compensation values cannot help avoiding the negative multiples of D'. The negative multiples can be calculated in parallel with the quotient selection. Further analysis shows that the compensation method is helpful to deal with the partial remainder in the form of "carry-save". This point will be discussed later.

4. Architecture

In this chapter, the architecture of the proposed radix-100 divider is shown. Before introducing the overall architecture, it is necessary to describe the architecture of some major blocks. There are two major parts in the overall architecture, the pre-scaling module and the iteration module. They are described in Chapter 4.2 and chapter 4.3, respectively.

4.1 Major Blocks

4.1.1 10's Complement

There should be a way to represent negative values in the radix-100 divider. Normally, negative numbers can be represented by two methods [19]:

1. Using sign and magnitude to represent negative values, which is also known as the signed-magnitude method. In this representation format, the first bit is a sign bit (with no weight), and the others are magnitude bits. For instance, -7 can be represented as 1111 while 7 can be represented as 0111.

2. Using the complement method. The 2's complement representation is widely used in modern ALUs. The 10's complement method uses the similar idea. To describe it in more detail, the 10's complement of an n-digit decimal number X can be obtained through the transformation equation $10^n - X$.

To work with the selected number-system and the representations of the partial remainder as well as the quotient digits, the second method is utilized. After this transformation, all the arithmetics involved in the radix-100 divider (addition, sub-

traction) can be done through decimal adders and the only thing used to distinguish the subtraction from addition is the sign digit which is the MSD of a number.

But it would cause a large time delay if the 10's complement value is obtained through a subtraction as described above. In hardware implementation, 10's complement of a number is usually calculated by adding "1" to its 9's complement. The 9's complement of X is obtained by subtracting each digit from 9. This subtraction, which is usually implemented in a look-up table in hardware, will not bring any carry between digits. Adding "1" is complicated since this operation should be done in a full-range adder, but alternatively, this can be done in separate steps with proper retiming.

In terms of the sign digit, 0 is defined as the positive sign. According to the rules of 9's complement, $9 - 0 = 9$ is defined as the negative sign digit. The sign digit is located at MSD. Table 4.1 represents the rules of obtaining 9's complement values. The block dealing with 9's complement is tagged as "Negative" in the architecture diagram.

Table 4.1 9's complement table

Input digit	0	1	2	3	4	5	6	7	8	9
Output digit	9	8	7	6	5	4	3	2	1	0

4.1.2 Doubler & Quintupler

As shown in the Power6 decimal divider [17], the BCD doubler and quintupler are easy to be implemented. As described in the previous chapter, the doubler and quintupler are widely used in the radix-100 architecture. The reason for the easy implementation of those two modules is that the results of $2X$ and $5X$ where X is a decimal digit are easy to predict. $2X$ is always an even number so the LSB of the result is always zero. This position can be used to represent the carry generated from the other digits. Since the maximum value of $2X$ is 19, the maximum carry is 1. So, the influence of the carry is limited to the LSB.

In terms of performing $5X$, the LSD of the result is either 5 or 0, so the value of each output bit can be predicted through simple logics. This prediction is shown in Table 4.2 where the "x" means "don't care". The values of the output bits are decided by two digits in X. Concluded from this table, only four levels of and/or gates are needed to implement the $5X$ and $2X$ logic. The hardware implementation is a group of logic gates following the logic expressions derived from Table 4.2.

Table 4.2 Algorithm for the quintupler

Assert output bit	Digit i	Digit i-1
[0]	xxx1	0x0x
	xxx0	xx1x
	xxx1	1xxx
[1]	xxx0	x1xx
	xxx1	x01x
	xxxx	x10x
[2]	xxx1	0x0x
	xxx1	x01x
	xxx0	1xxx
[3]	xxx1	x11x
	xxx1	1xxx

4.1.3 BCD DCSA

BCD DCSAs are widely used in the proposed radix-100 divider. Here, the implemented decimal CSA is based on the algorithm described in [38] and [39].

By the assumption that two 4-bit BCD input digits are X_i and Y_i. C_i is a 1-bit carry, the basic equation for the decimal CSA is as follows:

$$(C_{i+1}, S_i) = X_i + Y_i + C_i \tag{4.1}$$

Where the C_{i+1} is the carry output which 10 times of the weight of the current digit position i, S_i is a 4-bit BCD output. The details regarding the calculation and

analysis regarding equation (4.1) are discussed in [38]. Here in this thesis, only the process equations are given below where "j" is within $[0, 3]$:

$$\begin{aligned} g_i[j] =& X_i[j] \cdot Y_i[j] \\ p_i[j] =& X_i[j] + Y_i[j] \\ h_i[j] =& X_i[j] \oplus Y_i[j] \end{aligned}$$

$$\begin{aligned} k_i =& g_i[3] + (p_i[3] \cdot p_i[2]) + (p_i[3] \cdot p_i[1]) + (g_i[2] \cdot p_i[1]) \\ l_i =& p_i[3] + g_i[2] + (p_i[2] \cdot g_i[1]) \\ c_i[1] =& g_i[0] + (p_i[0] \cdot C_i) \end{aligned} \tag{4.2}$$

$$\begin{aligned} S_i[0] =& h_i[0] \oplus C_i[0] \\ S_i[1] =& ((h_i[1] \oplus k_i) \cdot \overline{c_i[1]}) + ((\overline{(h_i[1] \oplus l_i)} \cdot c_i[1]) \\ S_i[2] =& (\overline{p_i[2]} \cdot g_i[1]) + (\overline{p_i[2]} \cdot h_i[2] \cdot \overline{p_i[1]}) + ((g_i[3] + (h_i[2] \cdot h_i[1])) \cdot \overline{c_i[1]}) + \\ & (((\overline{p_i[3]} \cdot \overline{p_i[2]} \cdot p_i[1]) + (g_i[2] \cdot g_i[1]) + (p_i[3] \cdot p_i[2])) \cdot c_i[1]) \\ S_i[3] =& ((\overline{k_i} \cdot l_i) \cdot \overline{c_i[1]}) + (((g_i[3] \cdot \overline{h_i[3]}) + (\overline{h_i[3]} \cdot h_i[2] \cdot h_i[1])) \cdot c_i[1]) \\ C_{i+1} =& k_i + (l_i \cdot c_i[1]) \end{aligned}$$

Eight levels of and/or gates are used to implement the decimal CSA.

4.2 Pre-scaling

The basic function of the pre-scaling module is to transform D and X to D_m and X_m, then to use the first three MSDs of D_m to index a look-up table for the corresponding parameter. As illustrated in Figure 4.1, the inputs to the module are a 16-digit dividend or divisor. An MUX is utilized for the selection of input which is then stored in a register. The pre-scaling cycle starts from the input register and goes into a level of doubler and quintupler that is designed to generate $2D$ and $5D$ while at the same time, the first digit of the divisor is detected. The "detection" means after going through some logics, the divisor should generate three signals indicating

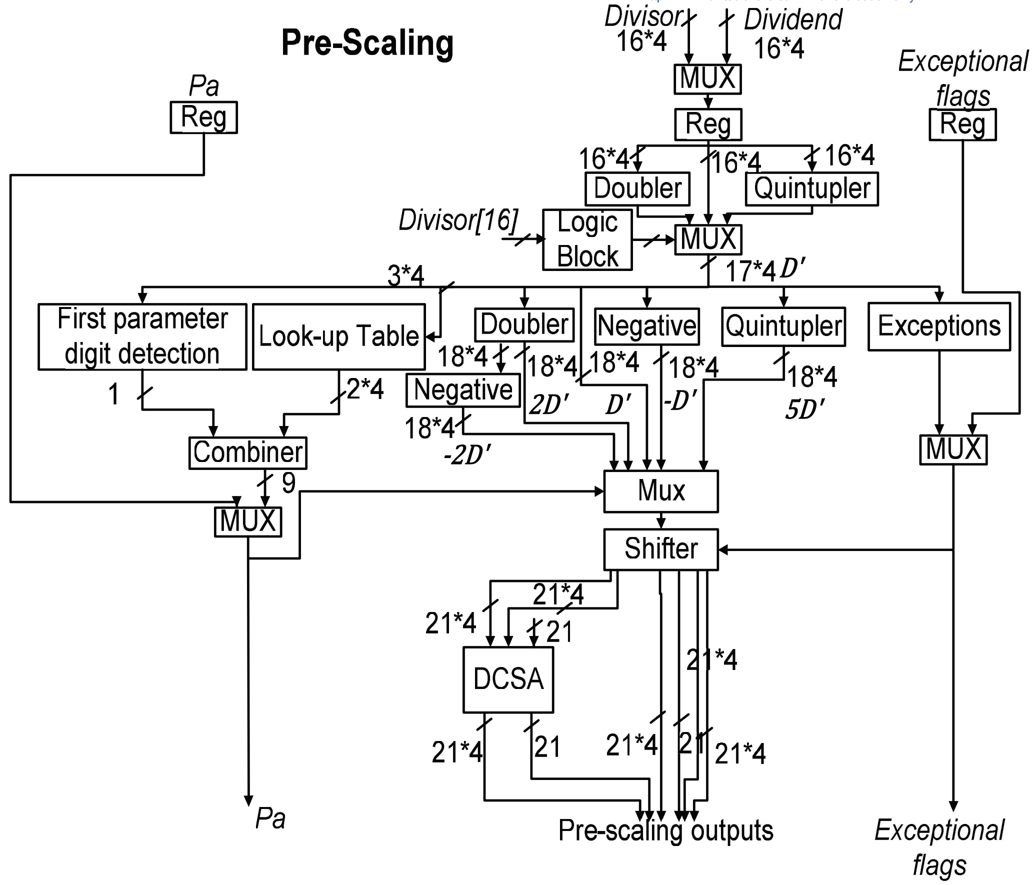

Figure 4.1 Architecture of the pre-scaling module

the three situations described in Table 3.3. These three signals are the inputs to the MUX selecting proper D_m. Notice that the transformation between D and D_m does not change the length of data, since the goal of that transformation is to convert the first digit of D to the range of $[4, 9]$ instead of generating any carry.

Three MSDs of D_m index a loop-up table containing all the parameters. To save the area of the look-up table, only two LSDs of the parameters are saved. We noticed that the MSD of all the required parameters is either 1 or 2. This feature results in a logic ("First parameter digit detection") working in parallel with the look-up table to assert an output signal indicating the condition of the first digit in the parameter. As concluded in equation (3.16), $-2D_m, -1D_m, D_m, 2D_m, 5D_m$ are needed by the split parameter digits. The generation of these multiples is performed in one level of doubler and quintupler followed by a module calculating the 9's complement values

of D_m and $2D_m$. These blocks are again in parallel with the look-up table.

Then, the two LSDs stored in the look-up table are used to select the multiples of D_m through an MUX. There are in fact five MUXes in this step, and each of them can select one multiple of D_m. As described in the last paragraph, one MUX is used to select the multiple of D_m corresponding to the first digit in the parameter. All of these selections happen in parallel. After going through the MUX, five sums and two carries (the "1"s used to generate 10's complement values) are ready to be used. But it should be noticed that there are several exceptions concluded in Table 3.2. The logic (named as "Exceptions" in the figure) detecting whether the divisor is one of the exceptional cases is also done in parallel. After the five sums and two carries are obtained, some of the multiples should be shifted if one of the exceptional flags is asserted.

As mentioned above, five sums should be added through three levels of decimal CSA addition, which would result in a huge latency. The idea to deal with this issue is to decompose the addition into two cycles. One level of addition (just one decimal CSA which can reduce one sum from five sums) is done in the pre-scaling module while the other two levels are done in a separate cycle and in a separate module as well. In fact, to reduce the area consumed by decimal CSAs, the other two levels of addition are done in the DCSAs in the iteration module.

Consequently, the outputs of the pre-scaling module are the parameter selected by D_m, four sums and two carries, and the exceptional flags. When the input to this module is a dividend, the pre-scaling parameter involved in the selection of multiples is the parameter sent from the input.

Overall, the pre-scaling of a divisor needs two cycles. The first cycle is done in the module described in this section. When the pre-scaling of a divisor goes into the second cycle, this module can be used for the pre-scaling of a dividend.

4.3 Iteration

In this section, the architecture of the iteration module is described. There are four major tasks that should be done in the iteration; they are quotients selection, partial remainder calculation, on-the-fly, and rounding. The architecture of the iteration module is shown in Figure 4.2.

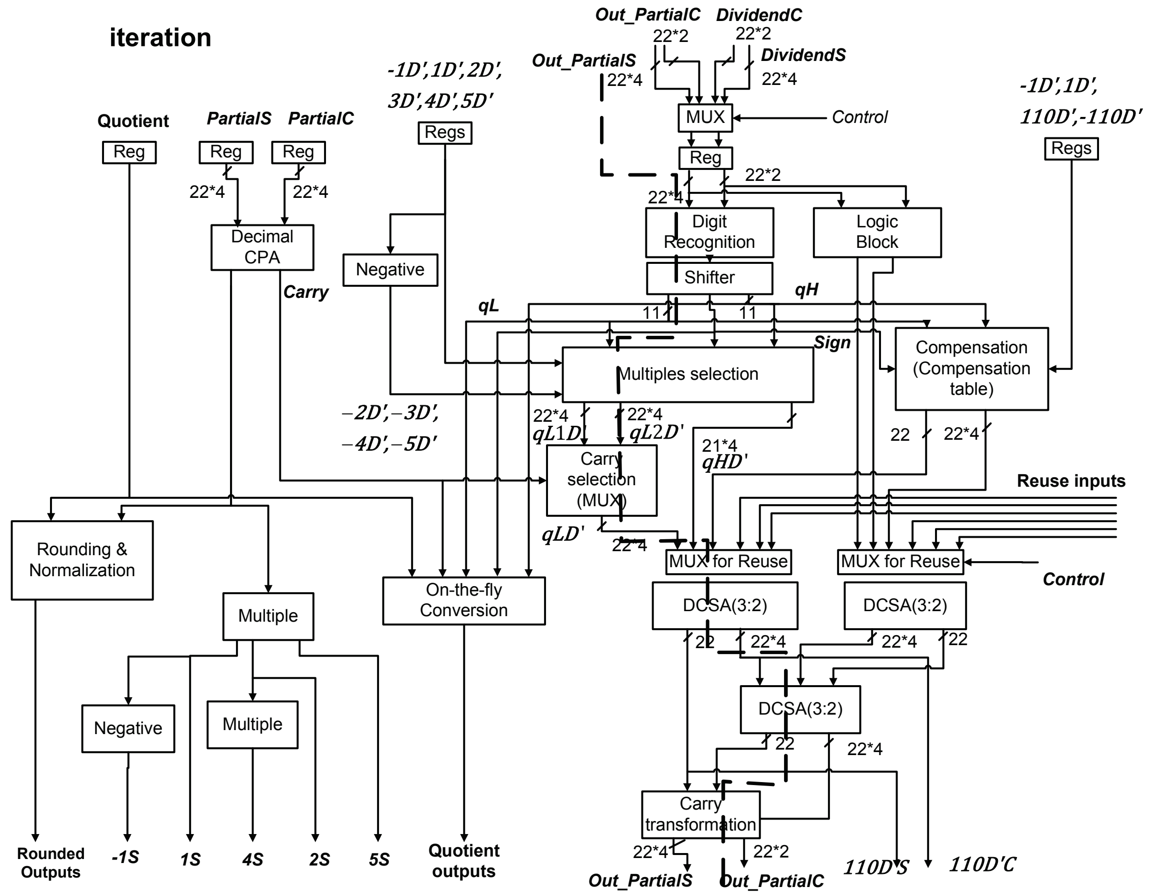

Figure 4.2 Architecture of the iteration module

4.3.1 Digit Recognition

As described before, the decimal CSAs are the major addition blocks in the iteration module. So the basic compositions of the partial remainder should be one 4-bit sum and one 1-bit carry. After some analysis, two- bit carry (maximum value is two) are chosen to reduce one level of decimal CSA addition, which will be explained later.

In the selection by truncation method, if the partial remainder is a compact value, the first two digits of the partial remainder are simply the quotients needed in the iteration. However, dealing with the partial remainder in the form of "sum-carry", some efforts should be made on the quotient selection module. Only the quotient digits and sign digits are considered in this step, while the influence from the other digits (LSDs) in the partial remainder will be dealt with in the next steps. There are two ways of doing the digit-recognition:

1. Using a decimal CPA to add the sign digit and two quotient digits separately, which is done through three 4-bit DCPA. The results are used to select the $qD's$.

2. Using logics to generate a sequence of one-hot coded bits. Each bit represents one value within the ranger from 0 to 9. For instance, there are three situations which can result in number 8: $6+2$, $7+1$, or $8+0$. These circumstances can be represented by a logic expression corresponding to one bit in the one-hot coded sequence.

Even though the first method is quite straightforward, the decimal CPA needs an operand setup which takes time. So, DCPA should be avoided for small additions. Besides, the results generated by the first method should be recoded to fit the MUXes in the following steps. Because of the small range of the carry, the expressions for the second method will not be too complicated and will not take much time. Besides, the one-hot coded results can reduce the complexity of the following MUXes. Consequently, the second method is chosen for the digit-recognition.

However, the second method cannot propagate carries within digits. To represent the cases where carries are involved, the number of bits in the one-hot coded sequences is 11, indicating value 0 to 9, and "larger than 9 (carry)". For instance, if the "sum" of q_L is 9 while its "carry" is 2, the value of q_L should be 11. The bit position 1 and 10 in the sequence are set, meaning "$q_L = 1$" and "Generate a carry" respectively.

The carries will influence the selection of multiples and the sign digit. The first issue is dealt with in the shifter following the step of digit-recognition, while the sign digit is obtained in this digit-recognition module since starting from here, the sign

is used in many places. The exact sign digit can have only two values, which are 9 and 0. Since the maximum carry generated from the other digits is 1, the sign digit without carry can be only 8, 9, or 0. The sign is "guessed" according to Table 4.3. ("x" means don't care, and "carry" means the value is larger than 9)

Table 4.3 Prediction of sign

Sign	Sign digit	q_H	q_L
Minus	8	x	x
Plus	9	carry	x
Plus	9	9	carry
Minus (Don't care)	9	9	9
Plus	0	x	x
Minus	9	others	others

The digits in the partial remainder are all positive, so after taking the influence of carries into consideration, the real sign digit can be only larger than or equal to the value obtained through the logic expression. Consequently, when the sign = 8, there is definitely a carry propagated to it, so that the sign digit can be meaningful. If the sign digit is 0, there will not be any carry changing the sign digit equal to 1 since the sign digit can be only 0 or 9. This is basically how Table 4.3 works.

Even though the sign can be predicted according to Table 4.3, there is one exception that the "carry" propagated to the quotient digits can influence the sign digit. When both of the quotient digits are 9, and the sign digit is also 9, the carry propagated from LSDs can convert the sign from 9 to 0. Since the non-restoring algorithm can tolerate one-unit error, when the sign conversion happens, the next partial remainder is nothing but the shifted current partial remainder while the quotients are treated as $0s$. The on-the-fly module and the rounding module are in charge of the correction. This is the "Don't care" condition described in Table 4.3.

4.3.2 Multiples Selection

There are 6 out of 18 multiples stored in registers before the starting of iteration. These multiples are $-1D', D', 2D', 3D', 4D', 5D'$, which are all generated and saved in the pre-scaling stage. Besides, there are two compensation values stored in registers as well, which are $\pm 110D'$. Even with the help of compensation values, some negative multiples $(-2D', -3D', -4D', -5D')$ are still missing.

The "Negative" block is designed to generate these negative multiples. By the time the value of the sign digit and the one-hot coded sequences are ready, all the multiples ranging from $-5D'$ to $5D'$ are available. With the help of the compensation values, the selection of multiples should follow Table 4.4 and Table 4.5. The values in the parentheses are the compensation values following the rules concluded previously in Table 3.5.

Table 4.4 $q_H D'$ and the corresponding compensation selection

q_H	$q_H D'$ Sign = Positive $q_L \leq 9$	$q_H D'$ Sign = Positive $q_L > 9$	$q_H D'$ Sign = Negative $q_L > 9$	$q_H D'$ Sign = Negative $q_L \leq 9$
0	0	$-10D'$	$-20D'(+100D')$	$-10D'(+100D')$
1	$-10D'$	$-20D'$	$-30D'(+100D')$	$-20D'(+100D')$
2	$-20D'$	$-30D'$	$-40D'(+100D')$	$-30D'(+100D')$
3	$-30D'$	$-40D'$	$50D'$	$-40D'(+100D')$
4	$-40D'$	$50D'(-100D')$	$40D'$	$50D'$
5	$50D'(-100D')$	$40D'(-100D')$	$30D'$	$40D'$
6	$40D'(-100D')$	$30D'(-100D')$	$20D'$	$30D'$
7	$30D'(-100D')$	$20D'(-100D')$	$10D'$	$20D'$
8	$20D'(-100D')$	$10D'(-100D')$	0	$10D'$
9	$10D'(-100D')$	0	$-10D'(+100D')$	0

In Table 4.5, two $q_L D's$ are selected. The $q_{L1}D'$ is selected when the carry generated from the digits other than the sign and the quotient digits is zero, and the $q_{L2}D'$

Table 4.5 $q_{L1}D'$, $q_{L2}D'$ and the corresponding compensation selection

q_L	$q_L D'_1$ Sign = Positive $carry = 0$	$q_L D'_2$ Sign = Positive $carry = 1$	$q_L D'_2$ Sign = Negative $carry = 1$	$q_L D'_1$ Sign = Negative $carry = 0$
0	0	$-1D'$	$-2D'(+10D')$	$-1D'(+10D')$
1	$-1D'$	$-2D'$	$-3D'(+10D')$	$-2D'(+10D')$
2	$-2D'$	$-3D'$	$-4D'(+10D')$	$-3D'(+10D')$
3	$-3D'$	$-4D'$	$-5D'(+10D')$	$-4D'(+10D')$
4	$-4D'$	$-5D'$	$4D'$	$5D'$
5	$5D'(-10D')$	$4D'(-10D')$	$3D'$	$4D'$
6	$4D'(-10D')$	$3D'(-10D')$	$2D'$	$3D'$
7	$3D'(-10D')$	$2D'(-10D')$	$1D'$	$2D'$
8	$2D'(-10D')$	$1D'(-10D')$	0	$1D'$
9	$1D'(-10D')$	$0(-10D')$	$-1D'$	0

is for the case where a propagate carry is generated. From the tables, three features should be noticed:

1. Among all the columns in those two tables, the sequences of multiples are exactly the same, except they are rotated.

2. Only the table for the selection of $q_L D'$ involves the different situations of "carry" while the other table does not care about the "carry". In fact, the selection of $q_H D'$ cares about the carry generated from q_L obtained in the digit-recognition step.

3. The compensation values in Table 4.5 are not influenced by the "carry".

The selection of the multiples is done in MUXes based on the one-hot coded sequences obtained from the digit-recognition block. However, four columns in each table means four MUXes for each quotient digit, which consumes a large area. Because of the first feature, shifters can be used to shift the one-hot coded sequences to certain positions, which can reduce the number of MUXes from 8 to 3.

"Carry" is calculated in parallel through a decimal CPA directly from the beginning of the iteration cycle. However, by the time the one-hot coded sequences are obtained through the digit-recognition block, the "carry" is not ready yet. To eliminate the precious time consumed in waiting for the "carry", two sets of multiples and compensation values can be chosen, one for the case where the "carry" is 1 and the other suits the opposite situation. By the time the "carry" is ready, an MUX will select one from the two sets.

However, two compact sets mean larger area consumption. So, the influence of the "carry" should be analyzed in order to minimize the area. Features 2 and 3 mentioned above show that the influence of the "carry" is limited on the selection of "$q_L D'$", while the selection of the compensation value and the $q_H D'$ are not influenced. This condition is fulfilled by the use of the compensation method, which is described in the following equations (4.3 and 4.4). These equations deal with the positive and negative partial remainder respectively. The quotient digits are X and 9 since this is the only case wherein the "carry" can influence the q_H.

$$\begin{aligned} &Without\text{``}carry\text{''} \\ &\qquad R_i = 100 \times R_{i-1} - 10 \times X \times D' - 9 \times D' \\ &\qquad\quad = 100 \times R_{i-1} - 10 \times X \times D' - 10 \times D' + D' \\ &With\text{``}carry\text{''} \\ &\qquad R_i = 100 \times R_{i-1} - 10 \times (X+1) \times D' \\ &\qquad\quad = 100 \times R_{i-1} - 10 \times X \times D' - 10 \times D' \end{aligned} \tag{4.3}$$

$$\begin{aligned} &Without\text{``}carry\text{''} \\ &\qquad R_i = 100 \times R_{i-1} + 10 \times (9 - X) \times D' \\ &With\text{``}carry\text{''} \\ &\qquad R_i = 100 \times R_{i-1} + 10 \times (8 - X) \times D' + 9 \times D' \\ &\qquad\quad = 100 \times R_{i-1} + 10 \times (9 - X) \times D' - D' \end{aligned} \tag{4.4}$$

Concluded from these equations, only the $q_L D'$ is changed when the "carry" is

changed, while the $q_H D'$ and the compensation value are not influenced. Consequently, two $q_L D's$ ($q_{L1} D'$ and $q_{L2} D'$) are selected through two MUXes in each iteration. After analysis, the "carry" can be obtained right after the selection of the $q_L D's$, so that only one $q_L D'$ is further selected by the "carry" through the "carry selection MUX" before going into the addition stage.

4.3.3 Addition

The addends to be added together for the calculation of a partial remainder are concluded in Table 4.6 where "carry-save" stands for a 4-bit sum and a 1-bit carry, "Sum" means a 4-bit sum, "1" means the "1" added to the 9's complement for the 10's complement values.

Table 4.6 Addends

Name	Form	Additional information
Partial remainder	Carry-save	none
$q_L D'$	Sum	1
$q_H D'$	Sum	1
Compensation	Carry-save	none

The original partial remainder is composed of a 4-bit sum a 2-bit carry. However, there are already four sums to be added, so there is no room for the 2-bit carry. The "logic block" illustrated in Figure 4.2 is used to recode the partial remainder to the "carry-save" format. The latency of this block is smaller than that of the digit-recognition and multiples selection, so the results are already available before the starting of addition.

Since the LSD of the numbers in the form of "carry-save" is not propagated by other digits, the LSB of the "carry" should always be 0. This bit position fits the additional "1s" that needs to be added. Concluded in Table 4.6, two "carry-save" numbers can fit two "1s", which is exactly the number of "1"s in that table. Therefore, four sums and two carries need to be added.

These addends can be fit into two decimal CSAs in the first level. After going through the first level, two sums and two carries are obtained. Then, two sums and one carry can be added by another level of decimal CSA, leaving the other carry "waiting" during the second level of addition. After two levels of addition, two carries and one sum are obtained. Instead of using another level of addition to add them up, a simple logic "carry combination" is utilized to "add" the two carries up, which generates a 2-bit carry and a 4-bit sum. They form the partial remainder of the current iteration and will be used in the next iteration.

4.3.4 Reuse Consideration

Even though the decimal CSA is considered as a fast way of addition, it consumes a large area. Division is performed based on additions so that the decimal CSA is an essential part in the overall architecture, especially for the pre-scaling method which requires several DCSAs for pre-scaling. The multiplication of the pre-scaling parameter, as described before, needs at least four decimal CSAs. The generation of $\pm 110D'$ needs another three DCSAs. In the iteration module as described in the previous section, there are three decimal CSAs. Therefore, 10 DCSAs should be implemented to fulfill the function without consideration of reusing.

By utilizing reuse, only four DCSAs are needed: one in the pre-scaling module while the others are in the iteration module. Even though more areas are consumed by the MUXes to fulfill the reuse, a significant amount of area is saved since the number of DCSAs is reduced. The following two paragraphs describe the reuse involved in the pre-scaling and the calculation of compensation values respectively.

1. Pre-scaling. After going through the DCSA implemented in the pre-scaling module, note that four sums and two carries are obtained from the pre-scaling module. These addends are selected by the MUXes selecting inputs for the DCSAs in the iteration module. Consequently, the addends from the pre-scaling module are added in the addition stage of the iteration module. Exactly like the partial remainder, the output is composed of a 4-bit sum and a 2-bit carry. The sum and carry will be added

up together later. To fulfill the pre-scaling, all of the three DCSAs in the iteration module are reused.

2. Calculation of compensation values. The last cycle in the pre-calculation step is to calculate the compensation values (only $\pm 110D'$). Before this step, $1D'$ and $-1D'$ (9's complement) are already available. Therefore, the calculation of $110D'$ is done by adding $100D'$ and $10D'$, while the calculation of $-110D'$ is to add $-100D'$ and $-10D'$ with two "1"s for 10's complement value. These additions should be done in two levels of DCSAs. In this case, all addends are selected into the addition stage of the iteration module. The inputs to the left DCSA are filled with $100D'$, $10D'$, and 0 (carry). Its outputs are connected directly to the output ports named $110D'$ in the form of "carry-save". The inputs selected for the right DCSA are $-110D'$, $-10D'$, and "1" (carry). Its outputs are sent to the second level of DCSA whose three inputs are "1", sum, and carry generated from the right DCSA in the first level. The "1" is set by one level of gate connecting to one of the outputs of the left DCSA in the first level. The outputs of the second level of DCSA forms the $-110D'$ in the format of "carry-save". Consequently, all three DCSAs are utilized.

Another reuse consideration is the decimal CPA. The decimal CPA located in the iteration module serves to calculate the "carry" generated from the LSDs in the partial remainder. According to Table 4.6, the multiples of D' are in the form of "Sum". But according to the descriptions above, the $1D'$ is reached through the DCSAs in the iteration module whose outputs are in the form of "Carry-sum". A decimal CPA is needed for the calculation of the compact $1D'$. Once the $1D'$ is available, $2D', 4D', 5D'$ can be generated by the doubler and quintupler, while $3D'$ needs another decimal CPA. Other than that, a DCPA is also needed in the rounding cycle to calculate the compact final partial remainder. Consequently, four DCPAs are needed in total. However, with proper retiming, the DCPA in the iteration module can fulfill all the usages discussed above.

The calculation of the compact $1D'$ can be done in the cycle following the calculation of its sum and carry. In the same cycle, the $2D'$,$-D'$,$4D'$,$5D'$ can be obtained

since two levels of doubler can be assigned in the same cycle with the DCPA. In the next cycle, $1D'$ and $2D'$ are selected as the inputs to the DCPA. The output will be $3D'$. During the rounding cycle, the sum and carry of the final partial remainder can be selected into the DCPA, and its output will go through the zero and sign detector needed to select the rounding results. Consequently, the DCPA in the iteration module can be reused in different steps without any conflicts.

4.3.5 On-the-fly Conversion

In our implementation, the on-the-fly method proposed in [40] is used. Originally, the on-the-fly is based on shift registers, which consumes a large number of registers and a huge energy consumption. So, instead of utilizing the traditional on-the-fly method, radix-100 divider saves the quotient digits in their assigned positions in an 18-digits' wide register directly.

In [40], the rounding is also considered. But in the case of the radix-100 divider, the quotients and sign can be obtained before the performing of the addition, which means that the on-the-fly conversion can be done in parallel with the addition. Consequently, at the end of each iteration, the on-the-fly conversion is already done, so the quotients saved in the registers are the correct positive values. Therefore the rounding is separated from the on-the-fly conversion.

An 18-bit register is used to indicate the condition of each digit. During the on-the-fly conversion, every two digits in the quotient are checked at the same time, since they are obtained through the radix-100 divider. For instance, every two adjacent digits can be 00, 01, 10, and 11. They are described in detail in the following paragraphs:

00: This case means that the two quotient digits are not zeros. If the coming new quotient digits are negative, the next flag bit should be checked: "1" means that the next quotient digit is either the end of the consistent zeros, or the place where the new quotients should be saved. In either case, the current two quotients should be subtracted by "1". Flags are set to "00" in all cases.

01: This is the case when the first digit is not zero while the second digit is. If the new coming digits are negative, the digits in the current position should be subtracted by 1, and the flags are set to “00”. Otherwise, if the new digits are not two zeros, the flags are set to “00” directly.

10: This means the current digit positions are the positions where new quotient digits should be saved. If the sign is negative while the two quotients are nines, the flags are set to “11” while “00” are stored as quotient digits. If the sign is positive, the flags are set to “00”, “01”, or “11” corresponding to different cases of the new quotient digits. Meanwhile, the next two flag bits are set with “10” indicating the positions of the digits from next iteration.

11: This means both of the digits are zeros. If the coming digits are negative, the current digits under these flags are set to “9”. If the coming digits are positive and not zeros, the flags are set to “00”. Otherwise, they are set to “11”.

4.3.6 Rounding

By the end of the last iteration (the generation of the 17th, 18th quotient digit), the values available for rounding are the 18-digit quotient and the final partial remainder in the form of a 4-bit sum and a 2-bit carry. The function of rounding is to determine whether the final quotient should be added by “1” or not.

As described in the first chapter, roundTiesToEven is the recommended rounding mode for the decimal arithmetic. The proposed radix-100 divider also utilizes this rounding method which follows the rules in Table 4.7, where QP stands for adding one to the current 16-digit quotient, and Q is the current quotient digits. “Negative R” means that the final partial remainder is negative while the “Positive R” means that the final partial remainder is larger than zero.

The rounding is performed in the last cycle of the whole division calculation. As discussed in equation (3.13), the shifted dividend is smaller than 1 while the pre-scaled divisor is larger than 1 but smaller than $1 + 1/99$. Consequently, in extrame

Table 4.7 Rounding rules

$Rounddigit$	$NegativeR$	$PositiveR$	$R = Zero$
0	Q	Q	Q
1	Q	Q	Q
2	Q	Q	Q
3	Q	Q	Q
4	Q	Q	Q
5	Q	QP	Q(Q[LSB]=0) QP(Q[LSB]=1)
6	QP	QP	QP
7	QP	QP	QP
8	QP	QP	QP
9	QP	QP	QP

cases, it is quite possible that the final quotient is in the form of $0.0xxxxx$ instead of $0.xxxxxx$, which means a leading zero is generated. Although the IEEE 754-2008 does not require normalization (removal of the leading zero), considering further usage of the quotients, it is better to remove the leading zero.

As described in [19], rounding is based on three digits, which are a guard digit, an round digit, and a sticky digit. The IEEE 754-2008 decimal 64 format needs 16-digit correct quotient, consequently, 18 digits are needed to do the rounding. In [23], an extra cycle is needed for the removal of leading zero, since decimal dividers can only generate one quotient digit at a time. However, in the radix-100 divider, 18 quotient digits can be generated through 9 iterations.

In the rounding cycle, a shifter is utilized to shift the leading zero out of the quotient if there is one at the MSD position. A 17-digit sequence should be generated by the shifter. Meanwhile, components of the final partial remainder are added up through the DCPA. The successive "9"s starting from the 16th digit to the MSD in the quotient are marked. After that, all the marked digits are changed to "0" while

the digit whose right digit is the last marked digit is added by 1. This step generates QP. The final rounded quotient can be selected by the results from the DCPA.

The partial remainder mentioned above is in fact the final partial remainder, while there is no partial remainder for the 17th digit. If there is no leading zero in the quotient, the partial remainder of 17th digit should be used. However, only the sign or the zero condition of the partial remainder is needed, which can be derived from the final partial remainder and the value of the 18th quotient digit. Only if the 18th digit is zero, the partial remainder of the 17th digit has the same sign and the same zero condition as the final partial remainder; otherwise, it is always positive since the final partial remainder follows equation (3.9). To explain this point in detail, even if the final partial remainder is negative, its abstract value must be smaller than D'. If the 18th digit is not zero, adding the "18th digit $\times D'$" back to the final partial remainder must result in a positive value that is the partial remainder of the 17th digit.

4.4 Operation Sequence

The overall operation sequence and the tasks in each cycle are summarized below:

Cycle 1: Divisor is imported to the pre-scaling module. Parameter Pa is obtained and the outputs (four sums and two carries) are saved.

Cycle 2: Task 1: A dividend is imported to the pre-scaling module. The parameter obtained from the first cycle is used here for the dividend. Task 2: The four sums and two carries got from the previous cycle are added up by the DCSAs in the iteration block. The output is $1D'$ in the form of "carry-sum".

Cycle 3: Task 1: The addends of the scaled dividend are added up in the DCSAs. The outputs are saved in a pair of registers to be used as R_0 in the first iteration. Task 2: The two parts representing $1D'$ are added up in the DCPA in the iteration block. After going through two levels of multiple logic and one level of negative logic, $1D'$, $2D'$, $4D'$, $5D'$,$-1D'$ are obtained and saved in registers.

Cycle 4: Task 1: $1D'$ and $2D'$, which are obtained from cycle 3, are selected into the DCPA to generate $3D'$. $3D'$ is saved directly into register. Task 2: $100D'$ and $10D'$ are added in one DCSA while $-100D'$ and $-10D'$ are added with two "1"s in the other two DCSAs. After these additions, $110D'$ and $-110D'$ are obtained and saved.

Cycle 5: The carry and sum of the scaled dividend are selected as the inputs to the iteration module. Also, the values are assigned to inputs of the DCPA. After this iteration, the first two quotient digits are obtained.

Cycle 6 to Cycle 13: There are 8 iteration cycles in total. They are same as the cycle 5, except that the inputs are the outputs from the previous iteration. During the calculation of the partial remainder, the on-the-fly module saves the quotient digits selected in the current iteration to the right places with appropriate modifications on the existing quotient digits.

Cycle 14: Two parts of the partial remainder from the previous iteration are added up. The result then goes through a "sign and zero detection" module. Meanwhile, a logic block serves to add 1 to the current quotient.

Overall, the radix-100 divider requires 14 cycles to produce a correct 16-digit quotient.

5. Implementation and Comparison

The proposed radix-100 divider, as illustrated in Figures 4.1 and 4.2, is modeled with Verilog, simulated in Modelsim and verified by System Verilog and MATLAB. 300,000 random cases are tested and verified. Finally, the design is synthesized by using STM 90-nm CMOS standard cells library with typical conditions (1.2 VDD core voltage and $25^{o}C$ operating temperature) in Synopsys Design Compiler. The clock, interfaces are assumed to be ideal. The synthesis tool computes the best latency.

5.1 Synthesis Results

After synthesis, the critical path is found in the iteration cycle as shown in Figure 4.2. Note that although the design compiler reports the cycle time in ns, it is the logic effort that can estimate the delay values in a technology independent parameter. The delay of an inverter of the minimum drive strength with a fanout of four 1x inverters (FO4) is used as the basic delay model. The area measurement metric is also transformed to NAND2 (two input NAND gate) from μm^2 which is the unit in the design compiler report. In the STM 90nm CMOS library, the basic equations for the units described above are

$$\begin{aligned} 1FO4 &\approx 45ps \\ 1NAND2 &\approx 4.4\mu m^2 \end{aligned} \tag{5.1}$$

The detail information of the critical path of the radix-100 divider is summarized in Table 5.1. Figure 4.2 illustrates the critical path on the architecture of iteration.

The area is 139,812 μm^2, which includes 111,094 μm^2 (80%) combinational logic

Table 5.1 Critical Path

Component	latency(ns)	latency (FO4)	%
Register	0.1	2.22	8.2%
Buffer	0.03	5.78	2.5%
Digit-recognition	0.13	2.89	10.7%
Shifter	0.16	3.56	13.1%
Multiples selection	0.2	4.4	16.4%
MUXes	0.08	1.78	6.5%
DCSAs	0.33	7.3	27.0%
Carry transformation	0.06	1.3	4.9%
Others	0.04	0.89	3.3%
Setup	0.09	2	7.4%
Total	1.22	27	100%

while the others are registers. The pre-calculation saved many values into registers, which results in a relatively large register area. The total area is 31,458 NAND2.

We used the same libraries and conditions as used for the radix-100 divider to synthesis [24] and [26]. Since these works are originally synthesised in TSMC 0.13 μm library, it is unfair to compare different works directly if they have different implementation environments. The divider in [23] used the same library as the radix-100 divider. Therefore, the data described in that paper can be used directly. In terms of the Power6 decimal divider, since we don't have the detailed architecture description, the latency represented in FO4 provided by [24] is utilized and transformed to ns, and the number of cycles is obtained by using the redundant adder.

Table 5.2 presents all the comparisons among different designs. Notice that reference [26], [24], and [23] utilize SRT division algorithm while [17] is based on the selection by truncation method. All the designs except the proposed radix-100 divider are decimal dividers.

Table 5.2 Results Comparison

Divider	Cycle time (ns)	No.of Cycles	Latency (ns)	Ratio	Area (NAND2)	Ratio
Radix-100	1.22	14	17.08	1	31,458	1
[26]	0.88	20	17.6	1.03	11,130	0.35
[24]	0.93	21	19.5	1.14	11,000	0.35
[23]	1	20	20	1.17	13,500	0.43
[17]	0.585	48	28	1.64	-	-

5.2 Comparison

According to Table 5.2, the proposed radix-100 divider is 3% faster than the latest decimal divider [26]. In the following sections, the major components in the critical path are analyzed and compared with those of other decimal dividers. In the end, the area is also analyzed.

5.2.1 Pre-calculation

All the designs included in Table 5.2 need pre-calculation. For instance, as described in Chapter 2, the pre-calculation of the divider in [23] takes one cycle to calculate and register the multiples of the divisor ($\pm 5D$, $\pm 2D$). Similarly, [26] and [24] require one cycle for the pre-calculation to generate the multiples they need. Especially in [26], multiples $-5D$ to $5D$, which are the same multiples needed in the radix-100 divider, are generated in one cycle.

Since no pre-scaling is needed by SRT, the divisor provided as input is the one used in the iteration. However, the selection by truncation method does not support one-cycle initialization. The generation of multiples must wait for the pre-scaling, which results in a large number of pre-calculation cycles. This delay is a major reason why the proposed radix-100 divisor does not show significant improvement of performance compared with other decimal SRT based dividers.

It is true that the radix-100 divider can reduce half of the iteration cycles (18 cycles to 9 cycles), but note that four cycles are used for the pre-calculation. In other

words, the cycles used for the pre-scaling take 28.6% of the total latency. In [24], [26] and [23], only one cycle of pre-calculation is needed, which takes at most 5% of the total latency.

The decimal divider in Power6, which utilizes the same selection by truncation method, uses 12 out of 48 cycles for the pre-scaling.

Consequently, one of the drawbacks of the selection by truncation is the large number of cycles consumed for the pre-calculation. In the Power6 decimal divider, a pipeline technique is utilized. Similarly, the large number of pre-calculation cycles can be fully used through pipeline. As described in [30], higher radix works better in terms of pipeline. Therefore, the latency consumed by the pre-scaling would be smaller if pipelined inputs are provided to the radix-100 divider.

5.2.2 Quotient Selection

There are two basic methods used by the designs shown in Table 5.2. Power6 [17] and the proposed radix-100 divider utilize the selection by truncation while the others are based on SRT.

SRT, as discussed in Chapter 1, is based on the comparisons between the partial remainder and the boundaries. All three SRT designs in Table 5.2 utilize CSAs to do the comparisons and prefix-trees to decide the signs. Besides, there should be a coder used to generate a signal helping to choose the multiple of divisor needed in the next iteration. In fact, the SRT used in these designs can be understood as a calculation of the current partial remainder plus a quotient selection.

To reduce the cycle time, all of those designs separate the quotient selection step and the calculation of the current partial remainder. Consequently, the cycle time shown in Table 5.2 can be understood as the latency of the quotient selection module. [26] combines the binary calculation technique in [23] and the digit set from [24]. Its cycle time is 0.69 ns, which is better than that of [23] and [24]. But only one quotient digit is selected within this latency while almost double of this time is needed in the

cascaded high-radix method [29].

Looking back to the proposed radix-100 divider, only 0.53 ns is needed to select two quotient digits. In addition, the latency consumed on selection by truncation is slightly influenced by the number of truncated digits. Consequently, even for higher radix divider, the latency of the quotient selection would be still around 0.53ns.

As a result, the selection by truncation method is suitable for very high-radix division from the view of the quotient selection.

5.2.3 Addition

The additions in [24], [23]and [26] are done in parallel with the quotient selection so they are not part of the critical path.

In both [17] and the proposed radix-100 divider, the addition is done in the second half part of each iteration. In the proposed radix-100 divider, two levels of addition and one carry combination block are needed. These modules take 0.39 ns in total. Because of the cycle time saved by the selection by truncation method, the addition does not increase the cycle time significantly compared to those SRT decimal dividers.

It is found that carry-save adder is definitely a good choice for multiple addends' addition in the radix-100 divider.

5.2.4 Area

Another issue to compare is the area. As shown in Table 5.2, the area consumed by the radix-100 divider is more than two times higher than that of the decimal dividers. In fact, high-radix division always means larger areas, since more operands are involved in each iteration (radix-100 needs more addends than the decimal dividers). As shown in [29], the radix-16 combined div/sqrt utilized in Penryn processor is around 70% larger than the radix-4 combined div/sqrt unit. Other than that, applying high-radix to the decimal field is more complex than the use of high-radix in the binary field. For instance, different decimal encodings can influence area, which

is not an issue in the binary field.

The pre-scaling module is found to take about 20% of the total area, which is a major source of the total area. In addition, there are three sources contributing to the large area: the DCSAs, the widely used MUXes, and the DCPA. The BCD encoding (low efficiency) also results in the large area.

5.3 Conclusion

The synthesised results show that the proposed radix-100 divider is 3% faster than the latest decimal divider [26]. More than 28% of the total latency of the radix-100 divider is consumed on the pre-calculation step, which can be reduced if pipeline is added. Therefore, even if the selection by truncation method is not efficient in terms of the pre-calculation, it is definitely a better solution than the SRT algorithm for the radix-100 divider if proper pipeline is utilized. One drawback of the radix-100 divider is that the radix-100 divider consumes a large area. This is caused by the utilization of the basic BCD coding style (although it contributes the easy quotient selection). The use of the unsigned partial remainder digit ($[0, 9]$) and the widely used registers are two other reasons. It is possible that by utilizing more advanced techniques in the future, radix-100 can be much faster than the highly developed decimal dividers.

6. Summary

Due to the increasing demand for fast and precise decimal divisions, designing a fast decimal divider has recently become a popular trend and a promising field. After analyzing some of the recent published dividers, instead of finding more tricks to be applied to the current dividers, it was decided to try to improve the performance of decimal dividers through another way, the high-radix. Proven in the binary field, the radix-16 divider has better performance than the radix-4 dividers [29], and the same concept can be applied to the decimal field.

Radix-100 divider is supposed to be a divider that can produce two decimal quotient digits in each cycle. Compared to the decimal dividers using the decimal format defined in IEEE 754-2008, radix-100 divider can reduce the iteration cycles by half. The cycle time should be smaller than two times of the cycle time of the decimal dividers.

A radix-100 divider is proposed in this thesis. As a first trial of applying the high-radix to the decimal dividers, the proposed divider has the following features:

1. The non-restoring and selection by truncation method are utilized. It is found that the popular SRT is not suitable for the radix-100 divider especially from the point of cycle time. Similar to the Power6 decimal divider, the non-restoring based selection by truncation method is used.

2. Partial remainder is in the form of sum and carry. To reduce the large latency consumed on calculating the compact partial remainder, DCSAs are widely used in the design to make addition as fast as possible.

3. Compensation method. To reduce one level of addition and the area, the compensation method is utilized to work with the multiples of scaled divisor saved in the registers.

4. Reuse. Many efforts have been made on the reuse, which can reduce the area significantly. By reusing the DCSAs and the DCPA, the number of DCSAs is reduced from 10 to 4 and the number of DCPAs is reduced from 4 to 1.

5. IEEE 754-2008 support. This design is based on Decimal 64 standard. Modules dealing with rounding and normalization are included.

The proposed radix-100 divider is faster than the current decimal dividers. In terms of the overall latency, the radix-100 divider is 3% faster than the latest decimal divider [26], and is 14% and 17% faster than two typical decimal dividers [24] and [23] respectively. The proposed radix-100 divider introduces a brand-new way of treating decimal dividers. Compared with a very early decimal divider proposed in 2006 [22], the latest decimal divider [26] achieves a much better performance than the old ones. Consequently, with the start point that the proposed radix-100 divider is slightly faster than the latest decimal divider, it is quite likely that the radix-100 divider can be much faster in the future by applying more techniques on the radix-100 divider.

Also, the proposed design proves that the selection by truncation, which is not as popular as SRT, is a good choice for very high radix division because of its fast quotient selection method. Even though with longer pre-calculation latency, selection by truncation can derive the quotient digits (regardless of the number) in a much shorter time than the SRT method. In the radix-100 divider, the quotient selection module can select two multiples of scaled divisor within 0.53ns while the fastest decimal divider needs 0.69ns to select one multiple of divisor. In addition, it is found that SRT is not suitable for the very high radix division since high radix division means more comparisons, larger areas and longer latency.

The area consumed by the radix-100 divider is 2.85 times of the smallest decimal divider. The major reasons for this large area are the use of BCD encoding and pre-

scaling. The pre-scaling module takes around 20% of the overall area. Although the reuse concept reduces the area significantly, there are still four DCSAs in the overall architecture.

The design utilizes many techniques derived from some typical previous works as described in Chapter 2. Also, it uses some already available decimal arithmetic components. By applying more techniques, the future of the radix-100 divider in industry area is promising.

7. Future Work

Some future work may help the proposed radix-100 divider reaching a better performance and a lower area. One way of doing this might be the usage of signed-digit.

Signed-digit is a form of redundant number system which has a sign bit. For instance, "1001" means "-7" while "0111" means "+7". The advantages of the signed-digit are:

1. The calculations of signed-digit numbers can be performed by many binary arithmetic modules. For instance, "$1001 + 1001 = 10010 = -14$" is done through binary addition.

2. With a proper selection of digit-set, the determination of "carry" in the current design can be removed. For instance, the sum locates in $[-5, 4]$ while the carry is in $[-2, 2]$. The compact partial remainder digit ranges from -7 to 6. The addition of the sum and the carry will not generate carries within digits.

Some drawbacks of using the signed-digit:

1. More complex quotient selection. The digit-recognition module should consider "borrow" as well as "carry". Also, it needs to transform the signed quotient digits to the one-hot coded sequence as discussed before, which may consume longer latency.

2. The multiples of scaled divisor should be transformed to the signed-digit format.

Another future work is adding a pipeline to the current design. As described before, pipeline can eliminate the influence of the pre-scaling and fully use all the

modules.

References

[1] H. Goldstine and A. Goldstine, "The Electronic Numerical Integrator and Computer (ENIAC)," *Annals of the History of Computing, IEEE*, vol. 18, no. 1, pp. 10 –16, 1996.

[2] D. E. Knuth, "The IBM 650: An Appreciation from the Field," *Annals of the History of Computing*, vol. 8, pp. 50 –55, Jan.-Mar. 1986.

[3] IBM, "Decimal Arithmetic Hardware Questions," *Available : http://speleotrove.com/decimal/decifaq3.html#hwsupport*, 2010.

[4] A. V. Alvarez, "High-performance Decimal Floating-Point Units," *University of Santiago de Compostela*, Jan. 2009.

[5] L. Eisen, J. W. Ward, 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, pp. 1 –21, Nov. 2007.

[6] IBM, "IBM z9 EC and z9 BC - Delivering greater value for everyone," *Available : http://www-01.ibm.com/common/ssi/rep_ca/6/877/ENUSZG07-0286/ENUSZG07-0286.PDF*, 2007.

[7] 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, pp. 4:1 –4:10, Jan. 2009.

[8] F. Busaba, C. Krygowski, W. Li, E. Schwarz, and S. Carlough, "The IBM z900 decimal arithmetic unit," in *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, vol. 2, pp. 1335 –1339, Nov. 2001.

[9] S. Oberman and M. Flynn, "Design issues in division and other floating-point operations," *Computers, IEEE Transactions on*, vol. 46, pp. 154 –161, Feb. 1997.

[10] M. Cowlishaw, "Decimal floating-point: algorism for computers," in *Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on*, pp. 104 – 111, June 2003.

[11] IBM, "Decimal Arithmetic General Questions," *Available : http://speleotrove.com/decimal/decifaq3.html#softgood*, 2012.

[12] D. Chen, "Algorithms and architectures for decimal transcendental function computation," *University of Saskatchewan*, Sept. 2010.

[13] Wikepedia, "IEEE floating-point," *Available : http://en.wikipedia.org/wiki/IEEEfloating_point*, 2012.

[14] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1 –58, 2008.

[15] M. Cowlishaw, "Densely packed decimal encoding," *Computers and Digital Techniques, IEE Proceedings* -, vol. 149, pp. 102 –104, May 2002.

[16] Wikipedia, "Densely Packed Decimal," *Available : http://en.wikipedia.org/wiki/Densely_packed_decimal*, 2012.

[17] E. Schwarz and S. Carlough, "Power6 Decimal Divide," in *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pp. 128 –133, July 2007.

[18] Wikipedia, "Binary Integer Decimal," *Available : http://en.wikipedia.org/wiki/Binary_Integer_Decimal*, 2012.

[19] I. Koren, *Computer Arithmetic Algorithms.* A K Peters,Ltd., 2002.

[20] Wikipedia, "NaN," *Available : http://en.wikipedia.org/wiki/NaN*, 2012.

[21] S. Obermann and M. Flynn, “Division algorithms and implementations,” *Computers, IEEE Transactions on*, vol. 46, pp. 833 –854, Aug. 1997.

[22] H. Nikmehr, B. Phillips, and C.-C. Lim, “Fast Decimal Floating-Point Division,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, pp. 951 –961, Sept. 2006.

[23] T. Lang and A. Nannarelli, “A Radix-10 Digit-Recurrence Division Unit: Algorithm and Architecture,” *Computers, IEEE Transactions on*, vol. 56, pp. 727 –739, June 2007.

[24] A. Vazquez, E. Antelo, and P. Montuschi, “A radix-10 SRT divider based on alternative BCD codings,” in *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pp. 280 –287, Oct. 2007.

[25] J.-P. Deschamps and G. Sutter, “Decimal division: Algorithms and FPGA implementations,” in *Programmable Logic Conference (SPL), 2010 VI Southern*, pp. 67 –72, Mar. 2010.

[26] A. Kaivani, A. Hosseiny, and G. Jaberipur, “Improving the speed of decimal division,” *Computers Digital Techniques, IET*, vol. 5, pp. 393 –404, Sept. 2011.

[27] L.-K. Wang and M. Schulte, “Decimal floating-point division using Newton-Raphson iteration,” in *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pp. 84 – 95, Sept. 2004.

[28] C. Wey, “Design of fast high-radix srt dividers and their vlsi implementation,” *Computers and Digital Techniques, IEE Proceedings* -, vol. 147, pp. 275 –282, July 2000.

[29] A. Nannarelli, “Radix-16 Combined Division and Square Root Unit,” in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pp. 169 –176, July 2011.

[30] H. B. James Coke, “Improvements in the Intel Core2 Penryn Processor Family Architecture and Microarchitecture,” *Intel Technology Journal*, vol. 12, pp. 179–192, Oct. 2008.

[31] N. Srivastava, “Radix 4 SRT Division with Quotient Prediction and Operand Scaling,” in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pp. 1 –6, Apr. 2007.

[32] M. Baesler, S. Voigt, and T. Teufel, “FPGA Implementations of Radix-10 Digit Recurrence Fixed-Point and Floating-Point Dividers,” in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pp. 13 –19, Dec. 2011.

[33] I. Castellanos and J. Stine, “Experiments for Decimal Floating-Point Division by Recurrence,” in *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, pp. 1716 –1720, Nov. 2006.

[34] P. Kornerup, “Revisiting SRT quotient digit selection,” in *Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on*, pp. 38 – 45, June 2003.

[35] E. Antelo, T. Lang, P. Montuschi, and A. Nannarelli, “Digit-recurrence dividers with reduced logical depth,” *Computers, IEEE Transactions on*, vol. 54, pp. 837 –851, July 2005.

[36] M. Baesler, S.-O. Voigt, and T. Teufel, “A radix-10 digit recurrence division unit with a constant digit selection function,” in *Computer Design (ICCD), 2010 IEEE International Conference on*, pp. 241 –246, Oct. 2010.

[37] M. Ercegovac, T. Lang, and P. Montuschi, “Very-high radix division with prescaling and selection by rounding,” *Computers, IEEE Transactions on*, vol. 43, p-p. 909 –918, Aug. 1994.

[38] M. Schmookler and A. Weinberger, “High Speed Decimal Addition,” *Computers, IEEE Transactions on*, vol. C-20, pp. 862 – 866, Aug. 1971.

[39] M. Erle and M. Schulte, “Decimal multiplication via carry-save addition,” in *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, pp. 348 – 358, June 2003.

[40] A. Nannarelli and T. Lang, “Low-power divider,” *Computers, IEEE Transactions on*, vol. 48, pp. 2 –14, Jan. 1999.