

Algorithms and Hardware Designs for Decimal Multiplication

by

Mark A. Erle

Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy

in
Computer Engineering

Lehigh University

November 21, 2008

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Date

Accepted Date

Dissertation Committee:

Dr. Mark G. Arnold
Chair

Dr. Meghanad D. Wagh
Member

Dr. Brian D. Davison
Member

Dr. Michael J. Schulte
External Member

Dr. Eric M. Schwarz
External Member

Dr. William M. Pottenger
External Member

This dissertation is dedicated to Michele, Laura, Kristin, and Amy.

Acknowledgements

Along the path to completing this dissertation, I have traversed the birth of my third daughter, the passing of my mother, and the passing of my father. Fortunately, I have not been alone, for I have been traveling with God, family, and friends.

I am appreciative of the support I received from IBM all along the way. There are several colleagues with whom I had the good fortune to collaborate on several papers, and to whom I am very grateful, namely Dr. Eric Schwarz, Mr. Brian Hickmann, and Dr. Liang-Kai Wang. Additionally, I am thankful for the guidance and assistance from my dissertation committee which includes Dr. Michael Schulte, Dr. Mark Arnold, Dr. Meghanad Wagh, Dr. Brian Davison, Dr. Eric Schwarz, and Dr. William Pottenger. In particular, I am indebted to Dr. Schulte for his mentorship, encouragement, and rapport.

Now, finding myself at the end of this path, I see two roads diverging...

Contents

List of Tables	ix
List of Figures	xi
List of Acronyms	xiii
Abstract	1
1 Introduction	3
1.1 Motivation for Decimal Computer Arithmetic	4
1.2 Overview of Research	9
1.3 Significance of Research	11
1.4 Outline of Dissertation	13
2 Background of Decimal Computer Arithmetic	15
2.1 History of Decimal Computer Arithmetic	16
2.1.1 Decimal Numbers	16
2.1.2 Binary Numbers	16
2.1.3 Fixed-point Numbers	17
2.1.4 Scaled Fixed-point Numbers	18
2.1.5 Floating-point Numbers	19
2.1.6 Early Computer Arithmetic Systems	20
2.2 Software Support of Decimal Arithmetic	23
2.3 Processor Support of Decimal Arithmetic	27
2.4 IEEE 754-2008 Standard	30
2.4.1 Differences Between BFP and DFP	32
2.4.2 Decimal Formats	34
2.4.3 Rounding	37
2.4.4 Exceptions	38
3 Related Research	41
3.1 Decimal Encodings	42
3.1.1 Digit Encodings	42

3.1.2	Significand Encoding	45
3.2	Decimal Addition	48
3.2.1	Bias, Binary Addition, and Correction	49
3.2.2	Binary Addition and Correction	52
3.2.3	Direct Decimal Addition	53
3.2.4	Redundant Addition	55
3.2.5	Subtraction via End-Around Carry Addition	59
3.3	Decimal Multiplication	63
3.3.1	Digit-by-Digit Multiplication	63
3.3.2	Word-by-Digit Multiplication	67
3.3.3	Word-by-Word Multiplication	75
3.3.4	Decimal Floating-point Multiplication	76
4	Iterative Multiplier Designs	78
4.1	Fixed-point Designs	80
4.1.1	Multiplier Employing Decimal CSAs	80
4.1.2	Multiplier Employing Signed-Digit Adders	93
4.1.3	Summary of Iterative DFXP Designs	111
4.2	Floating-Point Design	112
4.2.1	Algorithm	112
4.2.2	Features	115
4.2.3	Implementation and Analysis	131
4.2.4	Summary	134
5	Parallel Multiplier Designs	135
5.1	Fixed-point Designs	136
5.1.1	Multiplier Employing Decimal Carry-Save Adders	136
5.1.2	Multiplier Employing Binary Carry-Save Adders	142
5.1.3	Summary of Parallel DFXP Designs	148
5.1.4	Combined Binary/Decimal, Fixed-point Design	149
5.2	Floating-point Design	158
5.2.1	Algorithm	158
5.2.2	Features	161
5.2.3	Implementation and Analysis	165
5.2.4	Summary	167
5.3	Analysis of Iterative and Parallel Designs	168
6	Conclusion	171
6.1	Summary	172
6.2	Future Research	175
6.3	Closing	179
A	Glossary	197

B Notation	209
C Vita	215

List of Tables

1.1	Successive Division of Nine by Ten [1]	6
2.1	Time Line of Early Computer Systems and Notable Events [2]	22
2.2	Software Support of Decimal Arithmetic	25
2.3	Contemporary Processor Support of Decimal Arithmetic	29
2.4	Preferred Exponent of Select Decimal Operations	34
2.5	DFP Format Parameters	36
2.6	DFP Format Ranges	36
2.7	Combination Field for DFP Representations	37
2.8	Rounding Mode Descriptions	38
3.1	Select Binary-Coded Decimal Encodings	43
3.2	Some Binary-Coded Decimal Values	44
3.3	Some Signed-Digit Codes	45
3.4	Encoding a Densely Packed Decimal Declet [3]	46
3.5	Decoding a Densely Packed Decimal Declet [3]	47
3.6	Generation of Primary Multiples from Different Multiples Sets	72
4.1	Generation of Primary Multiples from A , $2A$, $4A$, and $5A$	88
4.2	Area and Delay of Iterative DFXP Multiplier (Decimal CSAs)	92
4.3	Complexity of Digit-by-Digit Products for Ranges of Decimal Inputs	96
4.4	Restricted-Range, Signed-Magnitude Products	103
4.5	Restricted-Range, Signed-Digit Sums [4] (All Digits Are Decimal)	106
4.6	Rounding Modes, Conditions, and Product Overrides for Overflow	123
4.7	Area and Delay of Iterative Multipliers (DFXP vs. DFP)	134
5.1	Multiplier Operand Digit Recoding Scheme [5]	138
5.2	Area and Delay of DFXP Multipliers (Iterative vs. Parallel)	141
5.3	Multiplier Operand Digit Recoding Schemes [6]	145
5.4	Area and Delay of DFXP Multipliers (Iterative vs. Parallel)	148
5.5	Binary Multiplier Operand Booth <i>Radix-4</i> Recoding Scheme	152
5.6	Area and Delay of Various Parallel DFXP Multipliers [7]	157
5.7	Area and Delay of Parallel Multipliers (DFXP vs. DFP)	166
5.8	Area and Delay vs. Pipeline Depth of Parallel DFP Multiplier	166

5.9	Area and Delay of Multipliers (Iterative vs. Parallel, DFXP vs. DFP)	168
B.1	Notation and Nomenclature of DFP Entity Components and Fields	210
B.2	Notation of Operands and Data	211
B.3	Unary Arithmetic Operations and Symbols	212
B.4	Binary Arithmetic Operations, Symbols, and Operand Names	212
B.5	Logic Operations and Symbols	213
B.6	Truth Tables of Logic Operations	213
B.7	Operator Precedence	214

List of Figures

2.1	Fixed-point Example	18
2.2	Scaled Fixed-point Example	19
2.3	Floating-point Example	20
3.1	Generalized Flow of DFXP Addition	48
3.2	Successive Correction Example	50
3.3	Bias and Correction Example	51
3.4	Carry-save Addition Example	56
3.5	Signed-digit Addition Example	58
3.6	Generalized Flow of DFXP Multiplication	63
3.7	Generalized Design of DFXP Digit-by-digit Multiplication	65
3.8	Generalized Design of DFXP Word-by-digit Multiplication	68
3.9	Generalized Design of DFXP Word-by-word Multiplication	76
4.1	Preliminary Iterative DFXP Multiplier Design	82
4.2	Flowchart of Iterative DFXP Multiplier Using Decimal CSAs	85
4.3	Iterative DFXP Multiplier Design Using Decimal CSAs	91
4.4	Flowchart of Iterative DFXP Multiplier Using Signed-Digit Adders	95
4.5	Example of Recoding into Signed Decimal Digits	95
4.6	Example for Iterative DFXP Multiplier Using Signed-Digit Adders	97
4.7	Recoder Block: (a) Single Digit, (b) n-Digit Operand	99
4.8	Digit Multiplier Block: (a) Single Digit, (b) n-Digit	102
4.9	Iterative DFXP Multiplier Using Signed-Digits Adders	109
4.10	Flowchart of Iterative DFP Multiplier Using Decimal CSAs	114
4.11	Top Portion of Iterative DFP Multiplier Design	116
4.12	Rounding Scheme	128
4.13	Bottom Portion of Iterative DFP Multiplier Design	132
5.1	Flowchart of Parallel DFXP Multiplier Using Decimal CSAs [5]	137
5.2	Partial Product Reduction Tree Employing Decimal CSAs [5]	140
5.3	Flowchart of Parallel DFXP Multiplier Using Binary CSAs [6]	143
5.4	Partial Product Reduction Tree: <i>Radix-10</i> Recoding, Binary CSAs [6]	147
5.5	Flowchart of Parallel BFXP/DFXP Multiplier Using Binary CSAs [7]	151
5.6	Binary/Decimal Multiplier Operand Recoding Example [7]	152

5.7	Combined Bin/Dec Partial Product Reduction Tree (33 Products) [7]	154
5.8	Split Bin/Dec Partial Product Reduction Tree (33 Products) [7] . . .	156
5.9	Flowchart of Parallel DFP Multiplier Using Binary CSAs [8]	159
5.10	Parallel DFP Multiplier Design [8]	162
B.1	DFP Storage Fields	210

List of Acronyms

ASIC	-	Application-specific Integrated Circuit
BCD	-	Binary Coded Decimal
BID	-	Binary Integer Decimal
BFXP	-	Binary Fixed-Point
BFP	-	Binary Floating-Point
CLB	-	Combinatorial Logic Block
CMOS	-	Complimentary Metal-Oxide Semiconductor
CSA	-	Carry-Save Adder
DFXP	-	Decimal Fixed-Point
DFP	-	Decimal Floating-Point
DPD	-	Densely Packed Decimal
HDL	-	Hardware Description Language
IC	-	Integrated Circuit
LSB	-	Least Significant Bit
LSD	-	Least Significant Digit
LUT	-	Look-Up Table
MSB	-	Most Significant Bit
MSD	-	Most Significant Digit
NaN	-	Not-a-Number
NFET	-	Negative-channel Field-Effect Transistor
FET	-	Field-Effect Transistor
PFET	-	Positive-channel Field-Effect Transistor
QNaN	-	Quiet Not-a-Number
SNaN	-	Signaling Not-a-Number
XS3	-	Excess-3 binary coded decimal

Abstract

Although a preponderance of business data is in decimal form, virtually all floating-point arithmetic units on today's general-purpose microprocessors are based on the binary number system. Higher performance, less circuitry, and better overall error characteristics are the main reasons why binary floating-point hardware (BFP) is chosen over decimal floating-point (DFP) hardware. However, the binary number system cannot precisely represent many common decimal values. Further, although BFP arithmetic is well-suited for the scientific community, it is quite different from manual calculation norms and does not meet many legal requirements.

Due to the shortcomings of BFP arithmetic, many applications involving fractional decimal data are forced to perform their arithmetic either entirely in software or with a combination of software and decimal fixed-point hardware. Providing DFP hardware has the potential to dramatically improve the performance of such applications. Only recently has a large microprocessor manufacturer begun providing systems with DFP hardware. With available die area continually increasing, dedicated DFP hardware implementations are likely to be offered by other microprocessor manufacturers.

This dissertation discusses the motivation for decimal computer arithmetic, a brief history of this arithmetic, and relevant software and processor support for a variety of decimal arithmetic functions. As the context of the research is the IEEE Standard for Floating-point Arithmetic (IEEE 754-2008) and two-state transistor technology,

descriptions of the standard and various decimal digit encodings are described.

The research presented investigates algorithms and hardware support for decimal multiplication, with particular emphasis on DFP multiplication. Both iterative and parallel implementations are presented and discussed. Novel ideas are advanced such as the use of decimal counters and compressors and the support of IEEE 754-2008 floating-point, including early estimation of the shift amount, in-line exception handling, on-the-fly sticky bit generation, and efficient decimal rounding. The iterative and parallel, decimal multiplier designs are compared and contrasted in terms of their latency, throughput, area, delay, and usage.

The culmination of this research is the design and comparison of an iterative DFP multiplier with a parallel DFP multiplier. The iterative DFP multiplier is significantly smaller and may achieve a higher practical frequency of operation than the parallel DFP multiplier. Thus, in situations where the area available for DFP is an important design constraint, the iterative DFP multiplier may be an attractive implementation. However, the parallel DFP multiplier has less latency for a single multiply operation and is able to produce a new result every cycle. As for power considerations, the fewer overall devices in the iterative multiplier, and more importantly the fewer storage elements, should result in less leakage. This benefit is mitigated by its higher latency and lower throughput.

The proposed implementations are suitable for general-purpose, server, and main-frame microprocessor designs. Depending on the demand for DFP in human-centric applications, this research may be employed in the application-specific integrated circuits (ASICs) market.

Chapter 1

Introduction

To set the stage for this dissertation, the motivation for research in decimal computer arithmetic is discussed. This takes the form in this chapter as a series of observations regarding the relevance and timeliness of decimal computer arithmetic and an account of the inabilities of binary computer arithmetic and the insufficiencies of decimal software solutions. With an argument for hardware support of decimal computer arithmetic having been made, an overview of the research to be presented in this thesis is described. The significance of this research is then advanced, followed by an outline of the remaining portions of the dissertation.

1.1 Motivation for Decimal Computer Arithmetic

The relevance and timeliness of research in the area of decimal computer arithmetic, particularly DFP, is underscored by three significant observations. First, there are a number of established computer languages now supporting DFP arithmetic, including C/C++ [9], COBOL [10], Eiffel [11], Java [12], Lua [13], PERL [14], Python [15], REXX [16], and Ruby [17] (see Section 2.2). Second, the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985) [18] and the IEEE Standard for Radix-Independent Floating-Point Arithmetic (IEEE 854-1987) [19] have been merged, expanded, submitted, and approved as a new standard named the IEEE Standard for Floating-Point Arithmetic [20]. This standard, hereafter referred to as IEEE 754-2008, includes a comprehensive definition of DFP arithmetic (see Section 2.4). And third, processor die area¹ continues to become more affordable [21], which allows new features to be added such as the introduction of decimal arithmetic. Both IBM and Intel have recognized and responded to the renewed market interest in DFP arithmetic. This is evidenced by Intel’s software support via their Decimal Floating-Point Math Library [22] and IBM’s hardware support via their Power6 [23], System z9 [24], and System z10 [25] microprocessors, all of which conform to IEEE 754-2008 (see Sections 2.2 and 2.3, respectively).

The primary motivation for decimal computer arithmetic is to enable users of computing systems to achieve results from decimal operations that are the same as if performed by hand, albeit tremendously faster. When performing a decimal operation on a system that does not support the storage of decimal numbers, adequate decimal datatypes, or decimal operations, the result is subject to representation error, conversion error, and rounding (or roundoff) error. Further, using a system designed

¹The higher cost in computation devices to implement decimal arithmetic in Boolean logic was a compelling reason to adopt binary arithmetic in early computing systems, see Section 2.1.

for binary arithmetic to perform decimal arithmetic often leads to errors that are difficult to diagnose.

A storage format specifically for decimal numbers is necessary as there is significant decimal input data whose value is exact and whose conversion to a binary representation would be inexact. Consider the value of a penny whose representation is an infinitely repeating binary fraction. This type of error is called representation error. Although a value such as $\frac{1}{3}$ cannot be represented exactly as a decimal number, this error is expected and accepted when the objective is to achieve the same results from the computing system as if performed by hand.

Decimal datatypes are needed not only to represent decimal values accurately, but also to enable the programmer to use a single variable for each datum and intended operation. If an exact decimal value is stored in an adequate decimal storage format, but is then read for use in a program and placed in an overloaded binary or decimal integer datatype, there is the potential for conversion error. Even if values destined for these datatypes are to be scaled to overcome this error, the use of an additional variable for the scaling is more error-prone and presumably less efficient than using a dedicated decimal datatype. Further, this approach limits the range of values. Thus, to eliminate the conversion error and reduce the programming error, a DFP datatype is needed (though a DFXP datatype is suitable in certain situations).

Without hardware support of decimal operations, rounding would need to occur in software in order to avoid potential rounding error. This is because existing binary operations do not recognize the concept of decimal digits, and therefore, may improperly round the data. Further, it is generally preferable to round a decimal number to a specified number of digits, which is not a supported operation in IEEE 754-compliant BFP systems. Having hardware support of decimal operations also has the benefit of significantly faster performance over software support [26–28].

To gain an appreciation of the potential for error, consider Table 1.1 from Cowlshaw [1] which shows the results of repeatedly dividing nine by ten. Further, consider the following true stories involving rounding error showing how these errors can lead to a loss of money or a loss of life. In 1981, the Vancouver Stock Exchange decided to create an index of its equities with an initial value of 1000.000. However, due to rounding error, over a period of 22 months the index, updated thousands of times each day for each change in a covered equity valuation, dropped in value to 574.081 when it should have increased to 1098.892 [29]. For those who bought and sold shares of this index during this period, a substantial percentage of their investment was lost. And in 1991, a Patriot Missile failed to intercept a SCUD Missile due to rounding error related to its internal timer used for velocity and range calculations [30]. The incorrect time value led the Patriot missile to search for the Scud missile in an incorrect location, ultimately leading to the death of 28 servicemen.

Table 1.1: Successive Division of Nine by Ten [1]

Decimal†	Binary‡
0.9	0.9
0.09	0.089999996
0.009	0.0090
0.0009	9.0E-4
0.00009	9.0E-5
0.000009	9.0E-6
9E-7	9.0000003E-7
9E-8	9.0E-8
9E-9	9.0E-9
9E-10	8.9999996E-10

† Results using Java BigDecimal class

‡ Results using Java float datatype

In research presented by Tsang and Olschanowsky [31], a study of the datatypes of over 1,000,000 columns in various commercial databases indicated 55% of the numeric data were decimal numbers, and an additional 43% could be represented with a decimal number. Therefore, with a preponderance of at least business data representable with decimal numbers, a system supporting decimal arithmetic, complete with decimal storage formats, decimal datatypes, and fundamental decimal operations, is desirable.

Toward this end, a variety of software packages that support decimal arithmetic have been developed. Two such packages are the Intel Decimal Floating-Point Math Library [32] and the IBM decNumber Library [33]. These and other software packages are described in Section 2.2. Further, the continued hardware support of DFXP arithmetic [34] and the emergence of hardware support of DFP arithmetic [23], underscore the growing demand for decimal computation solutions.

As for the need for decimal multiplication in hardware, Wang *et al.* [28] expanded upon the work in [27] and examined² five financial benchmarks compiled with the decNumber library and learned the percentage of execution time spent on decimal multiplication was 1.5%, 12.5%, 13.4%, 23.1%, and 27.5%³. In addition, the decimal divide operation, which consumed as much as 50% of the execution time in these same benchmarks, can be efficiently implemented in hardware using algorithms that rely heavily on multiplication [35]. Overall, the percentage of execution time spent on decimal operations ranged from 33.9% to 93.1% for the five benchmarks.

It should be noted that not every microprocessor developer agrees on hardware support for decimal arithmetic, similar to when hardware support for multimedia extensions were introduced [36]. The potential speedup and percentage of execution

²Wang *et al.* developed four of the five benchmarks.

³Platform: Intel Pentium 4 Processor, decNumber version: 3.32

times presented in [26–28] are in stark contrast with Intel’s research that indicates most commercial applications spend 1% to 5% of their execution time performing decimal operations [37]. And in research on commercial Java workloads [38], two Java benchmarks and one financial application written in Java exhibited 2.6%, 0.7%, and 0%, respectively of their execution time performing decimal arithmetic. Therefore, in the opinion of Intel researchers, hardware solutions of decimal arithmetic are not necessary at this time.

1.2 Overview of Research

The research presented in this dissertation is on the computer hardware multiplication of decimal numbers. The ideas and algorithms are based on the digits of the decimal input data being in a binary coded decimal (BCD) form, though many concepts are applicable to other decimal digit encodings. Decimal multiplier implementations are described for generic fixed-point environments, for which there is no standardization, and floating-point environments, as defined in the IEEE Standard for Floating-point Arithmetic [20] (IEEE 754-2008).

Prior to the description of these multiplier designs, a survey of decimal arithmetic is presented. This includes a brief history of decimal computing, an overview of current software and processor support of decimal arithmetic, and highlights of the decimal portion of IEEE 754-2008. Then, related research is presented that includes descriptions of both decimal digit and decimal significand encodings and an overview of both decimal addition and decimal multiplication techniques.

The multiplier research follows, which includes the details of two iterative DFXP implementations [39,40] along with one of these, [39], extended to support DFP multiplication [41]. Next, two parallel DFXP designs are described [5,6], along with one of these, the Vazquez *et al.* multiplier [6], extended to support DFP multiplication. Additionally, research accepted for publication [7] is presented on improvements to a parallel, combined BFXP/DFXP multiplier presented in [6]. The parallel designs of [5,6] are included for completeness and because Hickmann, Schulte, and I extended the Vazquez *et al.* fixed-point design to support DFP [8]. Latency, throughput, area, and delay information are presented for the multiplier designs of [5,6,8,39,41]. Further, comparisons are made amongst the iterative fixed-point designs, amongst the parallel fixed-point designs, and between the iterative and parallel implementations.

Descriptions and comparisons of the iterative and parallel DFP multiplier designs have been accepted for publication [42] and are included as part of this dissertation.

1.3 Significance of Research

My research presented in this dissertation centers on six publications in which I was the principal researcher and author [39–42] or in which I performed a significant role [7, 8]. This research, and hence this document, is divided into iterative decimal multiplication (both fixed- and floating-point) and parallel decimal multiplication (both fixed- and floating-point).

I developed two unique iterative DFXP multiplier algorithms and designs. The first design [39] contained the following novel features: decimal (3:2) counters and decimal (4:2) compressors, fast generation of multiplicand multiples that do not need to be stored, and a simplified decimal carry-propagate addition to produce the final product. Then, in the design of [40], I employed a recoding scheme and signed-digits in a new way to eliminate the pre-computation of multiples and accumulate the partial products in an efficient manner. As IEEE 754-2008 was converging at this time, I extended the fixed-point multiplier of [39] to support DFP. In so doing, I published the first DFP multiplier compliant with what is now IEEE 754-2008 [41]. This design was novel in other aspects as well, namely a mechanism to support on-the-fly generation of the sticky bit, early estimation of the shift amount, and efficient decimal rounding. Additionally, notable implementation choices include leveraging the leading zero counts of the operands' significands, passing NaNs through the dataflow with minimal overhead, and handling gradual underflow via minor modification to the control logic.

My work in the area of parallel DFXP multiplication followed articles by Lang *et al.* [5] and by Vazquez *et al.* [6]. Working with Hickmann and Schulte, we extended the fixed-point multiplier of [6] to support DFP in a manner similar to the approach described in [41]. Our publication [8] was novel in that it presented the first parallel

DFP multiplier compliant with what is now IEEE 754-2008 and implemented unique exception pass-through mechanisms to improve overall performance. A thorough introduction to DFP multiplication, descriptions of the iterative DFP multiplier design of [41] and the parallel DFP multiplier design of [8], and a comparison and analysis of these designs has been accepted for publication [42]. My latest research, also a collaboration, is regarding improvements to the parallel, combined BFXP/DFXP multiplier design described in [6]. This work, which includes several novel mechanisms to reduce the delay significantly, particularly of the binary multiplication path, was recently accepted for publication [7].

The research presented in this dissertation on decimal hardware multiplication has been used by a number of other researchers to improve or advance both decimal multiplication and related operations. In [35] and [43], Wang *et al.* extended my iterative DFXP multiplier design [39] to realize decimal division and square-root operations, respectively. In [5], Lang *et al.* utilized the operand recoding scheme described in [40], and they used decimal CSAs in a manner similar to that described in [39]. In [6], Vazquez *et al.* also employed the recoding scheme described in [40]. Additionally, my research provides ideas and solutions suitable for implementation and a foundation for further DFP multiplication research, such as decimal fused multiply-add. Finally, the research presented on DFP rounding for multiplication may be beneficial to those investigating other DFP operations.

1.4 Outline of Dissertation

The outline of this dissertation is as follows. First, an abbreviated history of decimal computer arithmetic is introduced in Chapter 2. This chapter also describes the support of decimal arithmetic currently available in numerous software and hardware offerings. As these offerings, and my research, are greatly influenced by IEEE 754-2008, details of this new standard are also included in this chapter. Next, an overview of related work is presented in Chapter 3. In particular, a number of decimal digit encodings, both non-redundant and redundant, are detailed. Additionally, the concepts, terminology, and some closely related research regarding decimal addition and decimal multiplication are described.

Following these prefatory chapters, the crux of my research is presented. Chapter 4 contains descriptions of two iterative DFXP multipliers [39, 40] and details on an iterative DFP multiplier [41], based on the design presented in [39]. Then, two parallel DFXP multipliers are presented in Chapter 5 along with details on a parallel DFP multiplier [8]. Note the fixed-point designs in Section 5.1 are attributable to Lang *et al.* and Vazquez *et al.*, but are included here for completeness, and because I participated in the research to extend a design of Vazquez *et al.* to support DFP. Section 5.1 also includes recent research on improvements to a parallel, combined BFXP/DFXP multiplier design [7]. In both Chapters 4 and 5, analysis and observations are made between the designs, and in Chapter 5, comparisons are presented between the iterative and parallel designs. Chapter 6 contains a summary of my research and some proposals for future research.

Supportive information is included in the dissertation as appendices. Appendix A contains definitions of mathematical terms, and Appendix B contains the mathematical and logical notation used throughout this dissertation. Finally, a short account

of the author's educational and professional career appears in Appendix C.

Chapter 2

Background of Decimal Computer Arithmetic

As this dissertation delves into hardware algorithms and implementations for decimal multiplication, it is appropriate to properly describe the context of this research. This chapter provides a brief history of binary numbers, decimal numbers, how these numbers have been represented in computers, and notable computer systems from the early electrical computing era. Additionally, current software and processor support for decimal arithmetic is presented, and the standard for DFP arithmetic is described.

2.1 History of Decimal Computer Arithmetic

In this section, a brief history is presented of decimal numbers, binary numbers, and the manner in which they have traditionally been represented in computer systems. Specifically, numeric values are represented as fixed-point numbers, scaled fixed-point numbers, or floating-point numbers. These three representations are described. Lastly, notable computer systems from the early electrical computing era are listed along with some salient attributes.

2.1.1 Decimal Numbers

The positional base ten (decimal) numeral system originated in India around 500 A.D. [44], presumably from the fact that humans have ten digits on their hands and feet. The Indian mathematicians developed the ten unique characters, the concept of zero as a number, and the idea that each character has a positional value and an absolute value within that position. The positional value of a character is ten times the positional value of the character to its right. The Arabs adopted this numeral system in the ninth century and propagated it in Europe. Because it was the Arabs who introduced the Europeans to the ten numerals, the Europeans refer the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ as *Arabic numerals*. However, the Arabs refer to this set as *Indian numerals*. In recognition of both of these cultures, these numerals are often referred to as *Hindu-Arabic numerals*.

2.1.2 Binary Numbers

The positional base two (binary) numeral system is attributed to an Indian author in the fourth century who used short and long syllables in a mathematical manner to represent the patterns in poetry and song. But it was not until the early 1600s

that Francis Bacon laid the groundwork for the general theory of binary encoding [45]. Then, in an 1854 publication [46], Boole describes the principles of what is now called *Boolean algebra*. Later, in 1937, Shannon [47] proved how Boolean algebra and binary arithmetic can be used to reduce an implementation of relays (gates) and further, how the electrical properties of relays (gates) can be used to implement Boolean algebraic problems.

2.1.3 Fixed-point Numbers

A fixed-point number is similar to an integer except the location of the radix point may be other than immediately to the right of the LSD (the digit of order $base^0$). Further, the number of digits to the right of the radix point remains fixed. A fixed-point number system is one in which all the operands and results have the radix point in the same position. The number of digits used to represent both the whole and fractional portions of the number defines the precision, p . Fixed-point numbers, for a particular base are comprised of an integer, I , and a fraction, F . Thus, a fixed-point number, FX , stored as a single variable, may be expressed in the following form.

$$FX = I.F \tag{2.1}$$

Fixed-point numbers offer the maximum amount of precision for a specified memory space as all the bits are available for use as significant digits. However, the range of a fixed-point number is limited, and therefore, a loss of precision is more likely as compared to a scaled fixed-point number or a floating-point number. Additionally, there are no standardized means to represent unique values such as infinity or erroneous data. Consider the example of Figure 2.1 which shows a loss of accuracy. The next subsection shows how *scaled fixed-point numbers* can reduce the occurrence of

A	$=$	024.69	fixed-point multiplicand
$\times B$	$=$	000.05	fixed-point multiplier
AB	$=$	001.2345	fixed-point product with no loss of accuracy
<hr/>			
A	$=$	024.69	fixed-point multiplicand
$\times B$	$=$	000.05	fixed-point multiplier
AB	$=$	001.23	fixed-point product with loss of accuracy (0.0045)

Figure 2.1: Fixed-point Example

loss of precision.

2.1.4 Scaled Fixed-point Numbers

To extend the range of fixed-point numbers, a system can provide a scaling variable for each fixed-point number. The scaling variable is a radix multiplier used to indicate the amount by which the fixed-point number is multiplied by the radix. The benefit of this approach is that a loss of precision can often be avoided, as the “window” of precision is adjusted by the scaling variable. Thus, for a base, b , a scaled fixed-point number, (FX, SV) , is stored as two variables and may be expressed in the following form. Typically, the fixed-point portion of a scaled fixed-point number is an integer.

$$(FX, SV) = I.F \cdot b^{SV} \quad (2.2)$$

The example in Figure 2.2 illustrates how a loss of precision may be avoided with a scaling variable.

The drawbacks of the scaled fixed-point number are 1) the additional and separate storage area required to contain the scale value, 2) the potential for programming error

$(A, A_{SV}) = (02469, -2)$	scaled fixed-point multiplicand
$\times (B, B_{SV}) = (00005, -2)$	scaled fixed-point multiplier
$(AB, AB_{SV}) = (12345, -4)$	scaled fixed-point product

Figure 2.2: Scaled Fixed-point Example

due to each number requiring two entities for its representation, and 3) there are no standardized means to represent unique values such as infinity or erroneous data. The next subsection shows how *floating-point numbers* enable a single entity to contain a number with a sliding window of precision.

2.1.5 Floating-point Numbers

To provide a number format comprised of a single entity, as are fixed-point numbers, yet with extended range, similar to scaled fixed-point numbers, mathematicians employ the floating-point number. Floating-point numbers, for a particular base, b , are comprised of a sign, s , an exponent, E , and a significand, C . A floating-point number, FL , may be expressed in the following form:

$$FL = -1^s \cdot C \cdot b^E, \tag{2.3}$$

where the significand has a precision of p digits. The exponent is generally an integer, and the significand can be an integer or a fixed-point number with an integer and fraction. Note, the sign bit is not a requirement, as the significand could be a radix-complement number, for example. A floating-point number system may produce a result whose radix point is in a different location than one or both its operands. As an example, consider the following equation.

$$\begin{array}{rcl}
A & = & -1^0 \cdot 2469 \cdot 10^{-2} \text{ floating-point multiplicand} \\
\times B & = & \frac{-1^0 \cdot 5 \cdot 10^{-2} \text{ floating-point multiplier}}{} \\
AB & = & -1^0 \cdot 12345 \cdot 10^{-4} \text{ floating-point product}
\end{array}$$

Figure 2.3: Floating-point Example

By allowing the location of the radix point to float, a computing system can provide results of maximum accuracy using a fixed number of digits.

2.1.6 Early Computer Arithmetic Systems

Three of the earliest electronic computers, the ENIAC [48], UNIVAC [49], and IBM 650 [50] performed their arithmetic functions in base ten [51]. In the same era, the EDSAC [52], EDVAC [53], and the ORDVAC [54] and its equivalent, Illiac, performed their arithmetic functions in base two. Even with the advent of solid state computers based on the two-state transistor, some computer manufacturers continued to process data in base ten by simply encoding each decimal digit in four binary bits (e.g., binary coded decimal (BCD) with weights of 8, 4, 2, 1 or biquinary coded decimal with weights of 5, 4, 2, 1) [55]. However, as system solutions for business and scientific applications were generalized and the demand for faster scientific calculations outpaced that of business calculations, binary arithmetic emerged as the de facto standard.

Unfortunately, the computer platforms that adopted binary arithmetic implemented their floating-point support without any noticeable collaboration. This resulted in proprietary data formats, unique instructions, and different results for the same operation [24]. Since the advancement of any technical, business, or social

discipline requires reproducible and consistent results from experiments and computations, a genuine standard was necessary. In 1985, the IEEE Standard for Binary Floating-Point Arithmetic was adopted [18]. Today, all major microprocessor platforms supporting floating-point arithmetic adhere to this unifying standard. More information regarding the IEEE 754-2008 Standard appears in Section 2.4.

Table 2.1 provides a time line of select early computer systems and some notable events in the history of modern computing. Of the systems below which supported decimal arithmetic (IBM ASCC, ENIAC, UNIVAC, IBM 650, NEAC 2201, IBM 7030, and IBM System/360) some supported floating-point operations (UNIVAC, though not on the earliest machines, and IBM 650). In the next subsection, software support of decimal arithmetic in modern programming languages is described. Following this, processor support of decimal arithmetic is presented.

Table 2.1: Time Line of Early Computer Systems and Notable Events [2]

Year	System or Event	Base	Comments
1913	analytical engine [56]	decimal	Torres, electro-mechanical
1938	Z1 [57]	binary	Zuse, electro-mechanical
1939	ABC [58]	binary	Atanasoff & Berry, electronic-digital
1943	Colossus [59]	binary	Flowers, Turing, <i>et al.</i> , code deciphering
1944	IBM ASCC/MARK I [60]	decimal	Aiken, electro-mechanical
1945	ENIAC [48]	decimal	Eckert & Mauchly
1947	first transistor [61]		Bardeen, Brattain, & Shockley
1949	EDSAC [52]	binary	Wilkes, first stored program machine
1951	UNIVAC I [49]	decimal	Eckert & Mauchly
	first junction transistor [62]		Shockley
1952	EDVAC [53]	binary	Eckert & Mauchly
1953	IBM 650 [50]	decimal	first mass-produced computer
1952	ORDVAC [54] & Illiac I	binary	first von Neumann architecture computer commercially available
1954	first silicon transistor [63]		Texas Instruments
1956	UNIVAC [49]	decimal	some transistor-based components
1958	first semiconductor IC [64]		Kilby & Noyce (independently)
1959	NEC NEAC 2201 [65]	decimal	Japan's first commercial transistor computer
1960	UNIVAC LARC [66]	decimal	Remington Rand mainframe
1961	IBM 7030 (Stretch) [67]	binary	supported decimal
1964	IBM System/360 [68]	binary	Amdahl, supported decimal, first family of computers
1964	CDC 6600 [69]	binary	Cray, first commercially successful supercomputer

2.2 Software Support of Decimal Arithmetic

As the desire for precise decimal computer arithmetic exists, a variety of software with decimal arithmetic support has been, and continues to be, developed to satisfy this desire. Some computing systems offer hardware support of decimal computer arithmetic, either as hardware instructions or hardware-assist instructions. When available, software solutions can take advantage of the speedup achievable in hardware. However, the software may not utilize the available hardware depending on the features of the software (e.g. if it is a purely interpreted language) or the intentions of the programmer (e.g., if portability and reproducibility across platforms is sought). In this section, a listing of current languages offering support of decimal arithmetic is provided.

Some prominent programming languages with primitive decimal datatypes include Ada [70], .NET Framework (C# [71], Visual Basic [72]), COBOL [10], PL/I [73], and SQL [74]. Beginning in 1995, Ada offered a decimal datatype as a scaled fixed-point number with a maximum precision of 18 digits. The .NET Framework's *System.decimal* class provides users of C# and Visual Basic (VB.NET) with a 29-digit DFP primitive. COBOL has a 32-digit DFP datatype, and is working to incorporate a decimal datatype based on the *decimal128* format defined in IEEE 754-2008 (described in Section 2.4). PL/I has had a *DECIMAL FLOAT* datatype since its inception in 1964. The *DECIMAL FLOAT* datatype is a scaled fixed-point number with up to 15 digits of precision. SQL has long since offered the *numeric* and *decimal* datatypes as scaled fixed-point numbers (the maximum precision varies between databases). Though not a programming language, XML [75], designed to store and transport data, contains a *precisionDecimal* datatype modeled on IEEE 754-2008. This is significant as XML-aware applications must be written in languages support-

ing this decimal datatype.

In addition to the aforementioned languages with primitive decimal datatypes, there are several notable languages which adhere to the General Decimal Arithmetic Specification [76] (GDAS) developed by Mike Cowlishaw. This specification, based on IEEE 854-1987 [19], forms the basis of decimal arithmetic in IEEE 754-2008 [20]. The GDAS is also standardized in ANSI X3.274-1996 (Programming Language Rexx [16]). Languages supporting the GDAS are able to operate within the confines of IEEE 754-2008. The following languages offer arithmetic packages conforming to the GDAS: Eiffel Decimal Arithmetic [11], IBM C DecNumber [33], Java BigDecimal [12], Lua decNumber [13], Python Decimal [15], and Rexx [16]. Partial conformance to the GDAS can be found in PERL BigNum [14] and Ruby BigDecimal [17].

Table 2.2 contains a list of programming languages, scripting languages, and compilers which support decimal arithmetic in some manner. Languages containing decimal datatypes and languages conforming to the GDAS are listed in the first two sections of the table, respectively. The next section of the table lists the proposed DFP extensions to the C and C++ language standards, the first C/C++ compiler designed to support these proposed extensions, and a C math library which supports IEEE 754-2008. The proposed DFP extensions conform to IEEE 754-2008. The C math library [32], developed by Intel, implements the DFP arithmetic defined in IEEE 754-2008. Its algorithms are designed specifically for operands stored in the BID format (see Section 2.4.2). The underlying algorithms used in the Intel DFP C Math library are described in [22, 37, 77].

The last section of Table 2.2 lists the compilers and programs which conform to IEEE 754-2008 and are able to utilize the DFP hardware instructions available on the IBM Power6, IBM System z9, and IBM System z10 platforms. These compilers can be passed an architecture parameter to either map into C decNumber library [33] calls

Table 2.2: Software Support of Decimal Arithmetic

Primitive Datatypes	
Ada [70]	15-digit scaled fixed-point number
.NET (C# [71], VB.NET [72])	29-digit DFP number
COBOL [10]	32-digit DFP number
PL/I [73]	15-digit scaled fixed-point number
SQL [74]	scaled fixed-point number (precision may vary)
XML [75]	IEEE 754-2008
Conformance with the GDAS (able to conform to IEEE 754-2008)	
Eiffel [11]	
IBM C decNumber [33]	based on DPD encoding (see Section 2.4.2)
Java BigDecimal [12]	
Lua decNumber [13]	
Python Decimal Class [15]	
Rexx [16]	
PERL BigInt [14]	partial conformance
Ruby BigDecimal [17]	partial conformance
Conformance with IEEE 754-2008	
C [78]	proposed extension
C++ [79]	proposed extension
GCC [80]	version 4.2 supports C [78] and C++ [79] proposals
Intel C DFP Math [32]	based on BID encoding (see Section 2.4.2)
Conformance with IEEE 754-2008 / Utilization of Hardware DFP Instructions	
GCC [9]	version 4.3
IBM XL C/C++ [81]	version 9
IBM Enterprise PL/I [82]	release 3.7
IBM DB2 [83]	version 9.1
IBM High-level Assembler [84]	release 6
IBM DFPAL [85]	
SAP NetWeaver [86]	version 7.10

or DFP hardware instruction mnemonics. The DFP Abstraction Layer (DFPAL [85]) enables users of applications utilizing DFPAL to compile once and run on a variety of IBM PowerPC systems. If the code is executing on an IBM Power6 system, then DFP hardware instructions will be used, otherwise the C decNumber library will be used. Processor support of DFP is described in Section 2.3.

2.3 Processor Support of Decimal Arithmetic

As mentioned in Section 2.1, some of the earliest programmable computer systems offered the capability of decimal arithmetic. As the two-state transistor became the basic building block of computing, computer systems were designed with limited or no capability of hardware decimal arithmetic. Instead, these systems offered binary arithmetic. This section lists some of the more recent computer systems providing hardware support of decimal arithmetic.

In [87], a three-stage, single accumulator, variable-precision, DFP unit prototype named Controlled-Precision Decimal Arithmetic Unit (CADAC) is described. It was designed using existing small- and medium-scale integrated circuits controlled by a microcontroller chip. This prototype supports some desirable high-level precision control constructs. Its operations are not pipelined; rather, each operation is performed iteratively, two decimal digits at a time, until the desired precision is obtained. In [88], the work in [87] is extended with a software system that supports variable-precision decimal arithmetic. Consideration is given to the hardware necessary to support the systems programmer, namely, exception handling, but no improvements to the actual hardware implementation are described. In [89], a prototype processor named Bit-slice Arithmetic Processor for Scientific Computing (BAP-SC), built on wire-wrap boards and controlled via a parallel interface from the host computer, is described that implements DFP add, subtract, multiply, and divide. The algorithms themselves are not particularly aggressive in terms of performance, but the hardware does provide significant acceleration over pure software solutions.

As for commercial microprocessors, several offer minimal fixed-point BCD arithmetic instructions. The Intel x86 processor series offers eight decimal instructions [90], i.e., DAA, DAS, AAA, AAS, AAM, AAD, FBLD, FBSTD. The Motorola 68k proces-

processor series provides five decimal instructions [91], i.e., ABCD, NBCD, PACK, SBCD, and UNPK. And the HP PA-RISC processor series offers two decimal instructions [92], i.e., DCOR and IDCOR. All three of these processors have instructions to correct the result of a binary add and binary subtract performed in a bias and correction manner on packed-BCD data. The bias and correction scheme is described in Section 3.2.1. Packed-BCD data is comprised of two BCD digits in each byte of the operand or register. Additionally, the Intel x86 processors has instructions to correct binary add, subtract, multiply, and divide on unpacked-BCD data.

More extensive support of decimal arithmetic in hardware can be found in IBM's mainframes, such as the S/390 [93] and System z900 [34]. Support for DFP arithmetic can be found in IBM's System z9 [24] and System z10 [25] mainframes, and in IBM's Power6 [23] server. The S/390 processor offers DFXP add, subtract, multiply, and divide via a dedicated decimal adder employing the direct decimal addition scheme (see Section 3.2.3). The multiply and divide instructions involve iterative additions and subtractions controlled by millicode. The System z900 processor offers the same core DFXP instructions but employs a combined binary/decimal adder.

The IBM System z9 is the first commercial platform to offer DFP arithmetic in conformance with IEEE 754-2008. Over 50 DFP instructions are supported through a combination of hardware instruction and millicode instructions. The fixed-point arithmetic unit on the z9 processor supports binary and decimal fixed-point operations. The unit uses the bias and correction method of add and subtract. Multiplication and division use temporary registers to speed up the iterative process. Further information regarding the multiplication instruction available on the IBM System z900, Power6, System z10 machines appears in Section 3.3.2.

Recently, SilMinds corporation (www.silminds.com) has made available the licensing of its synthesizable VHDL and Verilog code which performs DFP add, subtract,

Table 2.3: Contemporary Processor Support of Decimal Arithmetic

Limited Support	
Intel x86 family [90]	instructions to correct binary $+$, $-$, \times , \div
Motorola 68k family [91]	instructions to correct binary $+$, $-$
HP PA-RISC family [92]	instructions to correct binary $+$, $-$
early IBM mainframes, e.g., [34, 93]	firmware-assisted DFXP
Conformance with IEEE 754-2008	
IBM System z9 [24]	firmware-assisted DFP
IBM Power6 [23]	complete DFP hardware unit
IBM System z10 [25, 94]	extension of IBM Power6 DFP unit
SilMinds DFPA cores [95]	partial implementation ($+$, $-$, \times , $\times/+$, \div , $x^{\frac{1}{2}}$)

multiply [96], fused multiply-add, divide, and square root in conformance with IEEE 754-2008. SilMinds has indicated it is currently working on extending its code for use in FPGA-based accelerators. A presentation regarding their intellectual property is available [95]. The intent of SilMinds is for customers to use its off-the-shelf code when developing commercial processors or ASICs.

Table 2.3 contains a list of contemporary processor support for decimal arithmetic. The first portion of the table describes platforms offering limited support of DFXP arithmetic, while the second portion of the table lists platforms and available intellectual property with support for DFP arithmetic.

2.4 IEEE 754-2008 Standard

The existence of a standard for computer arithmetic benefits the hardware developer, the software developer, and the end user. When designing a standard-compliant design, hardware developers have confidence their solution will provide the same functionality as the designs of other developers. Further, for a given standard function, they can compare the performance of their offering against others' to determine their competitiveness. For software developers, a standard enables them to create code for compliant systems which is portable and yields consistent results. For end users, they are able to seamlessly migrate their data from one platform to the next as they are not locked into a proprietary storage format. Additionally, end users need not contend with idiosyncratic exception and rounding behavior among different systems.

Over the years, there have been a number of proposals regarding DFP operand formats and arithmetic. As early as 1913 [56], while developing an electro-mechanical design for an analytical engine, Torres proposed the concept of floating-point arithmetic. In 1969 [97], a representation for non-normalized operands was proposed in which the significand is an integer and the exponent represents a power of ten; both in binary form¹. Although the representation of the DFP number is reasonable, the binary format of the significand makes the shifting of an operand (needed for a variety of operations) time consuming as this would involve a multiplication or division by appropriate powers of ten instead of simply shifting left or right, respectively. In 1976 [98], a normalized format was proposed based on the encoding of three BCD digits into ten bits [99] with the exponent also stored in BCD format. This proposal was a precursor to the aforementioned BFP standard as it also includes rounding, exceptions, and instructions. In [100], another representation for the DFP number is

¹This representation is very similar to the Binary Integer Decimal (BID) storage format available in the new IEEE 754-2008 standard.

proposed that exhibits very good error characteristics. However, the mantissa is binary which, as mentioned earlier, leads to slow shifting. Recognizing the importance of extended word-lengths and DFP, IEEE 854-1987 [19] was adopted in 1987.

IEEE 854-1987 has seldom been implemented as the demand for decimal computer arithmetic has been for a relatively narrow set of business calculations, and this has been satisfied on commercial platforms. When solutions have been desired on general-purpose platforms, the user community has accepted differing decimal solutions utilizing a combination of software with binary fixed-point hardware, rudimentary BCD arithmetic or correction hardware, or decimal fixed-point hardware. Presumably, the end user has remained on the same platform due to their legacy investment. Pragmatically, IEEE 854-1987 lacks a description of the storage formats for DFP numbers. A second limitation of this standard is that it does not describe whether operands and results are to be normalized or not. The issue of normalization affects the arithmetic and the algorithms. These omissions may have led to its lukewarm reception. However, with the increase in processor speeds and the decrease in the cost of memory, two areas which advantaged binary arithmetic over decimal arithmetic, there has been interest in developing general-purpose solutions for decimal computer arithmetic. This interest, and the aforementioned absences from IEEE 854-1987, led to the inclusion of DFP arithmetic in a revised version of the IEEE Standard for Binary Floating-Point Arithmetic, IEEE 754-1985 [18].

IEEE 754-1985 defined four formats for representing BFP numbers (including negative zero and subnormal numbers) and special values (infinities and NaNs). Further, the standard defined a set of BFP operations for these BFP values, a set of four rounding modes, and a set of five exceptions. Upon its approval as a standard, IEEE 754-1985 had an authorized lifespan of five years. It has since been extended several times. In September of 2000, the IEEE Microprocessor Standards Committee spon-

sored the revision of IEEE 754-1985 to provide an updated standard for computer arithmetic. As part of the revision process for IEEE 754-1985 [18], the updated standard was to improve upon IEEE 754-1985, incorporate DFP (with consideration of IEEE 854-1987 [19]), and incorporate aspects of IEEE 1596.5 [101], the standard for shared-data formats.

After nearly eight years years of drafting, revising, and balloting, IEEE 754-2008 [20] was approved unanimously by the IEEE Standards Association Board in June, 2008. The standard was published in August, 2008. IEEE 754-2008 defines five floating-point basic number formats (three binary and two decimal) as well as binary and decimal interchange formats. Further, the standard defines a set of five rounding modes, a set of five exceptions (with default and alternate exception handling), the required support for program block attributes, and the requirements for expression evaluation. Also, a set of recommended attributes and a set of recommended correctly-rounded transcendental functions are presented. The most significant aspect of the updated floating-point standard, as related to the research presented in this dissertation, is the inclusion of DFP arithmetic and the external storage (interchange) formats of DFP numbers. In the remainder of this section, various aspects of IEEE 754-2008 are described.

2.4.1 Differences Between BFP and DFP

There are several differences between BFP and DFP, with respect to the numbers and the arithmetic. At a high level, for a given format length, DFP has a larger exponent range, while BFP has slightly greater precision. Additionally, BFP only defines the values, while DFP defines both the values and the representation. The representation of decimal data are particularly important as it facilitates the shar-

ing/migration of data. At a lower level, DFP results have a larger relative error than BFP results. Further, the relative error of DFP has a larger wobble than that of BFP [102,103]. Additionally, BFP operands and results contain normalized significands, while DFP operands and results do not. This has the following effects. First, when aligning significands such that digits of the same order are located in the same physical position for add-type operations, both operands may need to be shifted. Second, for multiply operations, if the number of significant digits in the unrounded product exceeds the format's precision, p , then this intermediate product may need to be left shifted prior to rounding. Third, if an intermediate product contains $p - i$ essential digits, then there exists i equivalent representations of the value. Note the i possible representations can only be realized if there is sufficient available exponent range to allow the leading non-zero digit to be placed in the MSD position and the trailing non-zero digit to be placed in the LSD position. For example, if p equals 5 and the operation is 32×10^{15} multiplied by 70×10^{15} , then possible results are 22400×10^{29} , 2240×10^{30} , or 224×10^{31} (leading zeros not shown).

Because of the possibility of multiple representations of the same value, IEEE 754-2008 introduces the concept of a preferred exponent. The preferred exponent, PE , is drawn from elementary arithmetic and based on the operation and the exponent(s) of the operand(s). Table 2.4 describes the preferred exponent for some operations. Note the preferred exponent, as described in the table, is for exact results. If the result is inexact, the least possible exponent is the preferred exponent as this yields the result with the greatest number of significant digits. In fact, with the exception of the `quantize` and `roundToIntegralExact`, all operations with inexact results are required to use the least possible exponent to yield the maximum number of significant digits. Table 2.4 shows the preferred exponent for DFP add, subtract, multiply, divide, and fused multiply-add. For multiplication, the preferred exponent, prior to any rounding

Table 2.4: Preferred Exponent of Select Decimal Operations

Operation	Example	Preferred Exponent
addition	$A + B$	$\text{minimum}(Q^A, Q^B)$
subtraction	$A - B$	$\text{minimum}(Q^A, Q^B)$
multiplication	$A \times B$	$Q^A + Q^B$
division	A/B	$Q^A - Q^B$
fused multiply-add	$(A \times C) + B$	$\text{minimum}(Q^A + Q^C, Q^B)$

or exceptions, is as follows. The *bias* needs to be subtracted avggs the result exponent should only be biased by the amount specific to its format (see Table 2.5).

$$\begin{aligned}
 PE &= Q^A + Q^B && \text{“unbiased value”} \\
 &= E^A + E^B - \textit{bias} && \text{“biased value in storage”}
 \end{aligned}$$

For example, the product of $A = 320 \times 10^{-2}$ multiplied by $B = 70 \times 10^{-2}$ is $P = 22400 \times 10^{-4}$. If an intermediate product with leading zeros and raised to the preferred exponent has essential digits to the right of the decimal point, the significand is left shifted while decrementing the intermediate exponent to yield a product with the maximum number of significant digits (so long as the exponent stays in range).

2.4.2 Decimal Formats

Three fixed-width interchange formats for DFP numbers are specified in IEEE 754-2008: *decimal32*, *decimal64*, and *decimal128* bits. For each format width, there is a one bit sign field (s), a combination field (G), and a trailing significand field (T). The number of bits in each format dedicated to s , G , and T are given in rows two through four of Table 2.5. Additionally, Table 2.5 shows the IEEE 754-2008

parameters for the proposed storage formats. A well-reasoned suggestion for DFP number parameters appears in a work by Johnstone *et al.* [100]. However, the choice of precision, exponent base and range, and significand representation and encoding ultimately approved for IEEE 754-2008, are based on the considerations and reasoning presented by Cowlishaw *et al.* [104] and the Decimal Subcommittee of the committee revising IEEE 754-1985.

Note in Table 2.5 that two sets of exponent ranges are presented. This is because the standard defines the parameters in terms of the significand, C , in a scientific form with range $0 \leq C < b$. An alternate view is of the significand as an integer with range $0 \leq C < b^p$, where p is the significand length, or precision, of the format. The exponent value in a DFP entity is a non-negative binary integer, which is the biased exponent value E shown in Table 2.5. Throughout this dissertation, either the exponent form Q ($E - bias$) or E ($Q + bias$) is used, depending on the context. The exponent form Q is the form of the exponent most often used when calculations are performed by hand, while the use of E allows the reader to appreciate the full complexity of the calculation. Table 2.6 shows the range in values for normal and subnormal numbers in each decimal format.

The combination field in each DFP datum is encoded to indicate if the representation is a finite number, an infinite number, or a non-number (i.e., Not-a-Number or NaN). It also contains the exponent and the MSD of the significand when the operand represents a finite number. The combination and trailing significand fields are jointly encoded to maximize the number of representable values or diagnostic information (i.e., NaN payload). The combination field is $w + 5$ bits wide and the biased exponent is $w + 2$ bits wide, where w is 6, 8, and 12, for the three interchange formats. Table 2.7 contains an interpretation of the combination field encoding. The trailing significand field is a multiple of ten bits and is either encoded via the Densely Packed Decimal

Table 2.5: DFP Format Parameters

Format name	decimal32	decimal64	decimal128
storage bits	32	64	128
sign	1	1	1
combination field	11	13	17
trailing significand	20	50	110
exponent bits	8	10	14
significand digits	7	16	34
exponent <i>bias</i>	101	398	6176
“Scientific form”: $0 \leq \text{significand value} < 10$			
exponent	$-95 \leq e \leq 96$	$-383 \leq e \leq 384$	$-6143 \leq e \leq 6144$
biased exponent	$6 \leq e + \textit{bias} \leq 197$	$15 \leq e + \textit{bias} \leq 782$	$33 \leq e + \textit{bias} \leq 12320$
“Integer form”: $0 \leq \text{significand value} < 10^p$			
exponent	$-101 \leq Q \leq 90$	$-398 \leq Q \leq 369$	$-6176 \leq Q \leq 6111$
biased exponent	$0 \leq E \leq 191$	$0 \leq E \leq 767$	$0 \leq E \leq 12287$

Table 2.6: DFP Format Ranges

Number	decimal32	decimal64	decimal128
“Scientific form”: $0 \leq \textit{Significand value} < 10$			
largest normal	$(10 - 10^{-6}) \times 10^{96}$	$(10 - 10^{-15}) \times 10^{384}$	$(10 - 10^{-33}) \times 10^{6144}$
smallest normal	1×10^{-95}	1×10^{-383}	1×10^{-6143}
largest subn'l	$(1 - 10^{-6}) \times 10^{-95}$	$(1 - 10^{-15}) \times 10^{-383}$	$(1 - 10^{-33}) \times 10^{-6143}$
smallest subn'l	$10^{-6} \times 10^{-95}$	$10^{-15} \times 10^{-383}$	$10^{-33} \times 10^{-6143}$
“Integer form”: $0 \leq \textit{Significand value} < 10^p$			
largest normal	$(10^7 - 1) \times 10^{90}$	$(10^{16} - 1) \times 10^{369}$	$(10^{34} - 1) \times 10^{6111}$
smallest normal	$10^6 \times 10^{-101}$	$10^{15} \times 10^{-398}$	$10^{33} \times 10^{-6176}$
largest subn'l	$(10^6 - 1) \times 10^{-101}$	$(10^{15} - 1) \times 10^{-398}$	$(10^{33} - 1) \times 10^{-6176}$
smallest subn'l	1×10^{-101}	1×10^{-398}	1×10^{-6176}

Table 2.7: Combination Field for DFP Representations

Special Classifications		
Combination ($g[0 : 4]$)	Combination ($g[5 : w + 4]$)	Meaning
11111	0 \langle <i>canonical declets</i> \rangle	quiet NaN
11111	1 \langle <i>canonical declets</i> \rangle	signalling NaN
11110	0...0	+ - infinity
Normal Classification Using Decimal Encoding (Finite Numbers)		
Combination ($g[0 : 4]$)	Exponent (E)	Significand (C)
0xxxx or 10xxx	$g[0] g[1] g[5 : w + 4]$	$(4 \cdot g[2] + 2 \cdot g[3] + g[4]) T^D$
110xx or 1110x	$g[2] g[3] g[5 : w + 4]$	$(8 + g[4]) T^D$
Normal Classification Using Binary Encoding (Finite Numbers)		
Combination ($g[0 : 4]$)	Exponent (E)	Significand (C)
0xxxx or 10xxx	$g[0 : w + 1]$	$g[w + 2 : w + 4] T$
110xx or 1110x	$g[2 : w + 3]$	$(8 + g[w + 4]) T$

(DPD) algorithm [3] or as an unsigned binary integer [20] called Binary Integer Decimal (BID). The multiplier designs presented in this dissertation assume the operands are stored in the decimal64 format with DPD encoding (described in Section 3.1). Software routines to convert between DFP values encoded using the BID format and the DPD format are available [32].

2.4.3 Rounding

A description of each rounding mode required by IEEE 754-2008 is listed in Table 2.8. The default rounding mode is language-defined, but is encouraged to be round to nearest, ties to even.

Table 2.8: Rounding Mode Descriptions

Rounding Mode	Choose representation closest to...
Round to nearest, ties to even	the infinitely precise result or with even LSD if two representations are equally close
Round to nearest, ties away from zero	the infinitely precise result or the larger of two equally close representations
Round toward $+\infty$	but no less than the infinitely precise result
Round toward $-\infty$	but no greater than the infinitely precise result
Round toward zero	and no greater in magnitude than the infinitely precise result

2.4.4 Exceptions

There are five exceptions that may be signaled during decimal operations: invalid operation, division by zero, overflow, underflow, and inexact. All but “division by zero” are possible with the multiply operation. With multiplication, the invalid operation exception is commonly signaled when either operand is a signaling NaN or when zero and infinity are multiplied. The default handling of the invalid operation exception involves signaling the exception and producing a quiet NaN for the result. If only one operand is a signaling NaN, then the quiet NaN result is created from the signaling NaN. Note that any produced or propagated NaN must have each ten-bit grouping in its trailing significand field as a defined pattern (i.e., only 1000/1024 possible combinations are defined; see IEEE 754-2008). If a NaN is in its canonical form, as shown in Table 2.7, it is guaranteed to be propagated. Non-canonical NaNs will be altered to realize a NaN in canonical form. If an operation is to yield a NaN and only one of its inputs is a NaN, that operation is to produce a NaN with the trailing significand of the input NaN. If an operation is to yield a NaN and two or more of

its inputs are NaN, that operation is to produce a NaN with the trailing significand of one of the input NaNs.

The division by zero exception is signaled when an operation on finite operands is to produce an exact infinite result. Under default exception handling, the result is infinity with the sign being the XOR of the operands' signs for divide or negative for $\log_B(0)$.

The overflow exception is signaled when a result's magnitude exceeds the largest finite number. The detection is accomplished after rounding by examining the computed result as though the exponent range is unlimited. Default overflow handling, as specified in IEEE 754-2008, involves the selection of either the largest normal number or canonical infinity and the raising of the inexact exception. When overflow occurs, the inexact exception, an indication essential digits have been lost, is raised because the most significant nonzero digit in the shifted intermediate product has been shifted off the most significant end of the register, effectively, in an effort to decrease the exponent into range. Numbers that are larger than the largest normal number (see Table 2.6) are sometimes called *supernormals*. A number is only supernormal if there is an insufficient number of leading zeros in the significand such that it cannot be sufficiently left shifted to decrease the exponent into range. As an example, the largest normal number in the *decimal32* format is $9999999 \times 10^{Q_{max}}$. If $7777000 \times 10^{Q_{max}}$ is multiplied by 90000×10^{-5} , the intermediate product is $0069993.0000000 \times 10^{Q_{max}+2}$. Since the intermediate product exponent is two greater than the maximum exponent, and there are two leading zeros in the intermediate product, the intermediate product can be left shifted two digit positions to decrease the exponent into range.

Under default exception handling, the underflow exception is signaled when a result is both tiny and inexact. Tininess is when the result's magnitude is between zero and the smallest normal number, exclusive. The detection of tininess is to occur prior

to rounding, when the intermediate result is viewed as if the exponent range and the precision are unlimited. As an example, the smallest normal number in the *decimal32* format is $1000000 \times 10^{Q_{min}}$. If $7777000 \times 10^{Q_{min}}$ is multiplied by 9000000×10^{-9} , the intermediate product is $6999300.0000000 \times 10^{Q_{min}-2}$. However, this exponent cannot be represented. Instead of abruptly converting this number to zero, a subnormal number is produced by shifting the significand to the right two digit positions and increasing the exponent by two to achieve the minimum exponent. Thus, the product significand is $0699930 \times 10^{Q_{min}}$. By reducing the precision in this manner, underflow occurs gradually. In the preceding example, the shifting to the right of the significand did not result in the loss of any nonzero digits. Thus, the results are exact, albeit subnormal, and the underflow exception is not raised. For an example of when the underflow exception is signaled, consider the following: $7777000 \times 10^{Q_{min}}$ multiplied by 9000000×10^{-13} . Here, the intermediate product is $6999300.0000000 \times 10^{Q_{min}-6}$. To achieve the minimum exponent, the significand must be right shifted and rounded² to produce $0000007 \times 10^{Q_{min}}$. Since one or more nonzero digits are “lost” to rounding, the result is both tiny and inexact, and the underflow exception is signaled. Default underflow handling is explained in the next subsection.

The overflow exception is accompanied by the inexact exception and the underflow exception is accompanied by the inexact exception³. When more than one exception is signaled and default exception handling is enabled, then default handling of both exceptions ensues. Conversely, if alternate exception handling is enabled, then alternate exception handling of only the overflow or underflow exception ensues. Overflow and underflow are deemed more important than inexact.

²The round to nearest, ties to even rounding mode is used in this example.

³The underflow exception may or may not be accompanied by the inexact exception under alternate exception handling.

Chapter 3

Related Research

A fundamental operation for any hardware implementation of decimal arithmetic is multiplication, which is integral to the decimal-dominant applications found in financial analysis, banking, tax calculation, currency conversion, insurance, and accounting. As will be explained, a key component of multiplication is the accumulation of multiplicand multiples or partial products. This accumulation involves decimal addition. Thus, related research in the areas of decimal addition and decimal multiplication are described in this chapter. The manner in which decimal numbers are added, and stored or otherwise manipulated for that matter, are influenced by the encoding chosen for each decimal digit. Some encoding is necessary when decimal numbers are represented as a string of digits, as opposed to a binary integer. Because of the importance of encodings, this chapter also covers several decimal digit encodings and a compression technique for the external storage of DFP data.

3.1 Decimal Encodings

In Section 2.4, the two competing external storage formats for DFP numbers defined in 754-2008 are described. For the BID format, the significand is stored as an integer. In contrast, the significand in the DPD format is a compressed string of BCD digits. As today's computers employ the two-state transistor for both memory and logic, the ten-state decimal digit must utilize some form of encoding (as ten is not a power of two) when the DPD storage format is being used. This section describes various decimal digit encodings and the Densely Packed Decimal significand encoding defined in 754-2008.

3.1.1 Digit Encodings

Several binary encodings appear in Table 3.1. Table 3.2 shows all 16 combinations of four binary bits and their meaning in the same BCD encodings of Table 3.1. Although there are many more encodings [105], the encodings appearing in this dissertation are presented in these two tables. Converting numbers stored in ASCII or EBCDIC, two common formats, to BCD-8421 is straightforward as the four LSBs are identical.

A common encoding, identified as BCD-8421 or just BCD, uses the binary weights of a four-bit binary integer. One advantage of BCD numbers over binary numbers is in entering and displaying decimal numbers. Because each digit is distinct, simple four-bit circuits can be used for I/O conversion. BCD has been in use since the EDVAC [53].

As can be seen in Table 3.1, the Excess-3 encoding, XS3, is the BCD encoding biased with 3 (0011_2). XS3 was used on the UNIVAC I [49] to avoid all zeros when representing a number and to benefit the addition of two numbers. This benefit

Table 3.1: Select Binary-Coded Decimal Encodings

Decimal Value	Encoding Format				
	BCD[-8421]	[BCD-]XS3	BCD-4221	BCD-5211	BCD-5421
0	0000	0011	0000	0000	0000
1	0001	0100	0001	0010 or 0001	0001
2	0010	0101	0100 or 0010	0100 or 0011	0010
3	0011	0110	0101 or 0011	0110 or 0101	0011
4	0100	0111	1000 or 0110	0111	0100
5	0101	1000	1001 or 0111	1000	1000 or 0101
6	0110	1001	1100 or 1010	1010 or 1001	1001 or 0110
7	0111	1010	1101 or 1011	1100 or 1011	1010 or 0111
8	1000	1011	1110	1110 or 1101	1011
9	1001	1100	1111	1111	1100

is because the six invalid combinations in BCD are bypassed during the add thus enabling the addition to be performed with binary adders followed by a ± 3 correction (see Section 3.2). Another advantage of XS3 is that it is self-complementing. That is, the nine's complement, used in decimal subtract operations, is achieved by logical inversion of each bit in the digit.

BCD-4221 is an important self-complementing code as the sum digit emerging from a binary CSA is also BCD-4221 (when both inputs are BCD-4221), so it can be used directly in a subsequent binary CSA. The last two codes in Table 3.1, BCD-5211 and BCD-5421, both self-complementing as well, are useful as their binary doubled value yields BCD-4221 and BCD-8421, respectively. More details about the benefits and uses of BCD-4221, BCD-5211, and BCD-5421 appear in Section 5.1.2.

Table 3.3 contains the signed-digit code developed by Svoboda [4]. This code, also self-complementing, is a redundant code which allows for carry-free addition [106]. There is a cost associated with carry-free addition, namely, the conversation from a

Table 3.2: Some Binary-Coded Decimal Values

Binary Nibble	Decimal Value				
	BCD[-8421]	[BCD-]XS3	BCD-4221	BCD-5211	BCD-5421
0000	0	invalid	0	0	0
0001	1	invalid	1	1	1
0010	2	invalid	2	1	2
0011	3	0	3	2	3
0100	4	1	2	2	4
0101	5	2	3	3	5
0110	6	3	4	3	6
0111	7	4	5	4	7
1000	8	5	4	5	5
1001	9	6	5	6	6
1010	invalid	7	6	6	7
1011	invalid	8	7	7	8
1100	invalid	9	6	7	9
1101	invalid	invalid	7	8	invalid
1110	invalid	invalid	8	8	invalid
1111	invalid	invalid	9	9	invalid

non-redundant code into a redundant code and vice versa. More information about the benefit and use of redundant codes, in particular, the code of Table 3.3, appears in Section 4.1.2.

As only ten values are represented by each of the four-bit BCD formats, approximately 38% of the combinations are wasted. With the Svoboda signed-digit code, approximately 56% of the combinations are wasted. To reduce the number of wasted combinations, IEEE 754-2008 defines a decimal encoding scheme for the external storage of DFP data.

Table 3.3: Some Signed-Digit Codes

Decimal Value	Alternate Representation	Svoboda [4]	RBCD [107]
+7	7	out of range	0111
+6	6	10010	0110
+5	5	01111	0101
+4	4	01100	0100
+3	3	01001	0011
+2	2	00110	0010
+1	1	00011	0001
+0	0	00000	0000
-0	$\bar{0}$	11111	unused
-1	$\bar{1}$	11100	1111
-2	$\bar{2}$	11001	1110
-3	$\bar{3}$	10110	1101
-4	$\bar{4}$	10011	1100
-5	$\bar{5}$	10000	1011
-6	$\bar{6}$	01101	1010
-7	$\bar{7}$	out of range	1001

3.1.2 Significand Encoding

The decimal encoding scheme for external storage (described in Section 2.4) is based on the Densely Packed Decimal, DPD, compression technique [3]. This technique, developed by Cowlishaw, offered an improvement in conversion delay over the prevailing compression technique developed by Chen and Ho [99]. Via DPD encoding, strings of three BCD digits (12 bits) are compressed into ten bits. By compressing 1,000 values (0 - 999) into 10 bits (1,024 possible values), the number of wasted combinations is reduced from approximately 38% to only about 2%. The following 24 combinations of 10 bits $\{01x11x111x, 10x11x111x, 11x11x111x\}$ are not to be pro-

duced as a result of a computation operation, according to IEEE 754-2008. However, all 1,024 combinations of 10 bits are accepted as valid for DPD decoding. The feature of not allowing any redundancy between the encoded and decoded values eliminates the need to store the original encoded data, as only the value of encoded data need be preserved through a decode/encode cycle. Table 3.4 shows how three BCD digits are compressed into ten bits, the result being called a *decret*. Table 3.5 shows how a decret is uncompressed into three BCD digits. Both the encode and decode functions can be achieved in a few CMOS gate delays. In these tables, the \cdot and $+$ symbols are arithmetic multiply and add, respectively. The reader is referred to Appendix B for a complete description of the notation used in this dissertation.

Table 3.4: Encoding a Densely Packed Decimal Decret [3]

$d_0[0]d_1[0]d_2[0]$	$b[0] b[1] b[2]$	$b[3] b[4] b[5]$	$b[6]$	$b[7] b[8] b[9]$
000	$d_0[1 : 3]$	$d_1[1 : 3]$	0	$d_2[1 : 3]$
001	$d_0[1 : 3]$	$d_1[1 : 3]$	1	$0\ 0\ d_2[3]$
010	$d_0[1 : 3]$	$d_2[1 : 2] d_1[3]$	1	$0\ 1\ d_2[3]$
011	$d_0[1 : 3]$	$1\ 0\ d_1[3]$	1	$1\ 1\ d_2[3]$
100	$d_2[1 : 2] d_0[3]$	$d_1[1 : 3]$	1	$1\ 0\ d_2[3]$
101	$d_1[1 : 2] d_0[3]$	$0\ 1\ d_1[3]$	1	$1\ 1\ d_2[3]$
110	$d_2[1 : 2] d_0[3]$	$0\ 0\ d_1[3]$	1	$1\ 1\ d_2[3]$
111	$0\ 0\ d_0[3]$	$1\ 1\ d_1[3]$	1	$1\ 1\ d_2[3]$

Table 3.5: Decoding a Densely Packed Decimal Declet [3]

$b[6, 7, 8, 3, 4]$	$d_0[0 : 3]$	$d_1[0 : 3]$	$d_2[0 : 3]$
$0xxxx$	$4 \cdot b[0] + 2 \cdot b[1] + b[2]$	$4 \cdot b[3] + 2 \cdot b[4] + b[5]$	$4 \cdot b[7] + 2 \cdot b[8] + b[9]$
$100xx$	$4 \cdot b[0] + 2 \cdot b[1] + b[2]$	$4 \cdot b[3] + 2 \cdot b[4] + b[5]$	$8 + b[9]$
$101xx$	$4 \cdot b[0] + 2 \cdot b[1] + b[2]$	$8 + b[5]$	$4 \cdot b[7] + 2 \cdot b[8] + b[9]$
$110xx$	$8 + b[2]$	$4 \cdot b[3] + 2 \cdot b[4] + b[5]$	$4 \cdot b[7] + 2 \cdot b[8] + b[9]$
11100	$8 + b[2]$	$8 + b[5]$	$4 \cdot b[7] + 2 \cdot b[8] + b[9]$
11101	$8 + b[2]$	$4 \cdot b[3] + 2 \cdot b[4] + b[5]$	$8 + b[9]$
11110	$4 \cdot b[0] + 2 \cdot b[1] + b[2]$	$8 + b[5]$	$8 + b[9]$
11111	$8 + b[2]$	$8 + b[5]$	$8 + b[9]$

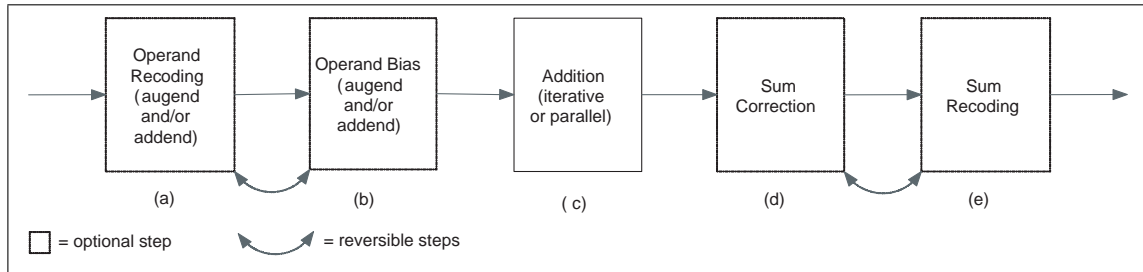


Figure 3.1: Generalized Flow of DFXP Addition

3.2 Decimal Addition

Figure 3.1 shows the generalized flow for decimal fixed-point (DFXP) addition, which is described as follows. A decoding step must be performed (Figure 3.1(a)), if one or both of the operands are in a compressed format or in a format incongruent with the adder hardware (Figure 3.1(c)). Some common recoding and compression schemes are presented in Section 3.1. If the format and weighting of bits are appropriate for the intended adder, a biasing step (Figure 3.1(b)) may still be necessary depending on whether the adder component performs direct decimal addition or binary addition. After adding the two operands (Figure 3.1(c)), a correction step (Figure 3.1(d)) may be necessary to remove any residual bias. The final step is to encode the result (Figure 3.1(e)), if it is to be expressed in a compressed format or a format different from the adder output. Note the decoding and biasing steps, as well as the correcting and encoding steps, may be combined, or performed in reverse order.

Because each operands' ten-state digits are represented with a set of two-state devices, there are either unused combinations or there is redundancy within the set representing the digit. The unused or redundant combinations complicate the addition in each digit position as well as the carry generation and propagation across digit boundaries.

Without increasing the number of bits used to represent the ten states beyond the minimum of four, three approaches have been developed for decimal addition. These are direct decimal addition, binary addition with bias and correction, and binary addition with correction. Distinct from these approaches are those that convert the four-bit digit encoding into five-bit digit encodings. The additional available representations enable carry-free addition. A variety of four- and five-bit encodings and associated adders have been developed. The different schemes are presented in the following subsections.

3.2.1 Bias, Binary Addition, and Correction

The use of a binary adder requires each digit to be in a binary weighted form. The addition of two decimal digits of the same order yields a sum in the range $\{0, \dots, 18\}$ ¹. When using a binary adder for the addition, there are four LSBs emerging from the adder, which represent the LSD of the sum, and there is a carry from the adder, which represents the MSD of the sum. The LSD of the sum may actually range from $\{0, \dots, 15\}$, instead of just $\{0, \dots, 9\}$. In the case of the four LSBs being in the range $\{10, \dots, 15\}$, the four LSBs need to be adjusted to bring them into the valid range for a decimal digit. This can be achieved by incrementing the four LSBs by six. Also for this case, the carry needs to be changed from zero to one to reflect a carry of ten. In the case of the carry being one and the four LSBs being in the range $\{0, 1, 2\}$ ², the sum needs to be incremented by six to adjust the weight of the carry from 16 to 10. Thus, there are two different reasons for the correction, but both situations are handled by the addition of six.

In the context of a binary carry-propagate adder, one may conclude a single cor-

¹The range is $\{0, \dots, 19\}$, if there is a carry-in.

²The range is $\{0, \dots, 3\}$, if there is a carry-in.

rective step is necessary to yield the proper decimal result. However, if during the correction step the addition of six in a particular digit position results in a carry-out, this carry may ripple to higher-order digits. Supporting this scenario requires successive correction steps or a second carry-propagate adder. Figure 3.2 shows how multiple corrections are necessary. An alternative approach to remove the latter carries is to employ a speculative approach in the following manner.

	0000 1001 0101 0110	=	956 ₁₀ first operand
	<u>+ 0000 1000 0100 0101</u>	=	845 ₁₀ second operand
=	0001 0001 1001 1011	=	119B ₁₆ first preliminary sum
	✓ ✓	:	carry emerged or invalid decimal digit
	<u>+ 0000 0110 0000 0110</u>	=	0606 ₁₆ first correction
=	0001 0111 1010 0001	=	17A1 ₁₆ second preliminary sum
	✓	:	invalid decimal digit
	<u>+ 0000 0000 0110 0000</u>	=	0060 ₁₆ second correction
=	0001 1000 0000 0001	=	1801 ₁₀ final result

Figure 3.2: Successive Correction Example

If one speculates the sum in each digit position will exceed nine, then the former correction step of adding six may be performed prior to the addition [105]. In so doing, any resulting carry is now of weight 10, not 16, and only the traditional carry propagation need be taken into account. A corrective step is still necessary in each digit position which does not yield a carry, as this indicates the sum is actually less than ten. The correction involves subtracting six (or adding ten), which does not affect the next more significant digits.

The benefits of this bias and correction approach include the ability to utilize highly optimized binary adders, the potential for re-use of existing circuitry, and the

possibility of supporting both binary and decimal addition with the same binary adder hardware [108–110]. If this scheme is used in a system in which BCD is the storage format for decimal numbers, then steps (b), (c), and (d) in Figure 3.1 are needed. Steps (a) and (e) are also required if the BCD digits are stored in a compressed format. An example of the bias and correction approach to perform the addition of $956 + 845$ in BCD format is shown in Figure 3.3.

	0000 1001 0101 0110	=	956_{10}	first operand
	<u>+ 0110 0110 0110 0110</u>	=	6666_{16}	biases
=	0110 1111 1011 1100	=	$6FBC_{16}$	
	0110 1111 1011 1100	=	$6FBC_{16}$	
	<u>+ 0000 1000 0100 0101</u>	=	845_{10}	second operand
=	0111 1000 0000 0001	=	7801_{16}	preliminary sum
	✓	:	no carry	
	0111 1000 0000 0001	=	7801_{16}	preliminary sums
	<u>+ 1010 0000 0000 0000</u>	=	-6000_{16}	corrections
=	0001 1000 0000 0001	=	1801_{10}	final result
	✓	:	carry ignored	

Figure 3.3: Bias and Correction Example

In [111], Grupe discloses a parallel adder that utilizes the bias, binary addition, and correction approach for operands comprised of BCD digits. In the first stage of this adder, one of the operands is either biased with six or binary complemented. The latter is performed when the operation is an effective subtraction. In [108], Anderson presents a combined binary/decimal adder which generates both an intermediate result and a precorrected result (i.e., six is added to each digit position of the intermediate result). Then, as the carry bit stabilizes in each digit position, it is used to select

the appropriate digit (non-corrected or precorrected) in each position as appropriate.

As described, the addition of a bias of six may be achieved by adding n to each digit in the augend (addend) and $6 - n$ to each digit in the addend (augend), where $0 \leq n \leq 6$. As mentioned in Section 3.1, the designers of some early computers chose $n = 3$. This XS3 encoding, shown in Table 3.1, is advantageous for subtraction as the nines' complement can be obtained by simply inverting all the bits (i.e., ones' complement). Different from the case for BCD when the sum (difference) is selectively corrected, with XS3 the sum (difference) is always corrected. For example, when the digit in the intermediate sum is less than ten, three must be subtracted. And when the digit is greater than or equal to ten, three must be added. If binary addition is used in a system in which XS3 is the storage format for decimal numbers, or in which XS3 is used internally, then only steps (c) and (d) in Figure 3.1 are needed. In [112], Thompson *et al.* present a DFP adder employing XS3 encoding and compliant with IEEE 754-2008.

3.2.2 Binary Addition and Correction

One may eliminate the bias step (Figure 3.1 (b)) at the expense of increasing the delay or complexity of the correction step (Figure 3.1 (d)). This is explored in [113] in the context of an iterative DFXP multiplier (see Section 4.1). In this work, decimal digits are “allowed” to range from $\{0, \dots, 15\}$, called an *overloaded decimal representation*. In this representation, each four-bit grouping of binary bits corresponds to a decimal digit and has the same weights as BCD-8421 code, however, its value can exceed ten. As the partial products are added one at a time in an iterative manner, the carries from each digit, representing 16 times the current order, result in the addition of a six in that digit position in the next iteration. The corrective

addition of six in select digit positions effectively reduces the weight of each carry from 16 to 10. At the end of the partial product accumulation, another corrective step must be taken to bring all the invalid values into a valid decimal range of $\{0, \dots, 10\}$. Referring to Figure 3.1 and assuming the operands are comprised of compressed BCD digits, the binary addition and correction scheme follows steps (a), (c), (d) and (e).

In [114], Vazquez *et al.* presents an improvement to this scheme by selectively speculating the addition of six. There is a single case in which the speculation may be incorrect, which is when sum in the digit position is eight. In this case, the correction can be achieved off the critical path, such that the overall latency of addition is not affected.

3.2.3 Direct Decimal Addition

An alternative to using binary addition is using *direct* decimal addition. That is, implementing logic which accounts for the unused or redundant combinations in each digit position and directly generates the correct decimal digit sum and carry. Assuming the operands are in BCD format, an adder using this approach accepts as inputs two, four-bit BCD digits, x_i and y_i , along with a one-bit carry-in, $c_{i+1}[0]$, and directly produces a four-bit BCD sum digit, s_i , and a one-bit carry-out, $c_i[0]$, such that

$$(c_i[0], s_i) = x_i + y_i + c_{i+1}[0] \quad (3.1)$$

where bit 0 is the MSB of the 4-bit BCD digit, and the weight of $c_i[0]$ is 10 times the weight of s_i . The reader is referred to [115] for a detailed description of this approach, which was used for example on the IBM System/360 Model 195³. The equations for performing the direct decimal addition of two BCD digits are as follows, where $s_i[0..3]$,

³This technique is also used in the multiplier described in Section 4.1.1.

$c_{i+1}[0]$, and $c_i[0]$ are the sum, carry-in and carry-out for the i^{th} digit position.

$$g_i[j] = x_i[j] \wedge y_i[j] \quad 0 \leq j \leq 3 \quad \text{“generate”}$$

$$p_i[j] = x_i[j] \vee y_i[j] \quad 0 \leq j \leq 3 \quad \text{“propagate”}$$

$$h_i[j] = x_i[j] \oplus y_i[j] \quad 0 \leq j \leq 3 \quad \text{“half-adder”}$$

$$k_i = g_i[0] \vee (p_i[0] \wedge p_i[1]) \vee (p_i[0] \wedge p_i[2]) \vee (g_i[1] \wedge p_i[2]) \quad \text{“columns 842 generate”}$$

$$l_i = p_i[0] \vee g_i[1] \vee (p_i[1] \wedge g_i[2]) \quad \text{“columns 842 propagate”}$$

$$c_i[1] = g_i[3] \vee (p_i[3] \wedge c_i[3]) \quad \text{“carry out of 1’s position”}$$

$$s_i[3] = h_i[3] \oplus c_{i+1}[0]$$

$$s_i[2] = ((h_i[2] \oplus k_i) \wedge \overline{c_i[3]}) \vee ((\overline{h_i[2]} \oplus l_i) \wedge c_i[3])$$

$$s_i[1] = (\overline{p_i[1]} \wedge g_i[2]) \vee (\overline{p_i[0]} \wedge h_i[1] \wedge \overline{p_i[2]}) \vee ((g_i[0] \vee (h_i[1] \wedge h_i[2])) \wedge \overline{c_i[3]}) \vee$$

$$(((\overline{p_i[0]} \wedge \overline{p_i[1]} \wedge p_i[2]) \vee (g_i[1] \wedge g_i[2]) \vee (p_i[0] \wedge p_i[1])) \wedge c_i[3])$$

$$s_i[0] = ((\overline{k_i} \wedge l_i) \wedge \overline{c_i[3]}) \vee (((g_i[0] \wedge \overline{h_i[0]}) \vee (\overline{h_i[0]} \wedge h_i[1] \wedge h_i[2])) \wedge c_i[3])$$

$$c_i[0] = k_i \vee (l_i \wedge c_i[3])$$

The k_i term is active when the sum of the three MSB columns of digit i is ten or greater (which will *generate* a carry). The l_i term is active when the sum of the three MSB columns of digit i is eight or greater (which will *propagate* a carry across these bit positions). Referring to Figure 3.1, only step (c) is necessary in this adder scheme with the operands comprised of BCD digits, and steps (a) and (e), if the operands’ digits are stored in a compressed form.

3.2.4 Redundant Addition

In each of the preceding subsections, the addition techniques have been presented under the assumption the operands are comprised of BCD digits and the result is of the same format. However, there are encodings which allow multiple ways to represent the same decimal digit value, and there are multiple ways to express multiple-digit decimal words. This flexibility is generally referred to as redundancy. The benefits of redundant representation may be twofold. Redundancy in the digits often enables a reduction in the complexity of the digit adder. Redundancy across the digits may reduce complexity, but depending on the manner of redundancy, it may enable carry propagation to be avoided. Avoiding carry propagation allows for the fast addition of multiple operands or partial products, which is necessary in a variety of multiplication and division schemes and in the accumulation portion of many digital signal processing algorithms. Further, this non-propagating addition can be performed in constant time regardless of the number of digits in the operands.

An example of redundancy within a decimal digit is the BCD-4221 encoding (see Table 3.1 on page 42). With this encoding, values in the range $\{2, \dots, 7\}$ can be represented in two different ways. Some examples of redundancy across decimal digits involve using more than four bits in each digit position such that the range of values can be extended. The extended range may allow more than one value in a particular bit position to be represented (such as *carry-save format* [116]), or it may also allow the value to be either positive or negative (*signed-digit format* [117]). Both of these categories of redundancy are now discussed.

Carry-save Format

Consider two operands of BCD digits entering a bank of binary full adders. The outputs of these full adders are a vector of four-bit sums and a vector of one-bit carries. These outputs, collectively called *carry-save format*, are redundant as the same value can be represented in multiple ways with these two vectors. Although *carry-save addition*, as this is called, is commonly used on binary data, the concept can be applied to decimal data. As an example of a decimal carry-save adder, consider the decimal 3:2 counter.

The 3:2 counter accepts three decimal inputs and produces a four-bit sum and a one-bit carry. By restricting one of the three inputs to be a one-bit carry-in signal, then the maximum sum in a single digit position is 19 ($9 + 9 + 1$). Thus, the carry out of each digit position is at most one, and therefore, only one bit is needed for the carry bit. With this restriction on the digits in one of the input operands, successive 3:2 counters can be used to reduce multiple operands by adding one new operand into each 3:2 stage. Figure 3.4 contains an example of a decimal 3:2 counter employing the direct decimal addition technique.

	0	1	0	1	=	101 ₁₀	carry input
	0000	1001	0101	0110	=	855 ₁₀	first input
	+	0000	1000	0100	0101	=	845 ₁₀ second input
=	0	1	0	1	=	1010 ₁₀	preliminary carry
	+	0000	0111	1001	0001	=	791 ₁₀ preliminary sum
		0001	1000	0000	0001	=	1801 ₁₀ non-redundant sum

Figure 3.4: Carry-save Addition Example

Carry propagation may still be avoided even if the range restriction on one input

is removed or more than three inputs are to be accumulated. This can be achieved by allowing the internal carries to propagate into the next two more significant digit positions or by allowing more than one bit to represent the carry out of each digit position. A decimal 3:2 counter and a decimal 4:2 compressor are introduced in [39].

In [118], Kenney *et al.*, introduce three algorithms for multi-operand addition. Two algorithms utilize a speculative approach by biasing prior to the binary carry-save addition, while the third algorithm performs all the binary carry-save additions first, and then takes successive steps to convert the binary sum into a decimal sum. Because of the speed of the binary carry-save adders, the non-speculative adder approach has less delay and similar area to the speculative adder approaches.

Signed-digit Format

Another way in which redundancy can occur across multiple digits is with signed-digits (Table 3.3 contains an example). With each digit having a sign associated with it, a value can be represented in multiple ways by converting a digit's sign, taking the radix complement of that digit, and incrementing the next more significant digit. Additionally, the encodings of the signed-digits may have redundancy as well.

In [119], Avizienis formalizes the class of signed-digit representations for any integer radix r . This research stood in contrast to the then contemporary methods [120] of redundant representations employing “stored carries or borrows” (analogous to the aforementioned carry-save format). In [4], a five-bit encoding is presented which represents the range of decimal numbers $\{-6, \dots, +6\}$. Converting from BCD to this encoding, and vice versa, requires only a couple gate delays. There is redundancy within each decimal digit position, and there are invalid combinations. Changing the sign of the digit involves a simple logical inversion of each bit. In [107], a four-bit encoding, called RBCD, is presented which represents the range of decimal numbers

$\{-7, \dots, +7\}$. The string “1000” is an invalid combination. The remaining fifteen combinations represent unique values (i.e., there is no redundancy within each decimal digit position). In contrast to Svoboda encoding, changing the sign of an RBCD digit involves taking the two’s complement. Both the Svoboda encoding and RBCD are shown in Table 3.3 on page 42. Figure 3.5 contains an example of the addition of two Svoboda signed-digit numbers.

=	<u>0000 1001 0101 0110</u>	=	956 ₁₀ first operand
=	00011 11100 10010 10011	=	1 $\bar{1}$ 6 $\bar{4}$
=	<u>0000 1000 0100 0101</u>	=	845 ₁₀ second operand
=	00011 11001 01100 01111	=	1 $\bar{2}$ 45
=	00011 11100 10010 10011	=	1 $\bar{1}$ 6 $\bar{4}$ first operand
=	<u>+ 00011 11001 01100 01111</u>	=	1 $\bar{2}$ 45 second operand
=	00110 11001 11111 00011	=	2 $\bar{2}$ 01 signed-digit sum
=	<u>00110 11001 11111 00011</u>	=	2 $\bar{2}$ 01 signed-digit sum
=	0001 1000 0000 0001	=	1801 ₁₀ non-redundant sum

Figure 3.5: Signed-digit Addition Example

After all the operands or partial products are accumulated to produce an intermediate result in carry-save form or signed-digit form, a single carry-propagation is all that is generally necessary to convert the redundant representation of the sum into a non-redundant sum. One advantage of the decimal carry-save format over signed decimal digits, is when the carry vector is comprised of single-bit in each digit position, the final carry-propagate adder can be simplified. Another advantage, if direct decimal addition is used in the carry-save adder, is that no conversion of the operands is necessary prior to entering the carry-save adder.

Both the carry-save format and the signed-digit format have been used to reduce the cycle delay or the overall latency of decimal multiplication. Decimal carry-save addition is employed in the multiplier designs presented in Subsections 4.1.1 and 5.1.1. Binary carry-save addition is employed in the multiplier design of Section 5.1.2. Decimal addition using signed-digits is utilized in the multiplier design described in Section 4.1.2.

3.2.5 Subtraction via End-Around Carry Addition

A straightforward method of subtraction entails the generation of the radix complement of one of the significands, the addition of this complemented significand with the other significand, and in the absence of a carry-out, the possible radix complement of the difference. For the last step, the inverse of the carry may be brought end-around and added into the LSD position of the difference (after performing diminished-radix complementation). The shortcoming of this method is twofold. First, the smaller operand is generally not known *a priori* and therefore either two adders are needed or the subtraction must be performed twice. Second, if the radix complement of the difference is needed, the addition of the inverted carry-out leads to a carry-propagate addition.

To address these problems, subtraction can be performed via an end-around-carry adder with some unique properties [121]. Namely, the adder has the properties of diminished-radix complementation on either significand and integrated support for handling the addition of the carry-out into the existing carry network. Stated another way, the benefit of using this type of adder is its ability to support $A - B$ and $B - A$ simultaneously via only diminished radix complements.

The following equations, wherein an overbar indicates a nine's complement, evince

how the same addition operation can be used to realize subtraction, regardless of which operand is larger.

$$\begin{aligned}
 A > B : A - B &= A + (-B) \\
 &= A + (\overline{B} + 1) \\
 &= A + \overline{B} + 1
 \end{aligned} \tag{3.2}$$

$$\begin{aligned}
 B > A : B - A &= -(A - B) \\
 &= -(A + (-B)) \\
 &= -(A + (\overline{B} + 1))
 \end{aligned} \tag{3.3}$$

$$\begin{aligned}
 &= -(A + \overline{B}) - 1 \\
 &= \overline{A + \overline{B}} + 1 - 1
 \end{aligned} \tag{3.4}$$

$$= \overline{A + \overline{B}} \tag{3.5}$$

It is evident from Equations 3.2.5 and 3.5 the same fundamental operation of $A + \overline{B}$ is used regardless of which significant is larger. If $A > B$, Equation should be used, and if $B > A$, Equation 3.5 should be used. There are two challenges to this proposed approach. One challenge is how to determine if $A > B$. And the other challenge is how to add a one to $A + \overline{B}$ to realize Equation when $A > B$ and we want this equation. Both of these challenges are overcome through the use of an end-around carry adder. This adder, as with all adders, can act as a greater than comparator; producing a carry when adding $A + \overline{B}$ and $A > B$. Unique to the end-around carry adder, however, is its ability to add its carry-out back into its carry network as a carry-in (described in the ensuing paragraphs). Thus, in a single pass through a single adder, B can be subtracted from A , and if $B > A$, a carry-out will be added back in to satisfy Equation . The following lists describes the steps to

performing subtraction using an end-around carry adder.

1. Use *effective_subtract* signal to selectively complement the addend
2. Add augend to selectively complemented addend using an end-around carry adder (i.e., using *carry_out* as *carry_in* when $A > B$)
3. Use $\overline{\text{carry_out}}$ from adder to selectively complement adder output

Obviously, an important design consideration with the end-around carry adder is the actual handling of end-around carry. If the carry is applied as a carry-in *after* the difference is determined, this carry could ripple from the LSD to the MSD. This approach is not desirable. A better approach is to integrate the end-around carry into the original carry generation terms. The following equations, where g_i and p_i are the generate and propagate from digit i , respectively, show the familiar carries from four digits. It is important to note that although the equations show the integration of the end-around carry into a four-digit carry network, the approach is scalable.

$$\begin{aligned}
 c_3 &= g_3 \vee p_3 c_i \\
 c_2 &= g_2 \vee p_2 g_3 \vee p_2 p_3 c_i \\
 c_1 &= g_1 \vee p_1 g_2 \vee p_1 p_2 g_3 \vee p_1 p_2 p_3 c_i \\
 c_0 &= g_0 \vee p_0 g_1 \vee p_0 p_1 g_2 \vee p_0 p_1 p_2 g_3 \vee p_0 p_1 p_2 p_3 c_i
 \end{aligned} \tag{3.6}$$

If the subtract operation involves only four digits, then c_0 is the carry out. This carry out, when it is added to the LSD (as described in Equation 3.2.5), will affect the carry of a particular digit when it is able to propagate through that respective digit.

This effect is shown in boldface type in the following updated carry equations.

$$\begin{aligned}
c_3 &= g_3 \vee p_3 c_i \vee \mathbf{P_3 C_0} \\
c_2 &= g_2 \vee p_2 g_3 \vee p_2 p_3 c_i \vee \mathbf{P_2 P_3 C_0} \\
c_1 &= g_1 \vee p_1 g_2 \vee p_1 p_2 g_3 \vee p_1 p_2 p_3 c_i \vee \mathbf{P_1 P_2 P_3 C_0} \\
c_0 &= g_0 \vee p_0 g_1 \vee p_0 p_1 g_2 \vee p_0 p_1 p_2 g_3 \vee p_0 p_1 p_2 p_3 c_i
\end{aligned}$$

Expanding c_0 in the last set of equations and reducing terms using the Boolean equivalences $x \vee xy = x$ and $xx = x$, yields the carry equations for a four digit end-around carry adder (the concept is extendable to larger groups of carries):

$$\begin{aligned}
c_3 &= \mathbf{P_3 G_0} \vee \mathbf{P_3 P_0 G_1} \vee \mathbf{P_3 P_0 P_1 G_2} \vee g_3 \vee p_3 c_i \\
c_2 &= \mathbf{P_2 P_3 G_0} \vee \mathbf{P_2 P_3 P_0 G_1} \vee g_2 \vee p_2 g_3 \vee p_2 p_3 c_i \\
c_1 &= \mathbf{P_1 P_2 P_3 G_0} \vee g_1 \vee p_1 g_2 \vee p_1 p_2 g_3 \vee p_1 p_2 p_3 c_i \\
c_0 &= g_0 \vee p_0 g_1 \vee p_0 p_1 g_2 \vee p_0 p_1 p_2 g_3 \vee p_0 p_1 p_2 p_3 c_i
\end{aligned}$$

Though not shown to simplify the presentation of the equations, each boldface type term must be ANDed with *effective_subtract* to prevent an inappropriate increment (as explained previously). This is accomplished by ANDing *effective_subtract* with the p_3 term. An interesting and desirable property of the reduced end-around carry equations is the delay to generate c_3 through c_1 is only modestly increased over that of c_0 , which had exhibited the worst delay of the carry generations.

The end-around carry approach described was not used in the multiplier designs presented in this dissertation. However, it is included to round out this section on decimal addition, as the technique could be used in decimal fused multiply-add.

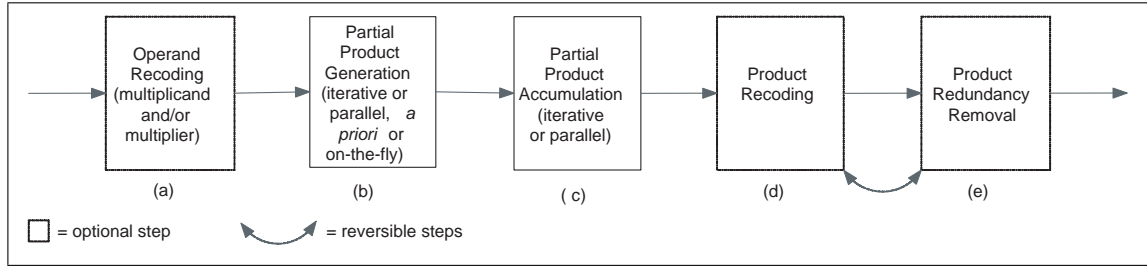


Figure 3.6: Generalized Flow of DFXP Multiplication

3.3 Decimal Multiplication

Figure 3.6 shows the generalized flow for DFXP multiplication. The multiplication operation, for both binary and decimal radices, requires the accumulation of multiples of the multiplicand. The accumulation may occur on a digit-by-digit basis, a word-by-digit basis, or a word-by-word basis. Decimal multiplication differs from binary multiplication in the number of multiplicand multiples required and the representation of digits. In binary multiplication, each non-zero binary digit in the multiplier contributes the value of the multiplicand at the order of the respective multiplier digit. However, in decimal multiplication, each nonzero decimal digit in the multiplier contributes one of nine different multiples of the multiplicand (based on the value and order of the respective multiplier digit).

3.3.1 Digit-by-Digit Multiplication

In a digit-by-digit multiplier, each digit of the multiplicand (A) is successively multiplied by each digit of the multiplier (B). Generally, both operands are traversed from the least significant digit to the most significant digit, as doing so reduces the width of the adder used to accumulate the partial products. That is, the digits in the intermediate product with less significance than the multiplier digit being evaluated

remain stable.

Digit-by-digit multipliers can be constructed with either a row-oriented or column-oriented approach to the digit-by-digit product accumulations. With the row-oriented approach, the multiplier operand is traversed one time from LSD to MSD while the multiplicand is traversed n times (where n is the number of digits in each operand) from LSD to MSD. Effectively, the partial products are developed and accumulated one digit at a time. Equation 3.7 shows the multiplication of A and B in terms of the sums of each digit-by-digit product.

$$A \cdot B = \sum_{i=n-1}^0 \sum_{j=n-1}^0 a_j b_i \cdot 10^{(2n-1)-(i+j)} \quad \text{“row-oriented approach”} \quad (3.7)$$

With the row-oriented approach, the multiplicand has to be left and right shifted, effectively, and the multiplier only has to be right shifted. Therefore, both a circular shifter and a right shifter are needed. As mentioned, each partial product is created in full, albeit one digit at a time. Since the LSD of a subsequent partial product must be added to the previous partial product, a carry-propagate adder $n + 1$ digits wide is needed. Also, the intermediate product must be shifted left and right to properly align it with the digit-by-digit product being produced. The row-oriented approach is shown in Figure 3.7.a.

With the column-oriented approach, the digit-by-digit products are generated and accumulated in such a manner that the product is finalized from the LSD to the MSD position. Thus, the shifter controlling the intermediate product only has to shift right (or not shift at all) to align with the next digit-by-digit product being produced. Also different from the row-oriented approach, both operands are traversed n times, being shifted left and right as appropriate through the use of circular shifters to product the appropriate digit-by-digit product. As the partial products are *not* produced in full,

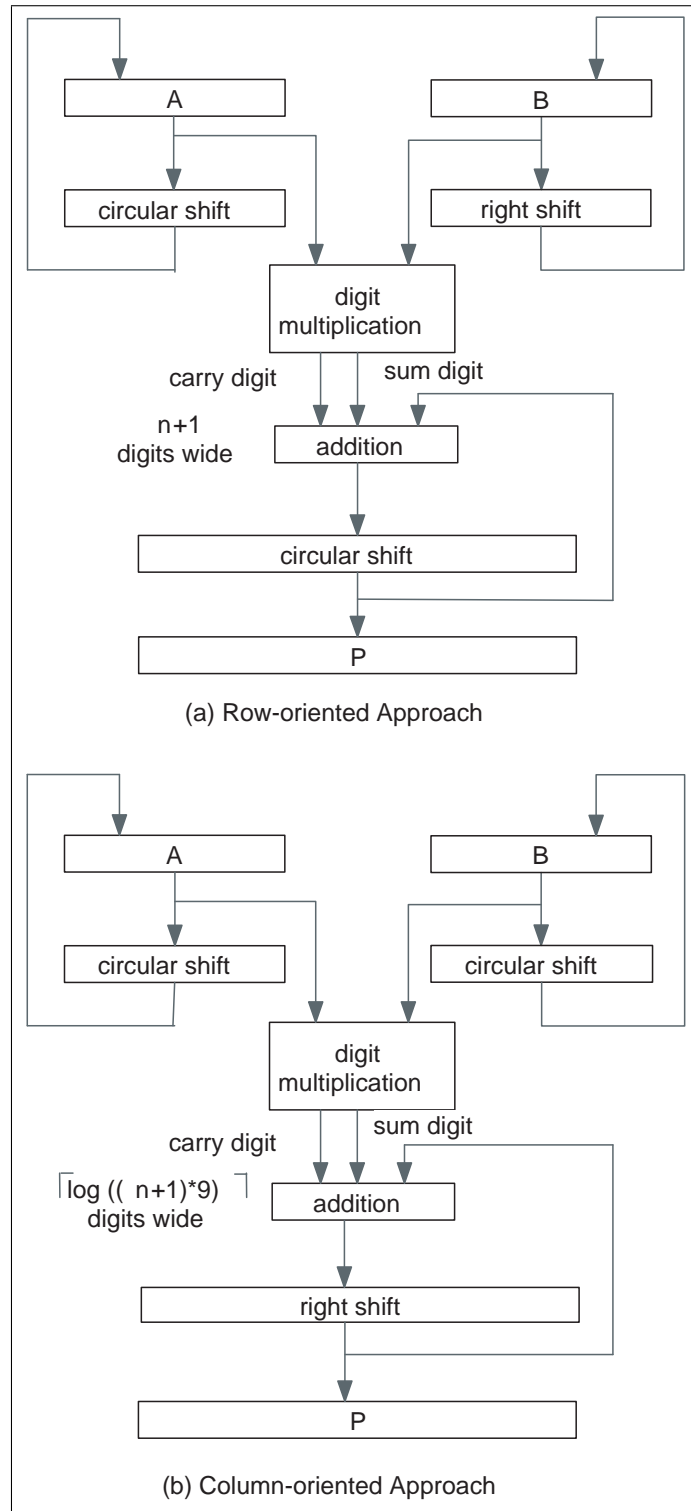


Figure 3.7: Generalized Design of DFXP Digit-by-digit Multiplication

the width of the carry-propagate adder only has to be as wide as the number of digits needed to represent the sum of all the digits of the same weight. The width of the adder can be expressed as $\lceil \log_{10}((n+1) \cdot 9) \rceil$ digits. The column-oriented approach is shown in Figure 3.7.b.

Referring to Figure 3.6, the digit-by-digit multiplication scheme using the row-oriented approach generates the partial products one digit at a time (Figure 3.6.b) and accumulates the partial products in an iterative, one digit at a time (Figure 3.6.c). The steps in Figure 3.6.a and Figure 3.6.e are necessary if the operands are comprised of compressed BCD digits. Further, steps in Figure 3.6.a and Figure 3.6.d are necessary if either or both operands are recoded to aid in the generation or accumulation of the partial products.

The digit-by-digit multiplies can be achieved via lookup tables (LUT) or via random logic. With a digit range of $\{0, \dots, 9\}$, there are a total of 100 input combinations which can result in 37 unique products. By recoding the operands using signed-digits, the number of unique products, ignoring the sign of the product, can be reduced. One such recoding changes the digit range to $\{-5, \dots, +5\}$, which reduces the number of unique products to 15 (ignoring the sign). This concept, described in [122], is explored in a multiplier design presented in Section 4.1.2.

In [123], Larson describes a digit-by-digit LUT scheme. The multiplier operand is traversed from least significant digit (LSD) to most significant digit (MSD) and a partial product is generated for each digit in the multiplier operand. The partial product is added along with the previous iteration's properly shifted intermediate product via a carry-propagate adder. In [124], Larson presents a second, faster implementation which employs the LUT scheme just described, but replaces the carry-propagate adder with a four-input carry-save adder. In [125], Ueda presents a LUT which accepts digits from each operand and carries from adjacent LUTs. Both of these schemes, and

similar digit-by-digit LUT schemes, require significant circuitry and delay to generate the digit-by-digit products, since each digit has a range of $\{0, \dots, 9\}$. Assuming the digit-by-digit multiply and its addition to the intermediate product can be accomplished in one cycle, the latency of the digit-by-digit multiplier of the resultant product is approximately n^2 cycles.

3.3.2 Word-by-Digit Multiplication

To reduce the latency of digit-by-digit multiplication, a word-by-digit approach may be used. Typically, designs using this approach iterate over the digits of the multiplier operand from LSD to MSD and, based on the value of the current digit, either successively add the multiplicand or a multiple of the multiplicand (i.e., a partial product). The partial products may be generated via lookup tables or random logic. Further, they can be generated on-the-fly or created *a priori* and stored. To reduce the complexity of partial product generation or to reduce the number of partial products to be stored, a subset of previously generated and stored multiplicand multiples may be used to generate the partial products on-the-fly. Equation 3.8 shows a summation of properly weighted partial products.

$$A \cdot B = \sum_{i=n-1}^0 b_i \cdot 10^{(n-1)-i} A \quad (3.8)$$

The generation of the multiplicand multiples is shown in Figure 3.6.b, with a simplified scheme which generates all the partial products, one of which is selected based on the value of the multiplier digit.

Referring to Figure 3.6, a word-by-digit multiplication scheme may generate the partial product based on a multiplier operand digit (Figure 3.6.b), add it to the accumulated set of partial products (Figure 3.6.c), and repeat these steps for the

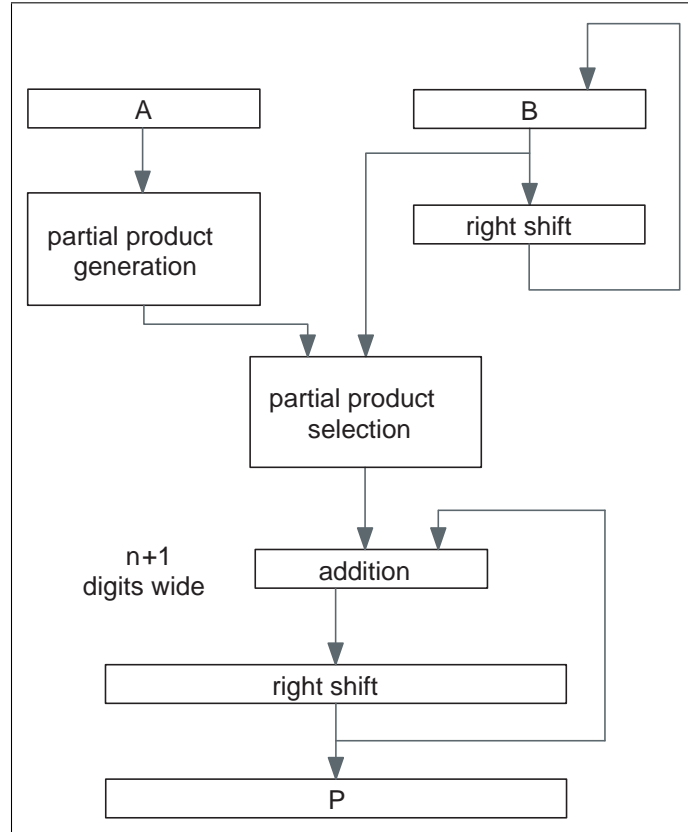


Figure 3.8: Generalized Design of DFXP Word-by-digit Multiplication

remaining multiplier operand digits. Alternatively, all the partial products may be generated *a priori* and then in a repetitive manner for each multiplier operand digit, a partial product is selected and accumulated. The steps 3.6.a and 3.6.e are necessary if the operands are comprised of compressed BCD digits. Further, steps 3.6.a and 3.6.d are necessary if either or both operands are recoded to aid in the generation or accumulation of the partial products.

Over the years, several word-by-digit multiplier designs have been proposed, including [126–131]. These designs, also called iterative DFXP multipliers, iterate over the digits of the multiplier and, based on the value of the current digit, either successively add the multiplicand or a multiple of the multiplicand. The multiples are

generated via LUTs or developed using a subset of previously generated multiples. All of these designs are DFXP multipliers.

To gain an appreciation of the latency of a word-by-digit multiplication scheme, consider Bradley *et al.*'s design [132] which requires up to nine cycles per multiplier digit with none of the multiples being stored. In an effort to improve performance, the leading zeros of both operands are counted, the shorter operand is made the multiplier, the iterative multiply is performed, and the leading zeros prefixed onto the product. Although the mechanism for addition is not described, it involves carry-propagation. Assuming decimal carry-propagate addition can complete in a single cycle, the designs employing an iterative addition approach require an average of $5n_1$ cycles, where n_1 is the number of significant digits in the smaller of the two operands.

The required number of multiplicand multiples is at most eight. If the multiplier operand digit is zero, no value is added to the intermediate product. If the multiplier operand digit is one, then the original multiplicand (or single) is added to the intermediate product. Hence, eight multiples are needed (i.e., double, triple, quadruple, quintuple, sextuple, septuple, octuple, and nonuple). To reduce the number of necessary multiples, the multiplier operand may be recoded into a redundant form such that the magnitude of each digit is less than nine. For example, if the multiplier operand is recoded into a range of $\{-5, \dots, +5\}$, then only four multiples are required (i.e., double, triple, quadruple, and quintuple) along with their additive inverse. This approach was first described by Cauchy [117] in 1840. The optional multiplier operand recoding step is shown in Figure 3.6.a.

As described in Section 3.1, a common representation for the operands' decimal digits is BCD. In order to reduce the complexity and/or latency of generating the multiplicand multiples and ultimately, the accumulation of partial products, the multiplicand operand may be recoded as well. This optional multiplicand operand

recoding step is shown in Figure 3.6.a. With recoding of the multiplicand or the multiplier comes negative digits, and the overhead of producing the additive inverse must be taken into account. Depending on the encoding chosen for the digits, the additive inverse may simply involve the logical inverse of each bit (single inverter delay), or it may involve an addition or subtraction (multiple gate delay).

As Chapter 4 focuses on research involving word-by-digit multiplier design, some introductory concepts are now presented. A straightforward approach to iterative DFXP multiplication is to iterate over the digits of the multiplier, B , and based on the value of the current digit, b_i , successively add multiples of A to a product register [105]. The multiplier is typically traversed from least significant digit to most significant digit, and the product register is shifted one digit to the right after each iteration, which corresponds to division by 10. This approach allows an $n + 1$ -digit adder to be used to add the multiples of A to the partial product register and one product digit to be retired each iteration. The multiples $2A$ through $9A$ can be calculated at the start of the algorithm and stored along with A to reduce delay. The equation for this iterative approach to decimal multiplication is as follows:

$$P_{i+1} = (P_i + A \cdot b_i) \cdot 10^{-1} \quad (3.9)$$

where $P_0 = 0$ and $0 \leq i \leq n - 1$. After n iterations, P_n corresponds to the final product P , albeit with the decimal point in the wrong location⁴. The most notable shortcomings of this approach are the significant area or delay to generate the eight multiples, the eight additional registers needed to store the multiples if they are not generated on-the-fly, and the delay to perform all these carry-propagate additions.

⁴The equation is shown in this manner as the proposed hardware implementations perform a right shift of the previous iteration's partial product. Ultimately, P_n needs to be left shifted to yield an integer result.

An alternative to storing all the multiples (called *primary multiples* or a *primary set*) is storing a reduced set of multiples. The ideal situation is to be able to quickly generate the reduced set of multiples and to easily generate the complete set of primary multiples from the stored multiples. If the reduced set of multiples can be generated from the multiplicand without any carry propagation, then this would allow for fast generation. Two properties are needed for the generation of multiplicand multiples without carry propagation. First, each digit position must be able to absorb any carry from the next less significant digit position. Second, a carry must not propagate beyond the next more significant digit. Two of the multiplicand multiples can be generated very easily when the digits are stored in BCD form, and therefore, are potential candidates for a reduced set of multiples. Specifically, generating the double and the quintuple of the multiplicand is straightforward as neither involves a carry propagation beyond the next more significant digit position [105].

When any BCD digit is doubled, its LSB initially becomes zero. Thus, when a carry-out of one occurs (for digit values in the range $\{5, \dots, 9\}$), it does not propagate beyond the next more significant digit's LSB, which is always zero. For quintupling, note that any BCD digit whose value is odd will initially become a five ('0101'), and any digit whose value is even will initially become a zero. Further, any digit whose value is one will not produce a carry-out, and any digit whose value is greater than one will produce a carry-out in the range $\{1, \dots, 4\}$. Thus, when a carry-out does occur, it will not propagate beyond the next more significant digit, which has a maximum value of five before the addition of the carry.

As an example of how a reduced set of multiplicand multiples can be used to generate all the multiples, suppose the $2A$, $3A$, $4A$, and $8A$ multiples are precalculated and stored along with A . Then all the other multiples can be obtained dynamically with, at most, a single addition (see second column of Table 3.6, where b'_i , b''_i , and b'''_i

represent factors of multiplicand multiples). This reduced set of multiples is called a *secondary set*, as no more than two members of the set need to be added to generate a missing multiple. Another reduced set of multiples is A , $2A$, $4A$, and $8A$ [130]. Advantages of this set include one fewer multiple and a one-to-one correspondence with the weighted bits of a BCD digit. That is, the bits in each BCD digit of the multiplier operand can be used directly to select the appropriate multiple(s). A disadvantage of this set is the increase in delay or area needed to handle the addition of three multiples. Three multiples are needed to create the $7A$ multiple (third column of Table 3.6). Because of this, the set A , $2A$, $4A$, and $8A$ is called a *tertiary set* of multiples. If the $-A$ multiple were available, then a secondary set could be formed as A , $-A$, $2A$, $4A$, and $8A$. The primary multiples can be obtained from this set as shown in the last column of Table 3.6. The $5A$ multiple can be used in addition to, or in replacement of, the $-A$ multiple to produce another secondary set.

Table 3.6: Generation of Primary Multiples from Different Multiples Sets

Multiplier Digit (decimal value of b_i)	Reduced Set of Multiples		
	$A, 2A, 3A, 4A, 8A$ $b'_i + b''_i$	$A, 2A, 4A, 8A$ $b'_i + b''_i + b'''_i$	$A, -A, 2A, 4A, 8A$ $b'_i + b''_i$
0	$0 + 0$	$0 + 0 + 0$	$0 + 0$
1	$0 + 1$	$0 + 0 + 1$	$0 + 1$
2	$0 + 2$	$0 + 2 + 0$	$0 + 2$
3	$0 + 3$	$0 + 2 + 1$	$4 + -1$
4	$0 + 4$	$4 + 0 + 0$	$4 + 0$
5	$4 + 1$	$4 + 0 + 1$	$4 + 1$
6	$4 + 2$	$4 + 2 + 0$	$4 + 2$
7	$4 + 3$	$4 + 2 + 1$	$8 + -1$
8	$8 + 0$	$8 + 0 + 0$	$8 + 0$
9	$8 + 1$	$8 + 0 + 1$	$8 + 1$

Subtraction via the $-A$ multiple can be accomplished using the ten's complement form or the nine's complement form. Since the result of the addition of the secondary multiples involving $-A$ will always be greater than zero, there is no difference between these two approaches. That is, the nine's complement form is obtained by subtracting each digit from nine ("1001") and adding the carry that would occur into the LSD. And the ten's complement form is obtained by developing the nine's complement form and adding one into the LSD; note that the carry that would occur is to be ignored.

At first glance, it appears a carry-in is needed to add one to the LSD when one of the secondary multiples is $-A$ (to perform ten's complement). But since one of the other partial multiples is going to be $4A$ or $8A$ when $-A$ is chosen, and the LSB of $4A$ and $8A$ is zero, it is acceptable and desirable to inject the one into this location. Thus, there is no carry-in.

If secondary multiples are used, Equation 3.9 is replaced by the following equation to describe the iterative portion of a decimal multiplier:

$$P_{i+1} = (P_i + A \cdot b'_i + A \cdot b''_i) \cdot 10^{-1} \quad (3.10)$$

In Equation 3.10, $A \cdot b'_i$ and $A \cdot b''_i$ are secondary multiples which together equal the proper primary multiple (i.e., $A \cdot b'_i + A \cdot b''_i = A \cdot b_i$). Although the secondary multiple approach reduces the delay and/or area and register count, it introduces the overhead of potentially one more addition in each iteration.

To appreciate the overhead of the additions to accumulate the partial product, consider the following. Assuming decimal carry-propagate addition can complete in a single cycle, a multiplier employing an iterative addition approach requires an average of $5n_1$ cycles, where n_1 is the number of significant digits in the smaller of the two operands. The number of cycles needed to accumulate the partial products can be

reduced in the ideal case to n_1 , if the multiplicand multiples are generated *a priori*. Consider the following designs from IBM.

In a recent publication [34], Busaba *et al.* describe the algorithm and hardware design for the fixed-point decimal multiplier on the IBM System z900. The goal of that design is to implement decimal multiplication in the fewest cycles with the hardware minimally modified to support a small set of DFXP operations. Thus, an existing binary adder was extended into a combined BFXP/DFXP adder capable of adding 16 BCD digits. Using a register file to store multiples of the multiplicand. The DFXP multiplication instruction requires an average of $11 + 2.4n_1$ cycles when the result is 15 digits or less and $11 + 4.4n_1$ cycles when the result is more than 15 digits. In the IBM Power6 [23] and System z10 [25] processors, a dedicated decimal arithmetic unit exists which supports DFP arithmetic in conformance with IEEE 754-2008. The goal of this design, originally developed for the Power6 and then extended to support the z/Architecture instruction set, was to consume as small an area as possible with as much of the IEEE 754-2008 DFP instructions implemented in hardware. The DFP multiplication instruction requires $19 + n$ cycles when $n = 16$ digits and $21 + 2n$ cycles when $n = 34$ digits. All of these designs have a decimal carry-propagate adder on the critical path in the iterative portion of the multiply algorithm.

As described in Subsections 3.2.4 and 3.2.1, the cycle time in the accumulate portion of the algorithm can be reduced by replacing the carry-propagate adder with a carry-save adder. In both [126] and [130], carry-save adders are employed. As an example, Ohtsuki's design [130] performs two binary carry-save additions and three decimal corrections in each iteration of partial product accumulation. In Section 4.1, several alternative word-by-digit multiplier designs are presented which take advantage of both the secondary multiple concept and the carry-save addition concept.

3.3.3 Word-by-Word Multiplication

Further reductions in latency can be achieved by performing word-by-word multiplication. Although this scheme became feasible for hardware implementation quite some time ago for binary multiplication (see [133] and [134]), it has only recently become feasible for decimal multiplication. Admittedly, a decades-old publication does exist which depicts the arrangement needed for word-by-word multiplication [135]. However, this design, by Pivnichny, presents an array of digit-by-digit multipliers and associated carry-save adders which is prohibitively expensive in both area and wiring.

Recently, two papers have been published describing word-by-word, or parallel DFXP multiplication. Lang and Nannarelli's parallel design [5] recodes each multiplier operand digit into two terms, $\{0, 5, 10\}$ and $\{-2, \dots, +2\}$. In so doing, only the decimal double and quintuple of the multiplicand are required, each of which is obtained quickly as there is no carry propagation [105]. This recoding scheme leads to two partial products for every digit of the multiplier operand. The set of partial products is reduced to two via BCD 3:2 carry-save adders (CSAs). In Vazquez *et al.*'s work, a family of parallel decimal multipliers is presented based on the recoding of the multiplicand digits into BCD-4221. This recoding enables the use of *binary* CSAs for the partial product accumulation, albeit with the carry digit needing a correction in the form of decimal doubling. Since binary CSAs are used in the partial product accumulation step, the tree can be overloaded to support binary multiplication. Further, three recodings of the multiplier operand are offered which trade-off the complexity/delay of partial product generation against the number of partial products to be accumulated. The recodings are $\{-5, \dots, +5\}$, $\{0, 4, 8\} + \{-2, \dots, +2\}$, and $\{0, 5, 10\} + \{-2, \dots, +2\}$. For a p -digit multiplier operand, the first recoding

option leads to $p + 1$ partial products with a relatively slow carry-propagate addition needed to yield the decimal triple, while the last two recoding options lead to $2p$ partial products with relatively fast generation of all the required multiples.

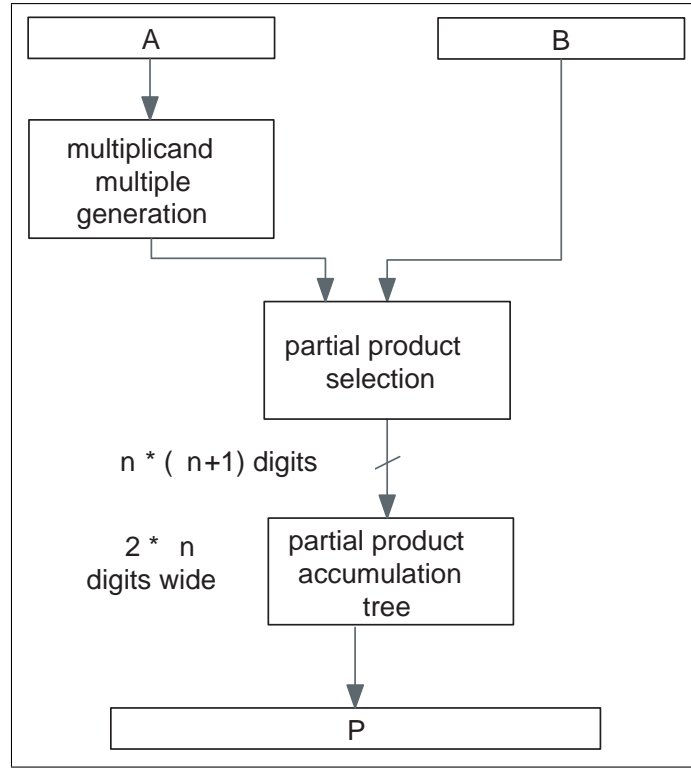


Figure 3.9: Generalized Design of DFXP Word-by-word Multiplication

The description of the Vazquez *et al.* parallel DFXP multiplier design [6] is deferred to Section 5.1.

3.3.4 Decimal Floating-point Multiplication

Aside from recently published papers by the author [8,41], there are several papers presenting hardware designs of DFP multiplication [23, 25, 87, 89]. The multiplier designs by Cohen *et al.* [87] and Bolenger *et al.* [89] are digit-serial and have long latencies. Furthermore, the results they produce do not comply with IEEE 754-

2008. In contrast, the multiplier designs used on the IBM Power6 [23] and System z10 [25] processors do conform with IEEE 754-2008. These multipliers are essentially the same, as they use the same word-by-digit algorithm. See Section 3.3.2 for a description of their latencies.

Two new DFP multiplier designs are presented in this dissertation. A DFP multiplier design using an iterative (i.e., word-by-digit) approach is described in Section 4.2. And a DFP multiplier design using a parallel (i.e., word-by-word) approach is described in Section 5.2. The two multiplier designs are compared and contrasted in Section 5.3.

Chapter 4

Iterative Multiplier Designs

The multiplier designs presented in this chapter iterate over the digits of the multiplier operand and successively produce and accumulate partial products based on multiples of the multiplicand. Because of the iterative accumulation, the throughput of these designs is less than one multiply per cycle, i.e., a new multiply instruction cannot start each cycle. However, the iterative nature of the algorithms implies a high degree of hardware re-use, and therefore, these designs are considered area efficient. Though one can “unroll” the accumulation of partial products to support a throughput of one multiply per cycle, the overhead in hardware is quite significant. Thus, these designs are most applicable to systems in which area is more important than throughput. The reader is referred to Section 3.3 for an overview of the fundamental steps of hardware multiplication.

The first multiplier design presented was published as [39] and appears in Section 4.1.1. This design utilizes decimal carry-save addition to produce a redundant internal form of the intermediate values during partial product accumulation. The design [39] contained the following novel features: decimal (3:2) counters and decimal (4:2) compressors, fast generation of multiplicand multiples that do not need to be

stored, and a simplified decimal carry-propagate addition to produce the final product. The second design presented was published as [40] and appears in Section 4.1.2. This design recodes the operands, generates each partial product on-the-fly, and utilizes a redundant digit encoding to improve the speed of partial product accumulation. The novelty in this design is due to the recoding the operands and the employing of signed-digits, thereby eliminating the pre-computation of multiples and efficiently accumulating the partial products. The iterative DFXP multiplier design employing decimal carry-save addition was chosen as the basis for an iterative DFP multiplier design, and the details of this DFP implementation [41] are presented in Section 4.2. This iterative DFP multiplier, the first published design compliant with IEEE 754-2008 [41], is novel in its mechanisms to support on-the-fly generation of the sticky bit, early estimation of the shift amount, and efficient decimal rounding. A comparison of the iterative DFP multiplier described in this chapter and a parallel DFP multiplier described in the next chapter appears in [42].

4.1 Fixed-point Designs

The iterative multiplier designs described in this chapter employ two different strategies yet achieve very similar latencies. Both approaches feature a reduced set of multiplicand multiples. However, the design of Section 4.1.1 precomputes the multiples and uses decimal CSAs to accumulate the partial products, while the design of Section 4.1.2 generates the multiples as needed and uses signed-digit adders to accumulate the partial products. The presented designs are compared in Section 4.1.3.

4.1.1 Multiplier Employing Decimal CSAs

Before describing the finalized multiplier design utilizing decimal CSAs, it may be beneficial to introduce a preliminary multiplier design and speak to its shortcomings. In Section 3.3.2, justification is provided for producing a subset of the multiplicand multiples and then adding at most two of these multiples together to form a partial product. For reference, Equation 3.10, representing the addition of a partial product ($A \cdot b'_i + A \cdot b''_i$) to the previous iteration's accumulated partial product (P_i), is shown here:

$$P_{i+1} = (P_i + A \cdot b'_i + A \cdot b''_i) \cdot 10^{-1}$$

Since the multiplier may need to handle operands up to 34 decimal digits in length to support IEEE 754-2008 [20], it is unlikely a single decimal carry-propagate addition, let alone the two additions shown in this equation, can be performed in one cycle. A substantial improvement in delay can be obtained by using decimal CSAs (see Section 3.2.4). Using the decimal carry-save addition approach, the equation to represent the addition of a partial product to the previous iteration's accumulated

partial product can now be expressed as two equations:

$$(PS'_i, PC'_i) = PS_i + PC_i + A \cdot b'_i \quad (4.1)$$

$$(PS_{i+1}, PC_{i+1}) = (PS'_i + PC'_i + A \cdot b''_i) \cdot 10^{-1} \quad (4.2)$$

where PS_i and PS'_i are the partial product sums comprised of four-bit BCD digits and PC_i and PC'_i are the partial product carries (one bit for each PS digit) at the end of the i^{th} iteration.

Figure 4.1 is a diagram of a decimal multiplier that implements Equations 4.1 and 4.2 in a single cycle. The top portion of the design, which is above the partial product register, performs the iterative equations. The bottom portion generates and stores the multiples of A , and produces the final product. To understand the design better, note that the same decimal carry-propagate adder is used to generate the multiples at the beginning of the multiplication and produce the final product at the end. As shown, the design takes four cycles to generate the secondary multiples ($2A$, $3A$, $4A$, and $8A$), one cycle for each multiplier digit to add up all the partial products, and one cycle to produce the final product. Thus, it has a latency of $n + 5$ cycles and can begin a new multiplication every $n + 5$ cycles.

Note the figure for this multiplier design contains registers that are identified as “Master/Slave”, “Master”, or “Slave”. This is because the designs are implemented with a two-phase clock. The latches in the “Master” registers are active when the clock is high (phase 1), and the latches in the “Slave” registers are active when the clock is low (phase 2). Also noteworthy is the absence of selection logic on the data inputs to the registers. (For example, in Figure 4.1, the output of the “Mid-Cycle Register” toward the bottom of the design is fed to five different “Slave” registers.) The selecting of the data is accomplished through the use of gated clocks. Effectively,

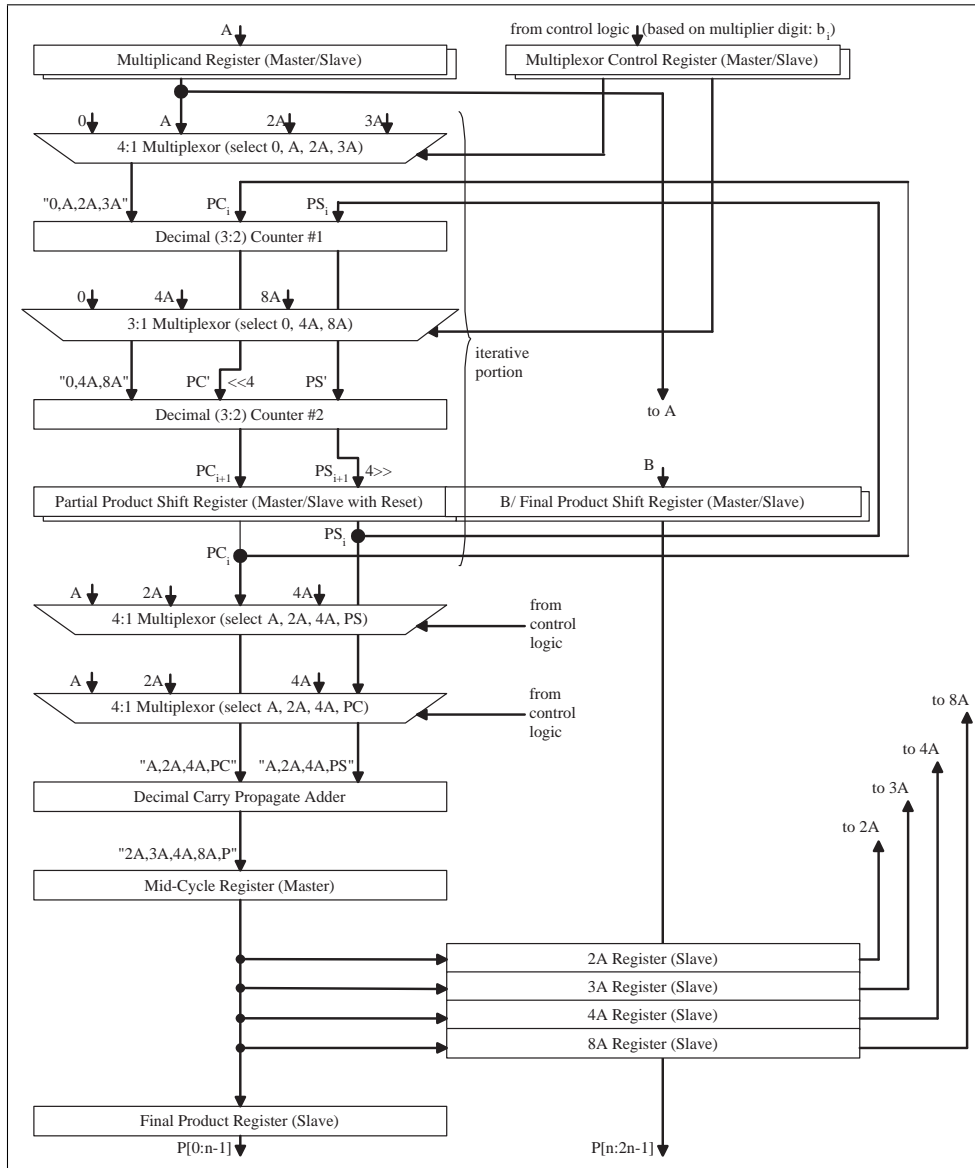


Figure 4.1: Preliminary Iterative DFXP Multiplier Design

instead of multiplexing on the data port of each latch to select between new data or existing data, the clocks controlling the latches are selectively enabled or disabled. In addition to reducing the delay in the dataflow portions of the design, this strategy also reduces the power consumption.

The design of Figure 4.1 has two frequency limiting issues. First, the two decimal

3:2 counters in series, along with the multiplexors needed to steer the appropriate multiples into the decimal counters, contain too much logic for a single fast cycle. Unfortunately, simply splitting the logic into two pipeline stages is undesirable as it doubles the number of cycles needed for the iterative portion of the algorithm. Second, the decimal carry-propagate adder used to generate the multiples and the final product is too slow as well. Splitting this logic into two pipes is reasonable in this case as the overall latency only increases by three cycles. This is because the four secondary multiples can be produced in six cycles (two additional cycles) and the final product produced in two cycles (one additional cycle).

In the remaining subsections, a finalized multiplier design employing decimal CSAs is presented which addresses these frequency-limiting issues. The design is an iterative DFXP multiplier which extends previous techniques used for decimal multiplication including generating a reduced set of multiplicand multiples [105], using carry-save addition for the iterative portion of the multiplier [124,130], and using direct decimal addition [115] to implement decimal carry-save addition. Novel features of the multiplier design include the use of decimal counters and compressors, fast generation of multiplicand multiples that do not need to be stored, and a simplified decimal carry-propagate addition to produce the final product. Referring to Figure 3.6 on page 63, this multiplier design generates a secondary set of multiplicand multiples¹ (Figure 3.6.b), iteratively accumulates the partial products (Figure 3.6.c), and then removes the redundancy in the intermediate product emerging from the partial product reduction tree (Figure 3.6.e) to yield the final product.

¹See Section 3.3.

Algorithm

A flowchart-style drawing of the algorithm described in [39] appears in Figure 4.2. After reading in the operands, the following multiplicand multiples are generated: $2A$, $4A$, and $5A$. The $2A$ and $5A$ multiples are created directly, without carry propagation, and the $4A$ is created by doubling the $2A$ multiple. The creation of the secondary multiples occurs in the block labeled “Generate Multiplicand Multiples” in Figure 4.2. Once the multiplicand multiples are developed, the portion of the algorithm which iterates over the multiplier operand digits begins, as shown in the block labeled “Iterative Portion” in Figure 4.2. This portion includes selecting the secondary multiplicand multiples, creating the partial product in sum and carry form, adding the partial product to the intermediate product in sum and carry form, shifting the intermediate product one digit to the right, and producing a final product digit. After all the partial products have been accumulated, an addition is performed to produce a non-redundant final product (shown in the “Add to Yield Final Product” block).

Features

As mentioned, the carry-propagate adder of Figure 4.1 used to generate the multiples and the final product is too slow for a high-frequency design. One alternative is to generate the multiples using carry-save addition. The benefit of this alternative is that, although the four secondary multiples are still generated in four cycles, the removal of the carry propagation allows a much higher frequency of operation. The drawback of this approach is that the decimal counter that implements Equation 4.3 becomes a decimal 4:2 compressor and additional registers are needed to store carry bits. A better alternative is to generate only the $2A$, $4A$, and $8A$ multiples, for the

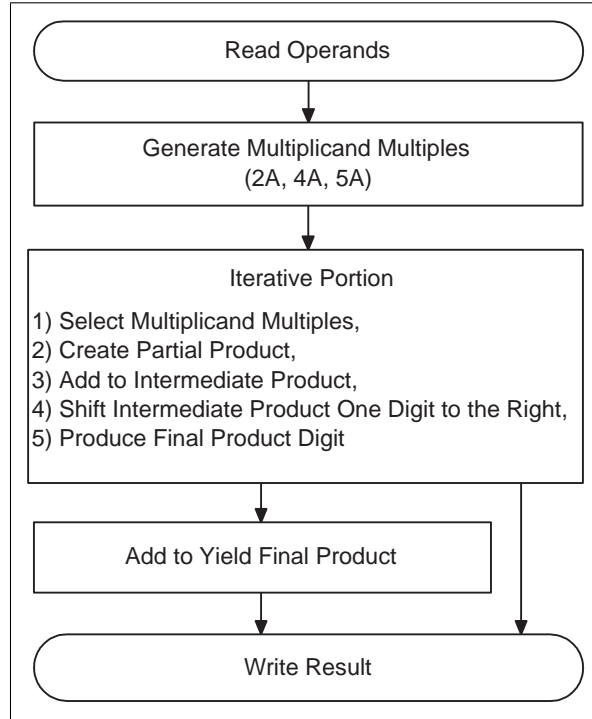


Figure 4.2: Flowchart of Iterative DFXP Multiplier Using Decimal CSAs

following reason.

As mentioned in Section 3.3, two properties are needed for the generation of multiplicand multiples to be digit-wise independent. Both doubling and quintupling satisfy these requirements. So, in three gate delays, the $2A$ can be developed, in three more gate delays, the $4A$ can be developed, and in three more gate delays, the $8A$ can be developed. Depending on the technology, the desired frequency, and available area, the designer can develop these multiples in one cycle (with three doubling circuits) or in three cycles (with a single doubling circuit). Unfortunately, A , $2A$, $4A$, and $8A$ are insufficient to generate all of the primary multiples through the addition of only two multiples. A complete secondary set, however, can be formed by the inclusion of the $-A$ or $5A$ multiple.

To reduce the overall delay of this algorithm, the delay of the additions in the iter-

ative portion must be decreased. As mentioned in Section 3.3.2, the partial products can be created from a secondary set of multiples and quickly accumulated through the use of decimal CSAs. In a similar manner, decimal carry-save addition can be used for partial product reduction. That is, by reordering the addends in Equations 4.1 and 4.2, the secondary multiples can be added, followed by addition of the partial product sum and carry with the previous iteration’s intermediate product sum and carry to produce the new intermediate product sum and carry. To account for the increased order of each partial product to be added, the previous iteration’s intermediate product is right-shifted one digit position. Since the addition of the secondary multiples does not depend on the previous partial product or the intermediate product, the aforementioned reordering allows the decimal carry-save additions to be split into two stages and operate at a much higher frequency.

Using this approach, Equations 4.1 and 4.2 can be rewritten as:

$$(TS_i, TC_i) = A \cdot b'_i + A \cdot b''_i \quad (4.3)$$

$$(PS_{i+1}, PC_{i+1}) = (PS_i + PC_i + TS_i + TC_i) \cdot 10^{-1} \quad (4.4)$$

where TS_i and TC_i comprise the partial product in sum and carry format (i.e., $A \times b_i$) and PS_{i+1} and PC_{i+1} comprise the intermediate product for iteration i . The multiplier operand digit, b_i , is represented by b'_i and b''_i , where $b_i = b'_i + b''_i$, as two secondary multiples are used to produce the desired partial product. For example, if $b_i = 9$, $b'_i = 5$ and $b''_i = 4$ (further details follow). Note the tuple sum, TS_i , and the intermediate product sum, PS_i , are four-bit BCD digits, while the tuple carry, TC_i , and intermediate product carry, PC_i , are single bits. Only the first pass through the decimal CSA in Equation 4.3 contributes to the overall latency since the partial product is never fed back through this counter. When implementing Equation 4.3, a

simplified decimal CSA can be used, since only two decimal digits are added together (i.e., there are no carry bits associated with the secondary multiples). Although this optimization does not reduce the critical path delay, it does reduce the area slightly.

Each digit summation in Equation 4.4 can have a maximum value of 20, as each sum digit can have a maximum value of nine and there are two carry bits. A decimal 4:2 compressor, analogous to a binary 4:2 compressor, was designed to handle this addition. A decimal 4:2 compressor accepts as inputs two, four-bit BCD digits, x_i and y_i , and two, one-bit carry-ins, $c_i[0]$ and $c'_i[0]$, and produces a four-bit BCD sum digit, s_i , and a one-bit carry-out $c_{i+1}[0]$. The decimal 4:2 compressor uses a standard decimal CSA to compute

$$(c''_{i+1}[0], s'_i) = x_i + y_i + c_i[0] \quad (4.5)$$

where $c''_{i+1}[0]$ is an intermediate carry-out and s'_i is an intermediate sum. A simplified decimal 3:2 counter is then used to compute the final sum and carry as:

$$(c_{i+1}[0], s_i) = s'_i + c'_i[0] + c''_i[0] \quad (4.6)$$

The multiples $1A$, $2A$, $4A$, and $5A$ are chosen as the secondary multiple set, as they can be generated quickly since with both doubling and quintupling of BCD numbers there is no carry propagation beyond the next more significant digit [105]. See Section 3.3.2 for a description of doubling and quintupling and [39] for the Boolean equations for generating $2A$ and $5A$.

In summary, in three gate delays, $2A$ can be generated from A , in three more gate delays $4A$ can be generated from $2A$, and in parallel $5A$ can be generated from A . Thus, after six gate delays, which can easily fit into one cycle, all the secondary

Table 4.1: Generation of Primary Multiples from A , $2A$, $4A$, and $5A$

Multiplier Digit (decimal value of b_i)	Secondary Multiples		
	b'_i	+	b''_i
0	0	+	0
1	1	+	0
2	0	+	2
3	1	+	2
4	4	+	0
5	5	+	0
6	4	+	2
7	5	+	2
8	4	+	4
9	5	+	4

multiples are ready. Further, since each of the secondary multiples is generated from A using dedicated logic, and the value of A does not change throughout the iterations, it is not necessary to store any of the multiples of A in registers. Table 4.1 shows how all the primary multiples can be generated from the secondary multiples: A , $2A$, $4A$, and $5A$.

Referring to Equation 4.3 and Table 4.1, a 4:1 multiplexor function is needed to select $A \cdot b'_i$ and a 3:1 multiplexor function is needed to select $A \cdot b''_i$ based on the value of the current multiplier digit. The controls for these multiplexors, shown below for a one-hot encoding, are generated and latched one cycle before their use. This does not affect the latency of the multiplication, since the controls are generated in parallel with the multiples of A .

Left Tuple or $A \cdot b'_i$ from Equation 4.3

$$\begin{aligned}
\text{“select 0”} &= \overline{b_i[0]} \wedge \overline{b_i[1]} \wedge \overline{b_i[3]} \\
\text{“select A”} &= \overline{b_i[0]} \wedge \overline{b_i[1]} \wedge b_i[3] \\
\text{“select 4A”} &= (b_i[0] \vee b_i[1]) \wedge \overline{b_i[3]} \\
\text{“select 5A”} &= (b_i[0] \vee b_i[1]) \wedge b_i[3]
\end{aligned}$$

Right Tuple or $A \cdot b_i''$ from Equation 4.3

$$\begin{aligned}
\text{“select 0”} &= \overline{b_i[0]} \wedge \overline{b_i[2]} \\
\text{“select 2A”} &= b_i[2] \\
\text{“select 4A”} &= b_i[0]
\end{aligned}$$

Here, \wedge represents logical AND while \vee represents logical OR (see Table B.5). These equations are simplified based on the fact the BCD sum digits only have values in the range $\{0, \dots, 9\}$. Note that if static logic is used for the multiplexors, and no select signals are active, then the multiplexor will source a logic 0 value. In this case, the selection of 0 occurs by default and only 3:1 and 2:1 multiplexors are needed to select the two multiplicand multiples needed to realize Equation 4.3.

Another way decimal carry-save addition benefits decimal multiplication is in removing the redundancy in the intermediate product after all the partial products have been accumulated. Because the intermediate product is represented by a four-bit BCD sum and a one-bit carry, the final adder used to yield a non-redundant product is less complex and faster than one which must accommodate two, four-bit sums in each digit position. The digit generate signal only occurs when the sum equals nine and the carry equals one (a three input AND), and the digit propagate only occurs when the sum equals eight and the carry equals one or when the sum equals nine. Thus, the equations for the digit generate and digit propagate in the j^{th} digit position are as

follows:

$$\begin{aligned}g_j &= ps_j[0] \wedge ps_j[3] \wedge pc_j \\p_j &= (ps_j[0] \wedge pc_j) \vee (ps_j[0] \wedge ps_j[3]) \\ &= ps_j[0] \wedge (ps_j[3] \vee pc_j)\end{aligned}$$

Note these equations are simplified based on the fact the BCD sum digits only have values in the range $\{0, \dots, 9\}$.

Implementation and Analysis

The iterative DFXP multiplier design, which includes the features just described, is shown in Figure 4.3. The multiplier takes one cycle to generate the secondary multiples (A , $2A$, $4A$, and $5A$), one cycle to produce the first partial product to be added, one cycle per each multiplier digit to accumulate all the partial products, and two cycles to produce the final product. Thus, it has a worst-case latency of $n + 4$ cycles and can initiate successive multiplications of $n + 1$ cycles (i.e., its initiation interval is every $n + 1$ cycles). This design scales well to larger operand sizes, since increasing the operand size only affects² the number of iterations and the delay of the final carry-propagate adder, which can be further pipelined.

A register transfer level model of the presented multiplier supporting 64-bit (16-digit) operands was coded in Verilog. The multiplier design was synthesized using LSI Logic's gflxp 0.11um CMOS standard cell library and the Synopsys Design Compiler. Table 4.2 contains area and delay estimates for the multiplier design presented. The values in the *FO4 Delay* column are based on the delay of an inverter driving four same-size inverters being $55ps$ in the aforementioned technology.

²There are also electrical affects such as increased fan-out and loading.

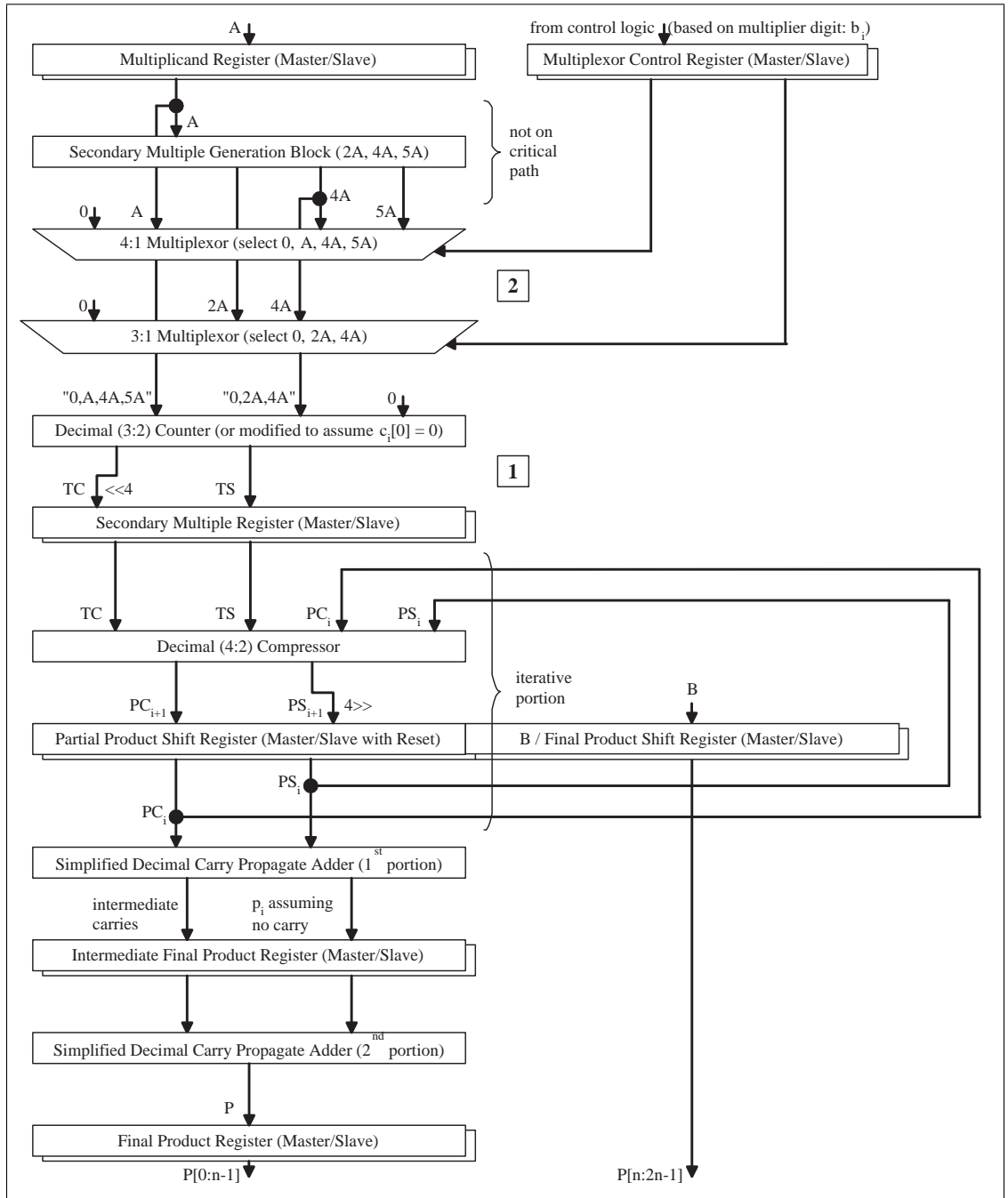


Figure 4.3: Iterative DFXP Multiplier Design Using Decimal CSAs

Note there are a number of data-dependent optimizations that can be applied to the described multiplier. These include: skipping zeros in the multiplier operand,

Table 4.2: Area and Delay of Iterative DFXP Multiplier (Decimal CSAs)

$p = 16$	Iterative (Decimal CSAs) [39]
Latency (cycles)	20
Throughput (ops/cycle)	1/17
Cell count	59,234
Area (μm^2)	119,653
Delay (ps)	810
Delay ($FO4$)	14.7

effectively performing two digit multiplications at the same time for a subset of two-digit strings in the multiplier operand, and exiting early after the handling of the leading nonzero multiplier operand digit. The early exit strategy is the only optimization of the three that does not add delay to the dataflow portion of the design. Although skipping zeros and double-digit multiplication will add delay, the delay is not introduced in the critical iterative portion of the design. Details of these optimization strategies can be found in [39].

Summary

An iterative DFXP multiplier utilizing decimal CSAs for the multiplicand multiple generation and partial product accumulation was presented. This design can support a high operating frequency, scales well, has a worst-case latency of $n + 4$ cycles, and an initiation interval of $n + 1$ cycles, where n is the number of significant digits in the multiplier operand. Additionally, several data-dependent optimizations can be implemented.

Follow-on research was performed to reduce the cycle time delay of this multiplier

algorithm by replacing the direct decimal addition scheme used in the 4:2 compressor with a binary addition and correction scheme using the *overloaded decimal representation* (see Section 3.2.2). This modified iterative DFXP multiplier, published in [113], exhibited a 12% reduction in cycle time (14% improvement in clock frequency) at the expense of four additional cycles of latency and a 78% increase in area. The additional cycles are required to remove all the redundancy in the product. In the next subsection, another iterative DFXP multiplier is presented which utilizes a recoding of the operands and signed-digit adders.

4.1.2 Multiplier Employing Signed-Digit Adders

In the DFXP design of the last subsection, a secondary set of multiplicand multiples was used to produce the partial products (1A to 9A). In contrast, the design presented in this subsection uses the novel approach of restricting the range of the operand digits such that only the 1A to 5A multiples are needed. Restricting the range of the operand digits leads to a faster generation of a smaller number of unique partial products. A second significant and novel difference is that the partial products are not created from stored multiplicand multiples, rather, they are created on-the-fly through digit-by-digit multiplier logic blocks. Referring to Figure 3.6 on page 63, this multiplier design recodes the multiplicand and the multiplier operand (Figure 3.6.a), generates the partial products in an iterative manner on-the-fly (Figure 3.6.b), accumulates the partial products iteratively (Figure 3.6.c), and then removes the redundancy in the final intermediate product while converting the signed-digits to BCD (Figure 3.6.d and 3.6.e) to yield the final product.

Algorithm

A flowchart-style drawing of the algorithm described in [40] appears in Figure 4.4. After reading in the operands, both the multiplier and multiplicand operands are recoded into the range of $\{-5, \dots, +5\}$. The recoding of the operands occurs in the block labeled “Recode Multiplicand and Multiplier” in Figure 4.2. Once the operands are recoded, the portion of the algorithm which iterates over the multiplier operand digits begins, as shown in the block labeled “Iterative Portion” in Figure 4.2. This portion includes creating the partial product in a redundant form, converting the partial product to signed-digits, adding the partial product to the intermediate product in a redundant form, shifting the intermediate product one digit to the right, and producing a final product digit. After all the partial products have been accumulated, an addition is performed to produce a final product comprised of positive digits (shown in the “Add to Yield Final Product” block). A final step is needed to convert each digit of the final product to BCD. This occurs in the block labeled “Convert to BCD” in Figure 4.2.

Features

Using signed-digits, the range of all digit positions in a number can be restricted by replacing each digit whose value exceeds the desired range with the additive inverse of its radix complement and incrementing its next more significant digit [119]. Figure 4.5 provides an example of this recoding technique resulting in digits within the range of $\{-5, \dots, +5\}$. Each digit greater than or equal to five is recoded by subtracting ten and incrementing the next more significant digit. This is explained later in more detail. (The reader is referred to [136] for further reading on signed-digit numbers.)

The range $\{-5, \dots, +5\}$ was chosen as it is close to the minimum of ten

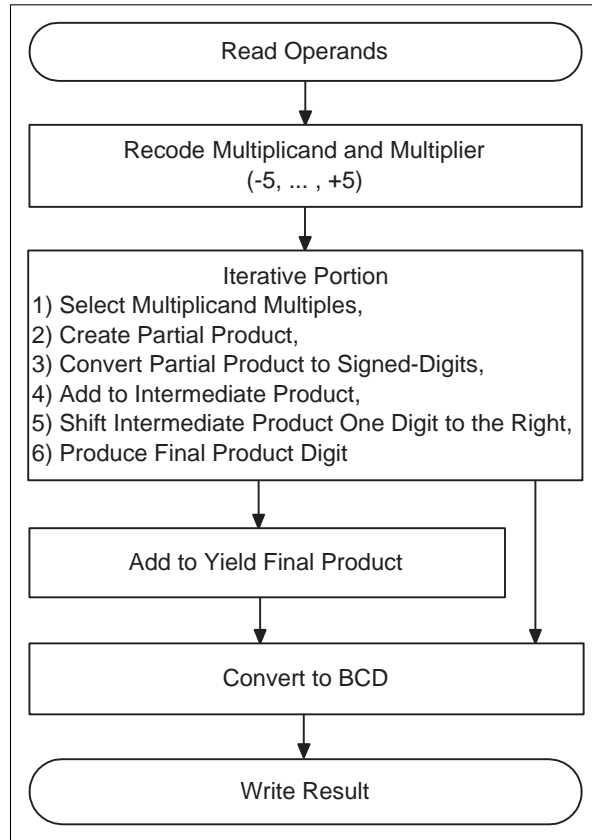


Figure 4.4: Flowchart of Iterative DFXP Multiplier Using Signed-Digit Adders

$$9837 = (1 \cdot 10^4) + (0 \cdot 10^3) + (-2 \cdot 10^2) + (4 \cdot 10^1) + (-3 \cdot 10^0) = 10\bar{2}4\bar{3}$$

Figure 4.5: Example of Recoding into Signed Decimal Digits

representable values needed for decimal digits which means fewer unique digit-by-digit products [122]. That is, since the magnitude of a digit-by-digit product is independent of the sign of the multiplicand and multiplier inputs, and this recoding range is balanced, this choice significantly reduces the combinations of inputs needing to be multiplied. Table 4.3 shows the reduction in both input combinations and complexity achievable by restricting the range of inputs for which digit-by-digit products must

Table 4.3: Complexity of Digit-by-Digit Products for Ranges of Decimal Inputs

range of inputs	total products	unique products	total minterms [†]	maximum minterms per output [†]	maximum gate levels per output [‡]
$[0 - 9] \times [0 - 9]$	100	37	62	23	19
$[1 - 9] \times [1 - 9]$	81	36	61	21	17
$[2 - 9] \times [2 - 9]$	64	30	55	20	16
$[0 - 5] \times [0 - 5]$	36	15	20	7	8
$[1 - 5] \times [1 - 5]$	25	14	20	7	7
$[2 - 5] \times [2 - 5]$	16	10	15	5	6

[†] Espresso results in sum-of-products form

[‡] SIS results with library of INV, NAND2, NAND3, NAND4, NOR2, NOR3, AOI21, AOI22, XOR, and XNOR cells

be generated. As shown, with a digit range of $\{0, \dots, 9\}$, there are a total of 100 input combinations which can result in 37 unique products. Computing this product set via a combinatorial digit-by-digit multiplier block requires 62 minterms, with the worst-case output bit using 23 minterms or 19 gate levels, as determined by SIS [137]. The proposed range, $\{-5, \dots, +5\}$, leads to digit-by-digit products of $[0-5] \times [0-5]$, which requires 7 minterms or 8 gate levels for its worst-case output bit. By taking advantage of the facts that if either of the operand digits is zero, the digit-by-digit product is zero or if either of the operand digits is one, the digit-by-digit product is the other operand's digit, only 16 combinations are possible which requires only 5 minterms or 6 gate levels for the worst-case output bit. This is shown in the last line of Table 4.3.

To achieve the restricted range of $\{-5, \dots, +5\}$, one might assume that each digit greater than or equal to six must be recoded by subtracting ten and incrementing

line	cycle	function (<i>line # or "value"</i>)	example §
1	0	latch multiplicand	3 3 9
2		latch multiplier	2 6 5
3	1	recode multiplicand (1)	3 4 $\bar{1}$
4		recode multiplier digit [2] (2)	$\bar{5}$
5		generate partial product	$\bar{5}$ $\bar{0}$ 5
6		in overlapped form (3,4)	$\bar{1}$ $\bar{2}$ 0
7	2	convert partial product to	$\bar{2}$ 3 0 5
8		non-overlapped form (5,6)	
9		recode multiplier digit [1] (2)	$\bar{3}$
10		generate partial product	1 $\bar{2}$ 3
11		in overlapped form (3,9)	$\bar{1}$ $\bar{1}$ $\bar{0}$
12	3	add partial product (7) to	$\bar{2}$ 3 0 5
13		intermediate product ("0");	
14		convert partial product to	$\bar{1}$ 0 $\bar{2}$ 3
15		non-overlapped form (10,11)	
16		recode multiplier digit [0] (2)	3
17		generate partial product	$\bar{1}$ 2 $\bar{3}$
18		in overlapped form (3,16)	1 1 $\bar{0}$
19	4	convert LSD of intermediate	$\bar{5}$
20		product (12); transfer out	0
21		add partial product (14) to	$\bar{1}$ $\bar{2}$ 1 3
22		intermediate product (12,20)	
23		convert partial product to	1 0 2 $\bar{3}$
24		non-overlapped form (17,18)	
25	5	convert LSD of intermediate	$\bar{3}$
26		product (21); transfer out	0
27		add partial product (23) to	1 $\bar{1}$ 0 $\bar{2}$
28		intermediate product (21,26)	
29	6	first half of conversion	
30		to BCD (27)	
31		convert LSD of intermediate	$\bar{8}$
32		product (27); transfer out	$\bar{1}$
33	7	second half of conversion	$\underline{\underline{0}}$ $\underline{\underline{8}}$ $\underline{\underline{9}}$
34		to BCD (27,32)	

§ Digits in the final product are double underlined.

Figure 4.6: Example for Iterative DFXP Multiplier Using Signed-Digit Adders

the next more significant digit. However, since a digit can be incremented due to the value of the next less significant digit, the chosen strategy is to evaluate and recode all digits greater than or equal to *five*. By doing so, the recoding of the digits can occur in parallel as an increment of the next more significant digit will never propagate. Although it is not always necessary to recode a digit that has a value of five, the chosen approach decreases hardware as only one condition, greater than or equal to five, must be evaluated for each digit position. Figure 4.6, referred to throughout this subsection, provides two examples of three-digit numbers recoded in the range of $\{-5, \dots, +5\}$. The number on line 1 (339) is recoded into the number on line 3 ($34\bar{1}$), and the number on line 2 (265) is recoded into $3\bar{3}\bar{5}$, the digits of which can be found on lines 16, 9, and 4, respectively.

To restrict the range of the operand digits to be $\{-5, \dots, +5\}$, the multiplicand is sent to a set of n recoders, where n is the number of digits of the operands, and each multiplier digit is sent to a single recoder, as it is being used. Each recoder block receives as input one four-bit BCD operand digit, a_i , and a single bit, $ge5_{i-1}$, indicating if the next less significant digit is greater than or equal to five and produces as output a four-bit signed-magnitude digit, a_i^S , and a single bit, $ge5_i$, indicating whether the current digit is greater than or equal to five. The superscript S indicates the result of the recoding is a signed-magnitude digit.

Figure 4.7 shows a block diagram of a recoder, and Equation 4.7 describes its function as a collection of sub-functions selected by specific classifications of the input data. Although the equations in this section are shown based on digits of the multiplicand operand A , the same equations are applicable to the digits of multiplier operand B . The last three sub-functions are increment, complement, and increment & complement, respectively, hence the superscripts. The circuit implementations for each of these sub-functions are simplified based on the limited range of their

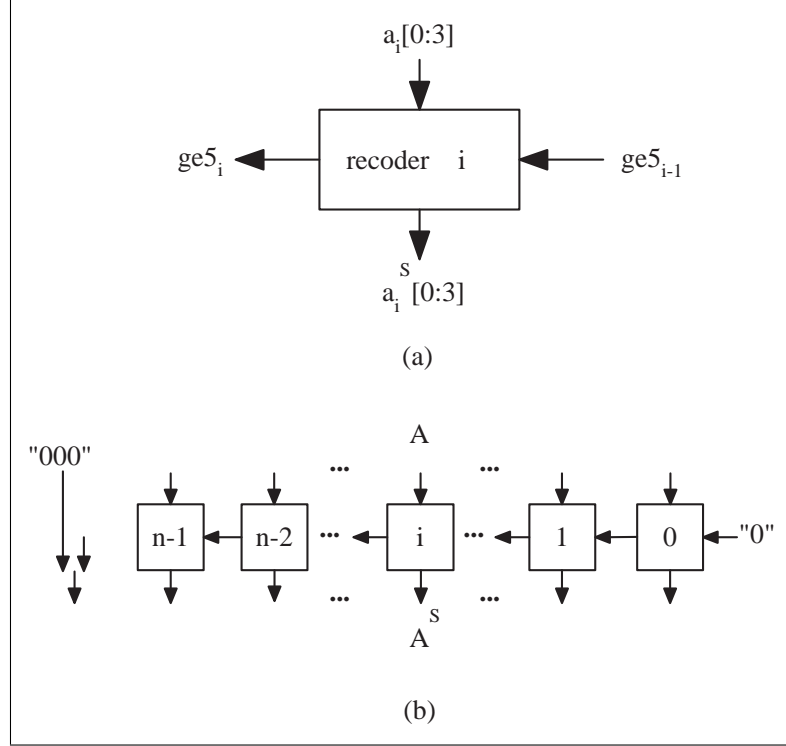


Figure 4.7: Recoder Block: (a) Single Digit, (b) n-Digit Operand

inputs. That is, increment only occurs on values zero through four ($a_i < 5$), and both complement and increment & complement only occur on values five through nine ($a_i \geq 5$).

$$a_i^S = \begin{cases} a_i & \text{if } a_i < 5 \ \& \ a_{i-1} < 5 \\ a_i^I = a_i + 1 & \text{if } a_i < 5 \ \& \ a_{i-1} \geq 5 \\ a_i^C = -(10 - a_i) & \text{if } a_i \geq 5 \ \& \ a_{i-1} < 5 \\ a_i^{IC} = -(9 - a_i) & \text{if } a_i \geq 5 \ \& \ a_{i-1} \geq 5 \end{cases} \quad (4.7)$$

The following sets of equations describe the logic of these three sub-functions. In each four-bit signed-magnitude digit, bit [0] represents the sign, and bits [0:2] represent the magnitude. Only Equations 4.8 through 4.14 are unique and require circuitry. The different forms of the operand digit, along with the unaltered operand

digit are input to multiplexor logic that selects the correct digit based on $ge5_i$ and $ge5_{i-1}$ (Equation 4.14).

$$\begin{aligned} a_i^I[0] &= 0 \\ a_i^I[1] &= \overline{\overline{a_i[1] \wedge (a_i[2] \vee a_i[3])}} \end{aligned} \quad (4.8)$$

$$a_i^I[2] = a_i[2] \oplus a_i[3] \quad (4.9)$$

$$a_i^I[3] = \overline{a_i[3]} \quad (4.10)$$

$$\begin{aligned} a_i^C[0] &= 1 \\ a_i^C[1] &= \overline{\overline{a_i[1] \vee (a_i[2] \wedge a_i[3])}} \end{aligned} \quad (4.11)$$

$$a_i^C[2] = \overline{a_i[2] \oplus a_i[3]} \quad (4.12)$$

$$a_i^C[3] = a_i[3]$$

$$\begin{aligned} a_i^{IC}[0] &= 1 \\ a_i^{IC}[1] &= \overline{a_i[0] \vee a_i[2]} \end{aligned} \quad (4.13)$$

$$a_i^{IC}[2] = a_i[2]$$

$$a_i^{IC}[3] = \overline{a_i[3]}$$

$$ge5_i = \overline{\overline{a_i[0] \wedge a_i[1] \wedge (a_i[2] \vee a_i[3])}} \quad (4.14)$$

In the case of recoding the multiplicand operand A , the n^{th} digit needs to be set to 1 if the MSD is greater than or equal to five (i.e., when $ge5_{n-1}$ is high). This means there will be one additional partial product, which can be realized by concatenating $ge5_{n-1}$ with three leading zeros. The recoded multiplicand operand A^S and a digit from the recoded multiplier operand, b_i^S , are input to digit multiplier blocks as described in the next subsection to generate a partial product P_i^O in overlapped

form.

Word-by-Digit Partial Product Generation To reduce the area and delay of generating partial products, the range of the input digits for which digit-by-digit products must be generated is restricted in three ways. The first restriction sets an upper bound on the input digits by recoding the operands into signed-magnitude digits with a range of $\{-5, \dots, +5\}$. The second restriction sets a limit on the possible input digit combinations by applying the principle that the absolute value of a product is independent of the sign of the input digits. The third restriction sets a lower bound on the input digits by applying the observation that if either digit is zero, the product is zero, and if either digit is one, the product is the other digit.

With these three restrictions on the input digits, the range is reduced to only 2 through 5 when computing a product. Thus, only 16 combinations of the inputs are possible resulting in ten different products with a range of $\{4, \dots, 25\}$. With existing schemes, the range of digits is $\{0, \dots, 9\}$, which yields 100 possible combinations of the two inputs³. (Table 4.3 illustrates for various input ranges the significant reduction in complexity achievable by restricting the number of input combinations.)

To generate a partial product on a word-by-digit basis, the recoded multiplicand and a recoded digit from the multiplier are input to $n + 1$ digit multiplier blocks (see Figure 4.8b). Note since the n^{th} digit of the recoded multiplicand has at most a magnitude of 1, the digit multiplier block in this position can be replaced with a simpler circuit to produce either 0 or the recoded multiplier digit, $|b_i^S|$. Each multiplier block receives as input two, four-bit, signed-magnitude digits, a_i^S and b_i^S , and produces as output two, signed-magnitude partial product digits, p_{i+1}^O and p_i^O . The superscript O indicates the partial product is in an overlapped form since each digit multiplier

³If zero and one are removed from the range in a similar manner, there are 64 combinations

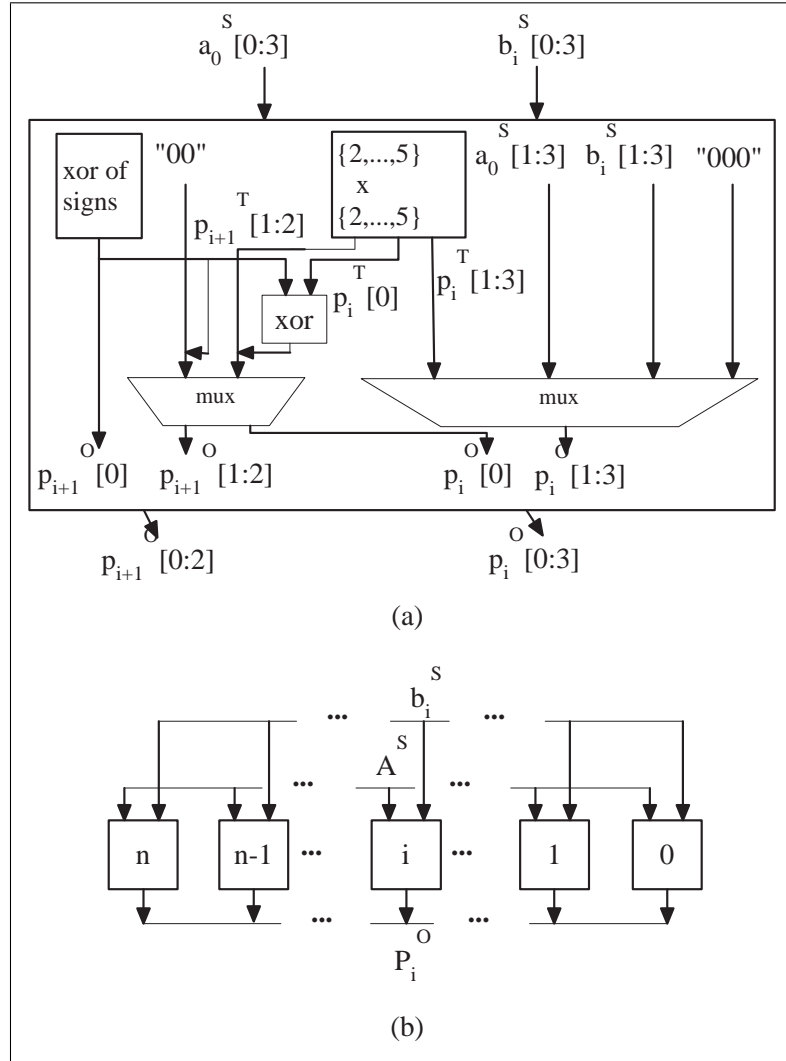


Figure 4.8: Digit Multiplier Block: (a) Single Digit, (b) n-Digit

block yields two digits. Equation 4.15 describes the function to generate a digit-by-digit product, in absolute-value form, as a collection of sub-functions selected by specific classifications of the input digits. The superscript T indicates the sub-function output is realized via a lookup table or a combinational circuit structure.

To simplify the removal of the overlap in the partial product, the range of $|p_i^T|$ is restricted to $\{0, \dots, 5\}$ by again using signed-magnitude digits. With this restriction, which matches the inherent restriction on the other sub-functions in Equation 4.15,

four bits are needed for the product's LSD (range of $\{-4, \dots, +5\}$), and two bits are needed for the product's MSD (range of $\{0, 1, 2\}$).

$$|p_{i+1}^O, p_i^O| = \begin{cases} 00,0000 & \text{if } |a_i^S| = 0 \text{ or } |b_i^S| = 0 \\ 00,0b_i^S[1:3] & \text{if } |a_i^S| = 1 \text{ \& } |b_i^S| > 0 \\ 00,0a_i^S[1:3] & \text{if } |a_i^S| > 1 \text{ \& } |b_i^S| = 1 \\ |p_{i+1}^T, p_i^T| & \text{if } |a_i^S| > 1 \text{ \& } |b_i^S| > 1 \end{cases} \quad (4.15)$$

Table 4.4 shows for inputs ranging from 2 to 5 the two-digit, signed-magnitude products conforming to this magnitude restriction. Although the LSD has a negative sign in some instances, the MSD is always positive, and thus the two-digit product is a positive value. Figure 4.8a shows the block diagram of a digit multiplier block, and Equations 4.16 - 4.21 show how the two-digit products are developed.

Table 4.4: Restricted-Range, Signed-Magnitude Products

x	2	3	4	5
2	04 ₁₀	1 $\bar{4}$ ₁₀	1 $\bar{2}$ ₁₀	10 ₁₀
	00,0100 ₂	01,1100 ₂	01,1010 ₂	01,0000 ₂
3	1 $\bar{4}$ ₁₀	1 $\bar{1}$ ₁₀	12 ₁₀	15 ₁₀
	01,1100 ₂	01,1001 ₂	01,0010 ₂	01,0101 ₂
4	1 $\bar{2}$ ₁₀	12 ₁₀	2 $\bar{4}$ ₁₀	20 ₁₀
	01,1010 ₂	01,0010 ₂	10,1100 ₂	10,0000 ₂
5	10 ₁₀	15 ₁₀	20 ₁₀	25 ₁₀
	01,0000 ₂	01,0101 ₂	10,0000 ₂	10,0101 ₂

Since the signs of the recoded operand digits were not considered when generating the digit-by-digit products, the partial product at this point is in absolute-value form. Thus, the sign of the recoded operand digits must be used to convert $|P_i^O|$ into a

$$|p_i^T[3]| = \overline{\overline{a_i^S[3]} \vee \overline{b_i^S[3]}} \quad (4.16)$$

$$|p_i^T[2]| = \overline{\overline{a_i^S[2] \wedge \overline{b_i^S[2]} \wedge \overline{b_i^S[3]}} \wedge} \quad (4.17)$$

$$|p_i^T[1]| = \overline{\overline{a_i^S[2] \wedge \overline{a_i^S[3]} \wedge \overline{b_i^S[2]}} \wedge} \quad (4.18)$$

$$|p_i^T[0]| = \overline{\overline{a_i^S[3] \wedge b_i^S[1] \wedge b_i^S[3]} \wedge} \quad (4.19)$$

$$|p_i^T[0]| = \overline{\overline{(a_i^S[2] \oplus a_i^S[3]) \wedge b_i^S[2]} \wedge \overline{b_i^S[3]}} \wedge} \quad (4.19)$$

$$|p_i^T[0]| = \overline{\overline{a_i^S[2] \wedge b_i^S[2] \wedge \overline{b_i^S[3]}} \wedge} \quad (4.19)$$

$$|p_i^T[0]| = \overline{\overline{a_i^S[3] \wedge \overline{b_i^S[2]} \wedge \overline{b_i^S[3]}} \wedge} \quad (4.19)$$

$$|p_i^T[0]| = \overline{\overline{a_i^S[2] \wedge b_i^S[2] \wedge b_i^S[3]} \wedge} \quad (4.19)$$

$$|p_i^T[0]| = \overline{\overline{a_i^S[2] \oplus a_i^S[3] \wedge b_i^S[2] \wedge \overline{b_i^S[3]}} \wedge} \quad (4.19)$$

$$|p_{i+1}^T[1]| = \overline{\overline{a_i^S[2] \wedge b_i^S[1]} \wedge} \quad (4.20)$$

$$|p_{i+1}^T[1]| = \overline{\overline{(a_i^S[1] \vee a_i^S[3]) \wedge b_i^S[2]} \wedge} \quad (4.20)$$

$$|p_{i+1}^T[1]| = \overline{\overline{a_i^S[2] \wedge b_i^S[3]} \wedge} \quad (4.20)$$

$$|p_{i+1}^T[0]| = \overline{\overline{a_i^S[1] \vee b_i^S[1]} \wedge} \quad (4.21)$$

properly signed partial product. This step is necessary before attempting to add the overlapping portions of the word-by-digit products as not doing so could yield an incorrect partial product. To develop a partial product with the correct sign, P_i^O , the exclusive-or (XOR) of the input signs (i.e., $a_i^S[0] \oplus b_i^S[0]$), is used in two places. First, it directly becomes the sign of the product's MSD, $p_{i+1}^O[0]$. Second, it is XORed with the sign of the product's LSD, $|p_i^O[0]|$, to produce $p_i^O[0]$. Figure 4.6, lines 5/6, 10/11, and 17/18, provide examples of the digit multiplier blocks yielding the sign-corrected partial products in overlapped form.

Ultimately, all the partial products need to be properly aligned with respect to one another and added together. The approach chosen in this work is to accumulate iteratively the partial products via the signed-digit adder described by Svoboda

in [4]. Svoboda's adder accepts two uniquely encoded signed-digits (see Table 3.3 in Section 3.1) in the range of $\{-6, \dots, +6\}$ and yields a sum in the same range. Note, with the encoding shown in Table 3.3, the additive inverse is obtained by taking the one's complement.

Recall the partial product at this point is properly signed but still in an overlapped form. Each digit position⁴ has one four-bit, signed-magnitude digit whose range is $\{-5, \dots, +5\}$ and one three-bit, signed-magnitude digit whose range is $\{-2, \dots, +2\}$. The sums for these ranges of overlapping signed-digits, suitable for entry into a Svoboda adder, are in bold type in Table 4.5. In each entry of this table, the digit on the right is a sum digit in position i , and the digit on the left is a transfer digit, which is added to the sum digit in position $i + 1$. The term transfer digit is used to indicate when a carry or a borrow occurs. To achieve the desired encoding, a combinatorial circuit is needed to recode the signed-magnitude digits in the partial product P_i^O into signed-digits (P_i). A straightforward implementation of this recoding step requires ten logic levels, as determined by SIS. The recoded partial product, P_i , is then added to the intermediate product, IP_{i-1} , as described in the next section. Figure 4.6, lines 7, 14, and 23, provide examples of generating a non-overlapped partial product from the sign-corrected partial products in overlapped form.

Accumulation of Partial Products and Generation of Final Product As the recoded multiplier operand is traversed from LSD to MSD, the partial product, P_i , needs to be added to the sum of the previous partial products. The accumulated sum of partial products is termed the intermediate product and is designated IP_i , where the subscript indicates how many partial products have been accumulated. The accumulation occurs in an iterative manner with the intermediate product being shifted

⁴The MSD and LSD only have one digit in their position.

Table 4.5: Restricted-Range, Signed-Digit Sums [4] (All Digits Are Decimal)

+	$\bar{6}$	$\bar{5}$	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$	$\bar{0}$	0	1	2	3	4	5	6
6	0 $\bar{0}$	01	02	03	04	05	1 $\bar{4}$	1 $\bar{4}$	1 $\bar{3}$	1 $\bar{2}$	1 $\bar{1}$	1 $\bar{0}$	11	12
5	0 $\bar{1}$	0 $\bar{0}$	01	02	03	04	05	05	1$\bar{4}$	1$\bar{3}$	1 $\bar{2}$	1 $\bar{1}$	1 $\bar{0}$	11
4	0 $\bar{2}$	0 $\bar{1}$	0 $\bar{0}$	01	02	03	04	04	05	1$\bar{4}$	1 $\bar{3}$	1 $\bar{2}$	1 $\bar{1}$	1 $\bar{0}$
3	0 $\bar{3}$	0 $\bar{2}$	0 $\bar{1}$	0 $\bar{0}$	01	02	03	03	04	05	1 $\bar{4}$	1 $\bar{3}$	1 $\bar{2}$	1 $\bar{1}$
2	0 $\bar{4}$	0 $\bar{3}$	0 $\bar{2}$	0 $\bar{1}$	0$\bar{0}$	01	02	02	03	04	05	1 $\bar{4}$	1 $\bar{3}$	1 $\bar{2}$
1	1 $\bar{5}$	0 $\bar{4}$	0 $\bar{3}$	0 $\bar{2}$	0$\bar{1}$	0$\bar{0}$	01	01	02	03	04	05	1 $\bar{4}$	1 $\bar{3}$
0	1 $\bar{4}$	0 $\bar{5}$	0 $\bar{4}$	0 $\bar{3}$	0$\bar{2}$	0$\bar{1}$	0$\bar{0}$	00	01	02	03	04	05	1 $\bar{4}$
0 $\bar{0}$	1 $\bar{4}$	1 $\bar{5}$	0 $\bar{4}$	0 $\bar{3}$	0$\bar{2}$	0$\bar{1}$	0$\bar{0}$	0$\bar{0}$	01	02	03	04	05	1 $\bar{4}$
1 $\bar{1}$	1 $\bar{3}$	1 $\bar{4}$	1 $\bar{5}$	0 $\bar{4}$	0$\bar{3}$	0$\bar{2}$	0$\bar{1}$	0$\bar{1}$	0$\bar{0}$	01	02	03	04	05
2 $\bar{2}$	1 $\bar{2}$	1 $\bar{3}$	1 $\bar{4}$	1 $\bar{5}$	0$\bar{4}$	0$\bar{3}$	0$\bar{2}$	0$\bar{2}$	0$\bar{1}$	0$\bar{0}$	01	02	03	04
3 $\bar{3}$	1 $\bar{1}$	1 $\bar{2}$	1 $\bar{3}$	1 $\bar{4}$	1$\bar{5}$	0$\bar{4}$	0$\bar{3}$	0$\bar{3}$	0$\bar{2}$	0$\bar{1}$	0 $\bar{0}$	01	02	03
4 $\bar{4}$	1 $\bar{0}$	1 $\bar{1}$	1 $\bar{2}$	1 $\bar{3}$	1$\bar{4}$	1$\bar{5}$	0$\bar{4}$	0$\bar{4}$	0$\bar{3}$	0$\bar{2}$	0 $\bar{1}$	0 $\bar{0}$	01	02
5 $\bar{5}$	1 $\bar{1}$	1 $\bar{0}$	1 $\bar{1}$	1 $\bar{2}$	1$\bar{3}$	1$\bar{4}$	1$\bar{5}$	1$\bar{5}$	0$\bar{4}$	0$\bar{3}$	0 $\bar{2}$	0 $\bar{1}$	0 $\bar{0}$	01
6 $\bar{6}$	1 $\bar{2}$	1 $\bar{1}$	1 $\bar{0}$	1 $\bar{1}$	1 $\bar{2}$	1 $\bar{3}$	1 $\bar{4}$	1 $\bar{4}$	1 $\bar{5}$	0 $\bar{4}$	0 $\bar{3}$	0 $\bar{2}$	0 $\bar{1}$	0 $\bar{0}$

to the right one digit position each iteration to achieve a multiplication of the current partial product by ten, thus accounting for the increase in weight of each successive multiplier digit. Each iteration, $n + 1$ digits from the partial product, P_i , and $n + 1$ digits from the intermediate product, IP_{i-1} , pass through $n + 1$ Svoboda digit adders. The range of inputs and their signed-digit sums are shown in Table 4.5. Figure 4.6, lines 12, 21, and 27, provide examples of accumulating the partial products.

In shifting the intermediate product one digit position to the right, the LSD is made available for completion as no subsequent partial product digits will be added to this digit. Since this emergent digit is still in the signed-digit code described in Table 3.3, it must be converted to BCD. During the conversion process, the transfer digit from the previous iteration's intermediate product LSD, t_{i-1} , must be taken into

account. Logically, the conversion is as follows. If the LSD is greater than zero, the LSD is simply converted to BCD and then decremented if the input transfer digit is -1 . If the LSD is less than or equal to zero, the radix complement of the additive inverse of the LSD is converted to BCD and then decremented if the input transfer digit is -1 (only the four least significant bits are kept). Lastly, an output transfer digit, t_i , is assigned a value of -1 if the LSD is negative or if the LSD is 0 and the input transfer digit is -1 , otherwise it is assigned a value of 0. Note, since the transfer digit in this situation only indicates a borrow or no borrow, a single bit can be used. Equations 4.22 and 4.23 show the different cases for converting the intermediate product LSD and generating the transfer bit, respectively. A straightforward implementation of this conversion and generation of a transfer bit requires twelve logic levels, as determined by SIS. The final product is identified by FP . The notation $\rightarrow BCD$ is used to indicate a mapping from the signed-digit form to BCD form.

$$fp_i = \begin{cases} (IP_i[LSD] \rightarrow BCD) + t_{i-1} & \text{if } IP_i[LSD] \geq 1 \\ 10 - (\overline{IP_i[LSD]} \rightarrow BCD) + t_{i-1} & \text{if } IP_i[LSD] \leq 0 \end{cases} \quad (4.22)$$

$$t_i = \begin{cases} 0 & \text{if } IP_i[LSD] \geq 1 \\ -1 & \text{if } IP_i[LSD] \leq 0 \ \& \ t_{i-1} = -1 \end{cases} \quad (4.23)$$

After all the multiplier digits have been processed, the signed-digit outputs of the Svoboda adders comprising IP_{n-1} need to be converted to BCD to produce the final product digits, fp_0 to fp_{n-1} . Additionally, the transfer bit, t_{n-2} , must be added to the LSD, i.e., $IP_{n-1}[n-1]$. The algorithm to convert the signed-digits, which is on the order of carry-propagate addition, is fully described in [4]. Figure 4.6, lines 29/33, provide an example of converting an intermediate product ($1\bar{1}0$) and a transfer bit

($\bar{1}$) into BCD digits.

Implementation

Figure 4.9 shows one possible multiplier implementation using the presented ideas. As shown, this implementation requires $n + 4$ cycles, which is the same latency as the design of Section 4.1.1 published in [39]. In the first cycle, operand A and a single digit of operand B are recoded. Then, the outputs of the recoder blocks are input to the digit multipliers to yield a sign-corrected partial product in overlapped form. In the second cycle, the overlap of the two-digit products is removed and the partial product is recoded in a manner appropriate for a Svoboda signed-digit adder. For the next n cycles, a partial product is added to the previous iteration's intermediate product, and a new partial product is generated. In the last two cycles, the final intermediate product is converted into BCD digits.

Figure 4.6 shows an example of multiplying 339 by 265 using the proposed multiplier implementation. In cycle 1, the multiplicand and the LSD of the multiplier are recoded into the signed-digit numbers $34\bar{1}$ (line 3) and $\bar{5}$ (line 4), respectively. Also in cycle 1, the recoded multiplicand (line 3) is multiplied by the LSD of the recoded multiplier (line 4) to yield the partial product in overlapped form (lines 5/6). In cycle 2, the partial product generated in overlapped form in cycle 1 is converted to non-overlapped form (line 7). Additionally, the next more significant digit in the multiplier is recoded (line 9) and a partial product based on this digit is generated in overlapped form (lines 10/11). In cycle 3, the accumulation of the partial product is initiated by adding the partial product in line 7 to the intermediate product, previously initialized to zero (line 12). Also in cycle 3, the partial product in overlapped form from the previous cycle is converted to non-overlapped form (line 14), the MSD of the multiplier digit is recoded (line 16), and a partial product based on this digit

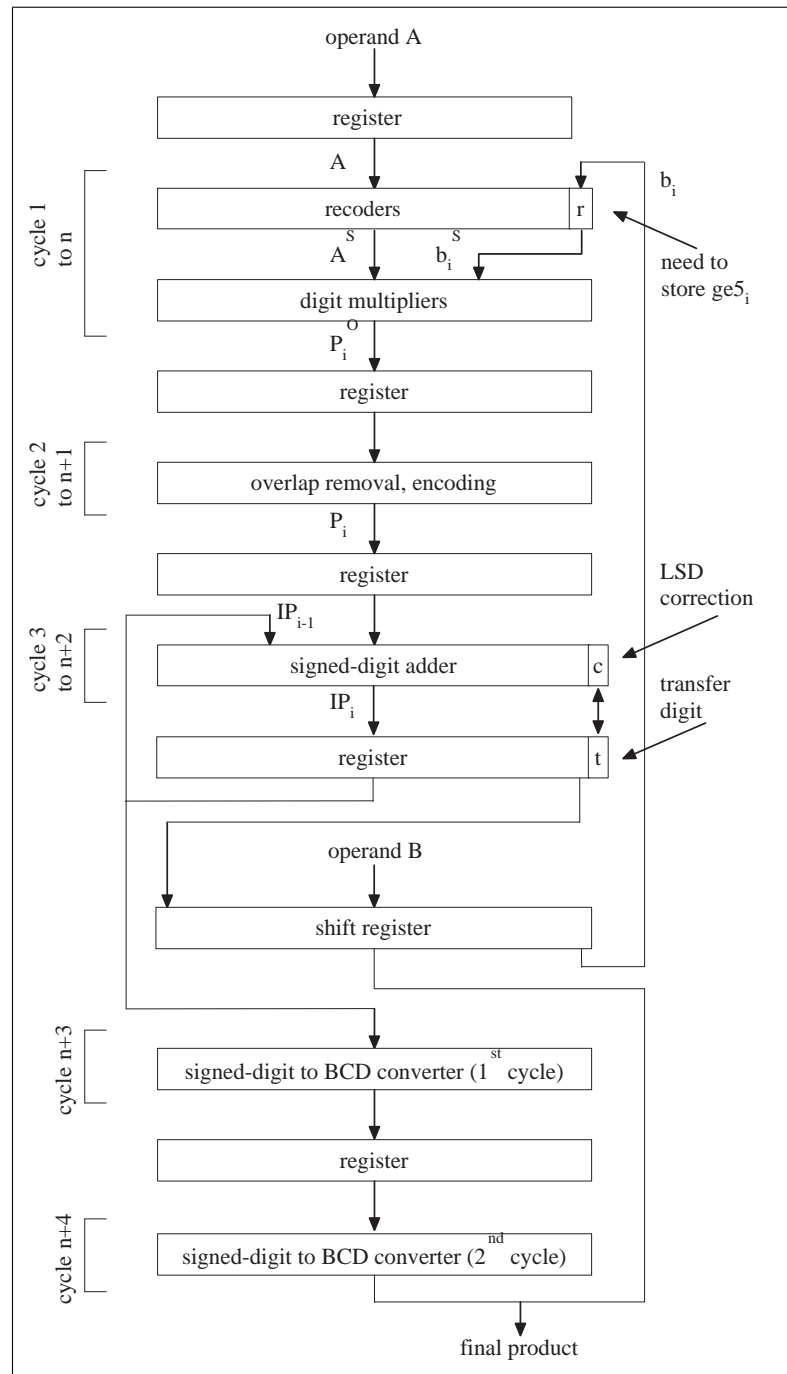


Figure 4.9: Iterative DFXP Multiplier Using Signed-Digits Adders

is generated in overlapped form (lines 17/18). In cycle 4, the first digit of the final product (i.e., the LSD) is produced by converting the LSD of the intermediate product to BCD (line 19). The conversion takes into account the previously cleared transfer bit and produces an output transfer bit for the next intermediate product's LSD conversion to BCD (line 20). Also in cycle 4, another partial product is added to the intermediate product (line 21) and the previous cycle's partial product is converted to non-overlapped form (line 23). Cycle 5's function includes the conversion to BCD of the intermediate product LSD developed in cycle 4 (line 25), the generation of an output transfer bit (line 26), and the addition of the partial product developed in cycle 4 to the intermediate product (line 27). In cycle 6, the two-cycle process of converting the final intermediate product to BCD digits is initiated (line 29). Also in cycle 6, another intermediate product LSD is converted to BCD (line 31) and an output transfer bit is developed (line 32). In the final cycle, 7, the conversion of the final intermediate product to BCD digits is completed (line 33).

Summary

This DFXP multiplier utilizes the novel approach of restricted-range, signed-digits throughout the multiplication process to generate and accumulate the partial products in an efficient manner. To achieve the restricted range, a simple recoding scheme is used to produce signed-magnitude representations of the operands. The partial product generation takes the recoded digits, which are in the range of $\{-5, \dots, +5\}$, and uses combinational logic to obtain products for input digits in the range $\{2, \dots, 5\}$. The results from the partial product generation logic are then recoded and added to the accumulated sum of previous partial products via a signed-digit adder. Finally, the signed-digit result is converted to BCD result. Original aspects of this work include: 1) the method used for recoding the digits into a

signed-magnitude representation; 2) the design of the decimal partial product generation; and, 3) the recoding of the partial products before sending them into the signed-digit adder. This design scales well to support larger width operands.

4.1.3 Summary of Iterative DFXP Designs

The two iterative DFXP multiplier designs of Subsections 4.1.1 and 4.1.2 achieve similar latencies, yet there are a couple notable differences. The multiplier design employing decimal CSAs develops and stores a reduced set of multiplicand multiples to reduce the delay in producing the partial product needed in each iteration. However, in the multiplier design employing signed-digit adders, there are no multiplicand multiples to distribute, only the multiplicand. The second difference relates to area, as described next.

Upon considering which design to extend to support DFP, the design employing decimal CSAs was chosen as it was deemed to have less area. This is because the combination of the decimal CSAs and decimal compressors contained less logic than the combination of the digit multiplier blocks and signed-digit adders. Had the ensuing research been more focused on extending the design to a parallel implementation, the multiplier using the signed-digit adders may have been chosen as it is far less demanding of wiring. In the next section, an iterative DFP multiplier design is presented.

4.2 Floating-Point Design

There are several papers presenting hardware designs for DFP multiplication [23, 25, 87, 89]. The multiplier designs by Cohen *et al.* [87] and Bolenger *et al.* [89] iterate over the multiplicand and multiplier operands on a digit-by-digit basis accumulating the partial products along the way. Furthermore, the results they produce do not comply with IEEE 754-2008. In contrast, the multiplier designs used on the IBM Power6 [23] and System z10 [25] processors do conform with IEEE 754-2008. These multipliers are essentially the same, as they use the same word-by-digit algorithm. See Section 3.3.2 for a description of their latencies.

The DFP multiplier described in this section is based on the iterative DFXP multiplier of [39] which utilizes decimal CSAs for partial product generation and decimal compressors for partial product accumulation. This research, presented in [41] and compared against a parallel DFP multiplier in [42], is believed to be the first DFP multiplier design in compliance with IEEE 754-2008.

4.2.1 Algorithm

The DFP multiplier design presented in this paper extends the iterative DFXP multiplier design published in [39] and described in Section 4.1.1 to comply with IEEE 754-2008. As a brief summary of this fixed-point design, the multiplication is achieved by generating the double, quadruple, and quintuple of the multiplicand, successively evaluating the digits of the multiplier from LSD to MSD, selecting two tuples which sum to the necessary partial product, adding the partial product to the previous iteration's accumulated product, and generating a double-width product from the iterative portion's intermediate product (produced in sum and carry form). Extending this DFXP functionality to support DFP requires the algorithm depicted

in the flowchart-style drawing shown in Figure 4.10. In the figure, the steps of the DFXP multiplier from [39] are surrounded by a dashed rectangle, with the exception of the “Generate Sticky Bit” in the “Iterative Portion” block and the “Shift Left” steps.

The operation begins with the reading of the operands from either a register file or from memory (block labeled “Read Operands” in Figure 4.10). As the operands are encoded via the DPD algorithm, each must be decoded (block “DPD Decode Operands”). Next, the double, quadruple, and quintuple of the multiplicand are generated in the datapath portion of the design (block “Generate Multiplicand Multiples”). Then, in an iterative manner starting with the LSD of the multiplier operand, each digit is used to select two multiplicand multiples to add together to yield the respective partial product, the partial product is added with the previous iteration’s accumulated product, the accumulated product is shifted one digit to the right, and the sticky bit is generated (see the block labeled “Iterative Portion”). All the additions in the iterative portion of the algorithm yield intermediate results in a redundant form.

In parallel with the generation of the multiplicand multiples and the accumulation of partial products, the significands are examined to determine their leading zero counts, LZ^A and LZ^B , the exponents are examined to determine the intermediate exponent, IE , and the signs are XORed to determine the product sign, s^P (see the block whose label begins with “Determine Leading Zero Counts”). Based on the leading zero counts and the intermediate exponent, two vital control values are generated: a shift left amount, SLA , and a sticky counter, SC (block label begins with “Generate Shift Left Amount”). The shift left amount is needed to properly align the intermediate product, IP , prior to rounding. The sticky counter is needed to generate the sticky bit, sb , created on-the-fly during the accumulation of partial

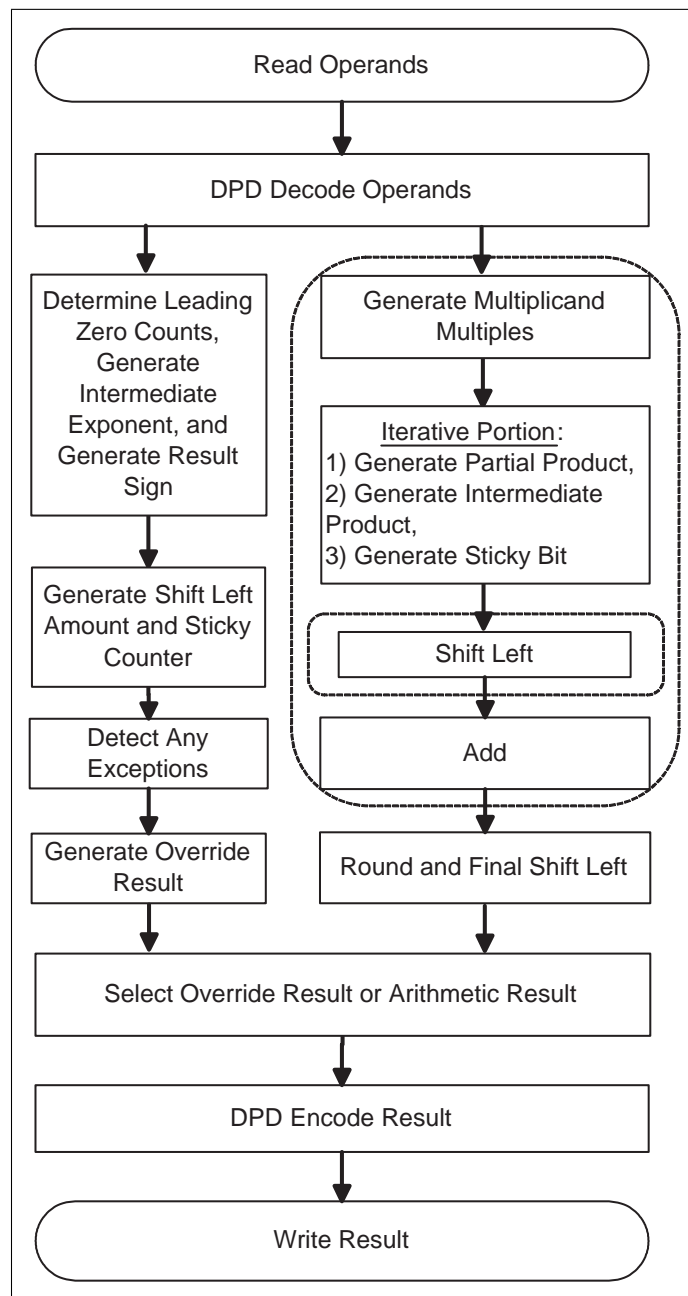


Figure 4.10: Flowchart of Iterative DFP Multiplier Using Decimal CSAs

products.

At the end of the iterative accumulation of partial products, the intermediate product is in the $2p$ -digit intermediate product register. The intermediate product is then shifted left based on the SLA in an effort to yield a product with the appropriate exponent value (see the “Shift Left” block). Since the shifted intermediate product is in redundant form, an add step is necessary to produce a non-redundant product (block labeled “Add”). Ultimately, a p -digit rounded product needs to be delivered. Thus, the shifted intermediate product, SIP , is rounded and a final shift is performed to yield a rounded intermediate product, RIP , which is the result barring any exceptions (see block labeled “Round and Final Shift Left”). After rounding the rounded intermediate product, the product exponent, E^P , product significand, C^P , and product combination field are generated⁵ to produce the arithmetic result.

Using the operands’ combination fields, the intermediate exponent, and information from the shift and round steps, a determination is made as to whether an exception needs to be signaled and corrective action taken (see blocks labeled “Detect any Exceptions” and “Generate Override Result”). Depending on whether an exception is detected, the override result or arithmetic result is selected (see “Select Override Result or Arithmetic Result”). Finally, the result is DPD encoded and written to a register file or memory (see blocks “DPD Encode Result” and “Write Result”, respectively).

4.2.2 Features

Figure 4.11 depicts the top portion of the DFP multiplier design. It is provided to aid in visualizing the manner in which partial products enter the intermediate

⁵In actuality, the leading digit of C^P and the leading two bits of E^P are contained in the combination field.

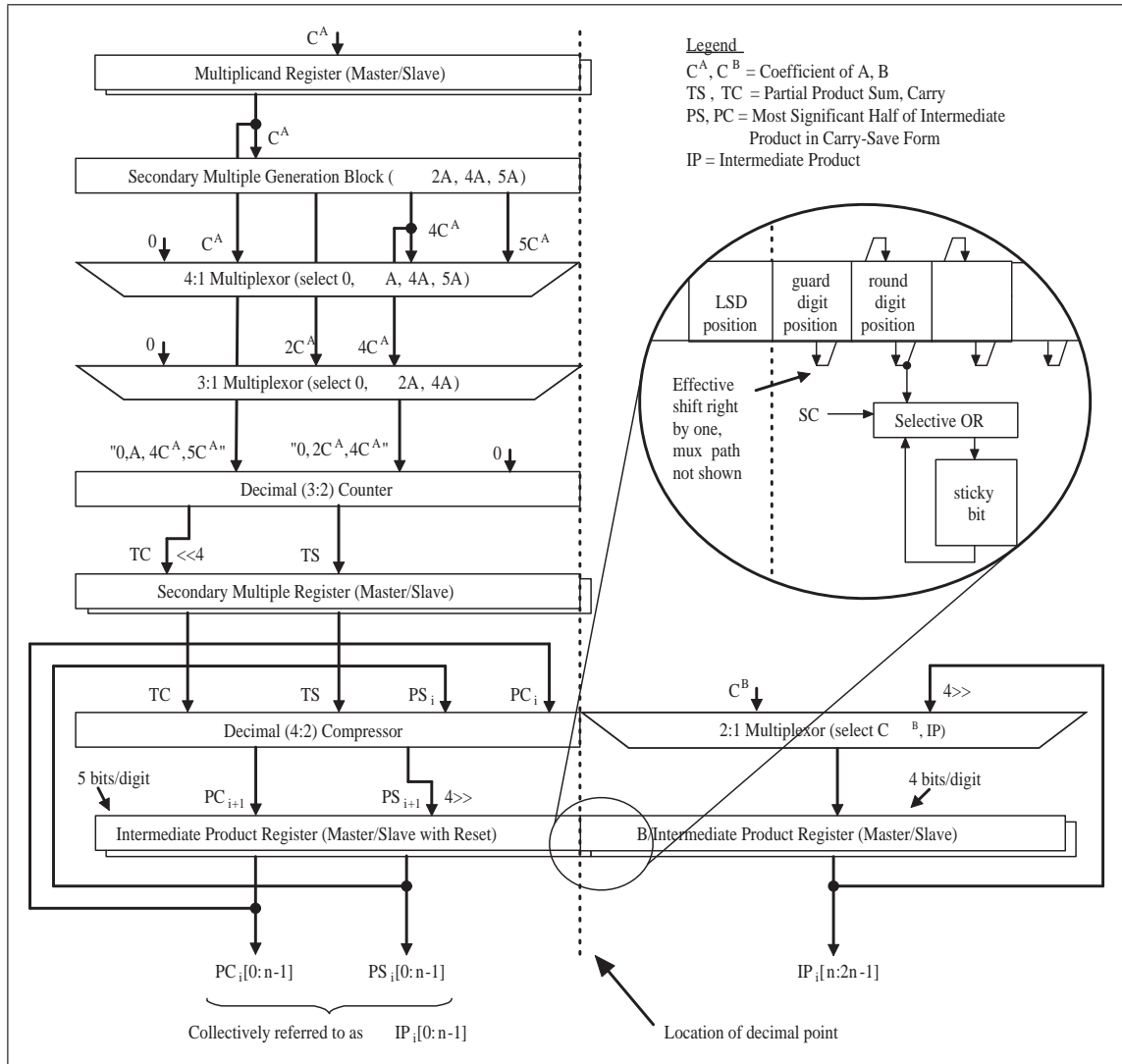


Figure 4.11: Top Portion of Iterative DFP Multiplier Design

product register for accumulation, the location of the decimal point in the datapath, and the generation of the sticky bit. With the exception of the sticky bit generation shown in the magnified inset, this top portion of the multiplier design is the DFXP multiplier design described in [39].

A critical design choice is the location of the decimal point in the datapath as this dictates the direction the intermediate product may need to be shifted and the

location and implementation of the rounding logic. For this design, the location is chosen to be exactly in the middle of the intermediate product register. This keeps the decimal point in the same location throughout the datapath. Further, the intermediate product need only be shifted in one direction to produce a rounded product (except when underflow occurs).

In the remainder of this subsection, the primary components necessary to perform DFP multiplication are described in detail. These include generating the intermediate exponent, shifting the intermediate product, generating the sticky bit, generating the result sign, rounding, and detecting and handling any exceptions.

Intermediate Exponent Generation

At the end of partial product accumulation, p digits of the intermediate product are to the right of the decimal point. Thus, the intermediate exponent of the intermediate product, IE^{IP} , is the preferred exponent increased by p :

$$\begin{aligned} IE^{IP} &= E^A + E^B - bias + p \\ &= PE + p \end{aligned} \tag{4.24}$$

After left shifting the intermediate product as part of preparing a p -digit final product, the intermediate exponent is decreased by the shift left amount, SLA (described in the next subsection). The exponent associated with this shifted intermediate exponent, IE^{SIP} , is calculated as follows.

$$\begin{aligned} IE^{SIP} &= E^A + E^B - bias + p - SLA \\ &= PE + p - SLA \\ &= IE^{IP} - SLA \end{aligned} \tag{4.25}$$

The shifted intermediate product, SIP , is then rounded to become the rounded intermediate product, RIP , and the associated exponent is named the intermediate exponent of the rounded intermediate product, IE^{RIP} . The IE^{RIP} is one less than IE^{SIP} or equal to IE^{SIP} , depending on whether or not a corrective left shift of one digit occurs during rounding. However, the product exponent may differ from IE^{RIP} due to an exception.

Related to the intermediate exponent calculations is the determination of the amounts by which IE^{IP} is less than the minimum exponent, E_{min} , or more than the maximum exponent, E_{max} . These comparisons are used to increase or decrease the shifting of the intermediate product in an effort to prevent an exception. As the shifting of the intermediate product may be affected by these comparisons, the generation of the sticky bit must be altered accordingly. The computation of the shift amount for the intermediate product, the generation of the sticky bit, and the handling of extreme numbers are described in the following subsections.

Intermediate Product Shifting

As mentioned earlier, the intermediate product may need to be shifted to achieve the preferred exponent or to bring the product exponent into range. Calculating the shift amount is dependent upon, among other things, the number of leading zeros in the intermediate product. However, instead of waiting until the intermediate product is generated to count the leading zeros, the latency of the multiplication is reduced by determining a shift amount based on the leading zeros in both the multiplicand and multiplier significands. With this approach, the pre-calculated shift amount may be off by one since the number of significant digits in the final product may be one less than the sum of the significant digits of each significand. Thus, the product may need to be left shifted one additional digit at some point after the initial shift.

Except when $IE^{IP} < Emin$, the shift is always to the left⁶. This is because each partial product is added to the previous accumulated product with its LSD one digit to the right of the decimal point. With an estimate of the significance of the intermediate product based on the significance of each significand, $S^{IP} = S^A + S^B$, the shift left amount is determined as follows. If $S^{IP} > p$, then one or more leading zeros of the intermediate product may need to be shifted off to the left to maximize the significance of the result. However, if $S^{IP} \leq p$, then the entire product will reside solely in the lower half of the intermediate product register (assuming all the multiplier significand digits have been processed). In the latter case, the less significant half of the intermediate product register can be placed into the upper half, by left shifting the intermediate product p digits. These two situations lead to the following equation for the shift left amount, SLA .

$$\begin{aligned}
 SLA &= \min((2 * p) - (S^A + S^B), p) \\
 &= \min((2 * p) - ((p - LZ^A) + (p - LZ^B)), p) \\
 &= \min(LZ^A + LZ^B, p)
 \end{aligned} \tag{4.26}$$

where LZ^A and LZ^B are leading zero counts of the significands, C^{A7} and C^B , respectively.

In the event the actual significance of the intermediate product is one digit less than the estimated significance, it may be necessary to left shift the intermediate product one more digit after the initial left shift. The potential for a corrective left shift of one digit necessitates maintaining an additional digit to the right of the

⁶This assumes every multiplier digit is processed during the partial product accumulation portion of the multiplication algorithm.

⁷Note, C^A is used instead of A as the discussion regards floating-point numbers. C^A represents the significand of the floating-point operand A .

decimal point. This digit is referred to as the *guard* digit and is analogous to the guard bit used in BFP multiplication. The handling of the final left shift by one digit occurs in the rounding portion of the algorithm and is described in Section 4.2.2.

The shift left amount, as estimated above, is dependent on all the multiplier significand's digits being processed. One design option is to exit the iterative portion of the algorithm after processing the most significant nonzero digit of the multiplier significand. Although this option complicates the processor's instruction issue and completion logic, substantial cycles may be saved for certain workloads. If early exit is to be supported, the shift left amount calculation, as well as other design components, needs to be altered accordingly.

Sticky Determination

After left shifting, any nonzero digits in the less significant half of the intermediate product register must be evaluated in the context of the rounding mode to determine if rounding up is necessary. As mentioned in the previous subsection, a corrective left shift of one digit may be needed if the actual significance of the intermediate product is one less than the sum of the significands' significance. In the event the corrective left shift is performed and the guard digit is shifted into the LSD position of the more significant half of the intermediate product register, the next digit must be maintained such that it can be determined if the remaining digits are less than one half the Unit in the Last Place (ULP), exactly one half ULP, or greater than one half ULP. Since this digit in the next less significant position to the guard digit is critical to rounding, it is called the *round* digit, which is analogous to the round bit in binary multiplication. The bits of the digits to the right of the round digit can all be logically ORed to produce a *sticky* bit.

To improve the latency and area of DFP multiplication, it is desirable to know *a*

priori which digits will be used in the sticky bit calculation. Having such knowledge allows the sticky bit to be generated on-the-fly with less hardware and wiring resource than waiting until the entire intermediate product is available and then selecting which digits should be ORed together. This can be readily accomplished in this design as a non-redundant digit, formed during the accumulation of a new partial product, enters the MSD of the less significant half of the intermediate product register while the intermediate product is right shifted one position.

To determine when a digit being right shifted from the round digit position to the next less significant digit position should be included in the sticky bit generation, a counter is used. The starting value of this counter is initialized just prior to the first partial product entering the intermediate product register and cleared between operations. Whenever the counter value is greater than zero, the digit being shifted out of the round digit position is ORed with the previous sticky bit value, which is cleared between operations. The initial value of the sticky counter, SC , is generally the significance of the intermediate product minus the format's precision, unless this difference yields a negative number. Thus,

$$\begin{aligned}
 SC &= \max(0, S^{IP} - p) \\
 &= \max(0, (p - LZ^A) + (p - LZ^B) - p) \\
 &= \max(0, p - (LZ^A + LZ^B))
 \end{aligned} \tag{4.27}$$

Note the counter is decremented twice before any nonzero data enters the digit position to the right of the round digit position. This insures the two digits which end up in the guard and round digit positions after left shifting are not included in the sticky bit generation. Up to two more cycles can be taken to generate SC so long as the value of the counter is correspondingly less than what is described in

Equation 4.27. Also, if the intermediate product needs to be left shifted one additional digit, the sticky bit calculated in the manner above is still legitimate. This is because the guard digit will be moved into the LSD position of the product, and the round digit and sticky bit calculated in the manner above are all that is needed for rounding.

Sign Processing & Exception Handling

Sign processing is relatively straightforward. If the result is a number, the sign of the result is simply the XOR of the signs of the operands. However, if the result is NaN, IEEE 754-2008 does not specify the value of the sign bit. For ease of implementation, the sign logic used when the result is a number is also used when the result is NaN.

As described in Section 2.4.4, there are four exceptions that may be signaled during multiplication: invalid operation, overflow, underflow, and inexact. The detection and handling of overflow, underflow, and inexact exceptions, being associated with the rounding mechanism, is explained in the paragraphs following the description of the rounding mechanism. The invalid operation exception behaves as described in Section 2.4.4 except for the when one operand is a signaling NaN and the other is a quiet NaN. Though not required behavior in IEEE 754-2008, this design converts the signaling NaN to a quiet NaN and returns this as the result. This behavior was chosen because the diagnostic information potentially contained in the signaling NaN is deemed more important than the diagnostic information potentially contained in the quiet NaN. The rounding mechanism is now described.

Rounding

Rounding is required when all the essential digits of the intermediate product cannot be placed into the product significand or when overflow or underflow occurs.

The description of each rounding mode required by IEEE-2008 and its associated condition(s) are listed in Table 4.6. In the case of rounding based solely on the number of essential digits, rounding is accomplished by selecting either the shifted intermediate product truncated to p digits or its incremented value. In order to determine which value is to be selected, the following are needed: the rounding mode, the product's sign, the shifted intermediate product, including a guard digit, g , round digit, r , and sticky bit, sb , and an adder.

Table 4.6: Rounding Modes, Conditions, and Product Overrides for Overflow

Rounding Mode	Condition for Round-up (Non-overflow)	Product Override (Overflow)	
		$s^{IP} == 0$	$s^{IP} == 1$
Nearest, ties to even	$(g > 5) \vee ((g == 5) \wedge (l[3] \vee (r > 0) \vee sb))$	$+\infty$	$-\infty$
Nearest, ties away from 0	$g \geq 5$	$+\infty$	$-\infty$
Toward positive infinity	$\overline{s^P} \wedge ((g > 0) \vee (r > 0) \vee sb)$	$+\infty$	$-N$
Toward negative infinity	$s^P \wedge ((g > 0) \vee (r > 0) \vee sb)$	$+N$	$-\infty$
Toward 0 (truncate)	<i>none</i>	$+N$	$-N$

$+N$ = largest finite positive number, $-N$ = largest finite negative number

The adder must be able to produce a non-redundant sum from its inputs, some of which may be in a redundant form. Further, it must be able to add a one into either its LSD position or its guard position. The two options for the position to inject a one are necessary as the estimate of the shift left amount may be off by one, in which case a corrective left shift is required. Though this may appear to require more than one adder, both situations can be supported by using a single compound adder. The following explains how a compound adder can be used to produce the truncated or incremented intermediate results needed for rounding.

The inputs to the adder are the data in the p MSD positions of the shifted inter-

mediate product. To understand why it is sufficient to use only one compound adder p digits wide, consider the four possibilities of adding a zero or a one into the LSD or guard digit positions. Clearly, adding a zero into the guard digit position is the same as adding a zero into the LSD position (so long as the original guard digit is concatenated). The remaining two possibilities are related in the following way. If a one is added into the guard digit position and a carry occurs out of the guard digit position (i.e., $g == 9$), then this is equivalent to adding a one into the LSD position and changing g to zero. Conversely, if a carry does not occur out of the guard digit position (i.e., $g < 9$), then this is equivalent to adding a zero into the LSD position and concatenating the incremented guard digit.

Before presenting the rounding scheme employing the compound adder, the following simplification is offered. This design need only contend with two alignment choices of the final p result digits from the $2p$ shifted intermediate product. The two alignment choices for the final result are the p digits starting in the MSD position or the p digits starting with the digit to the right of the MSD position. These choices are not affected by rounding, i.e., rounding will not cause a third alignment choice. The choice between the aforementioned two possible locations for the final p digits is affected by rounding, the preferred exponent, and the minimum exponent value. These dependencies are described in the paragraphs following the proof of why rounding cannot cause a third alignment choice for the final product. To see why rounding cannot cause a third alignment choice, the following proof is provided.

The proof is a generalization of the fact that two significands, each with significance equal to the precision, will yield an intermediate product with significance $2p - 1$ or $2p$. If the significance of the intermediate product, S^{IP} , is $2p - 1$, then there will be a zero in the MSD position after left shifting the intermediate product (see Section 4.2.2). In this case, an increment due to rounding will not propagate beyond

the MSD position. Alternatively, if $S^{IP} = 2p$, then in order for carry-out to occur, the minimum value of a string of nines p digits long must start in the MSD position. Thus, the minimum value of the intermediate product needed for a carry-out can be expressed as $IP = C^A \times C^B = 10^{2p} - (10^p - 1)$. However, the maximum intermediate product is $10^{2p} - 10^p - (10^p - 1)$, as demonstrated below. Since the maximum intermediate product is 10^p less than the minimum intermediate product needed for a carry-out, the intermediate product will never need to be shifted to the right. Thus, a third alignment choice for the final product cannot occur. The generalized proof that rounding does not cause a third alignment choice is given below.

Proof of Rounding Not Causing a Third Alignment Choice

Given the range of significands, C^A and C^B , as:

$$10^{S^A-1} \leq C^A \leq 10^{S^A} - 1, \text{ and}$$

$$10^{S^B-1} \leq C^B \leq 10^{S^B} - 1,$$

the intermediate product's, $C^{IP} = C^A \times C^B$, range is:

$$10^{S^A-1} \times 10^{S^B-1} \leq C^{IP} \leq (10^{S^A} - 1) \times (10^{S^B} - 1),$$

or equivalently:

$$10^{S^A+S^B-2} \leq C^{IP} \leq 10^{S^A+S^B} - 10^{S^A} - (10^{S^B} - 1).$$

Using the shift left amount, $SLA = 2p - S^A - S^B$, the

upper bound for the shifted intermediate product is:

$$C^{SIP} \leq C^{IP} \times 10^{2p-S^A-S^B}$$

or equivalently:

$$C^{SIP} \leq 10^{2p} - 10^{2p-S^B} - (10^{2p-S^A} - 10^{2p-S^A-S^B}).$$

Substituting p for S^A and S^B , as this is when the shifted

intermediate product is at its maximum, yields:

$$C^{SIP} \leq 10^{2p} - 10^p - (10^p - 1).$$

This maximum achievable value of the shifted intermediate product is less than the minimum value required for carry-out, $10^{2p} - (10^p - 1)$, thus demonstrating rounding does not cause a third alignment choice. ■

Returning to the function of rounding due to nonzero data in the guard, round, or sticky positions of the shifted intermediate product, the description of the design's rounding scheme is presented. The use of a single compound adder to generate both a p -digit significand and its incremented value, and the guarantee of no post-rounding normalization, allows a simple and efficient rounding scheme to be developed which is unique from recent binary rounding schemes such as those presented and referenced in [138]. Here, C^{+0} and C^{+1} are used to represent the plus zero and plus one sums, respectively, emerging from the compound adder.

First, an indicator, $grsb$, is set whenever there are nonzero digits to the right of the LSD position of the shifted intermediate product. That is, $grsb = (g > 0) \wedge (r > 0) \wedge sb$. This indicator, when set, may lead to a corrective left shift if there is a leading zero in the compound adder's outputs. The corrective left shift does not happen when round up is to occur and the first p digits of the shifted intermediate product are zero followed by all nines. In this case, a round up will produce a carry into the MSD position and the corrective left shift must be preempted. Fortunately, this unique case can be readily detected. It is the only situation in which the MSD of C^{+0} is a zero and the MSD of C^{+1} is a nonzero.

Next, for the given rounding mode, two round up values, $ru^{cls==0}$ and $ru^{cls==1}$, are computed for the cases of no corrective left shift by one and a corrective left shift by one, respectively. The computations are based on the shifted intermediate product and the round up condition(s) in Table 2.8. The difference between the two round up

values is for the case of a corrective left shift, the guard digit is treated as the LSD, and the round digit is treated as the guard digit. As an example, if the rounding mode is round toward zero, both round up values are zero. As another example, if the rounding mode is round to nearest, ties away from zero and the LSD, guard, round, and sticky values are 0, 5, 0, 0, then $ru^{cls==0} = 1$ and $ru^{cls==1} = 0$.

At this point, the guard digit must be incremented and an indicator developed when the original guard digit equals nine. The incremented guard digit is needed during a corrective left shift when round up is performed. The carry out of the decimal digit adder can be used as the $g == 9$ indicator, $g9$. However, it is important to note this carry out is never allowed to propagate into the LSD position of the compound adder.

Using C^{+0} , C^{+1} , $ru^{cls==0}$, $ru^{cls==1}$, $grsb$, g , $g + 1$, and $g9$, the algorithm presented in Figure 4.12 realizes rounding for the DFP multiplier design. The algorithm is presented as three distinct cases involving the MSDs of the two conditional sums, $C[0]^{+0}$ and $C[0]^{+1}$. The fourth case, when the plus zero sum has a zero in its MSD and the plus one sum has a nonzero digit in its MSD, cannot happen.

Though the rounding scheme of Figure 4.12 may appear complex, the choice is simply between C^{+0} , C^{+1} , or these values left shifted one digit with either g or $g + 1$ concatenated. For those cases in which a left-shifted form of a conditional sum is chosen, the intermediate exponent of the shifted intermediate product is decremented.

Handling Overflow and Underflow

In the case of rounding due to overflow, the product is rounded according to column IV in Table 2.8. The table describes the product to be generated for each rounding mode under default exception handling as specified in IEEE 754-2008.

Ultimately, the detection of overflow occurs by comparing the intermediate ex-

“No leading zeros, no corrective left shift”

1. $C[0]^{+0} \neq 0$ and $C[0]^{+1} \neq 0$

(a) $ru^{cls==0} == 0 \Rightarrow C^P = C^{+0}$

(b) $ru^{cls==0} == 1 \Rightarrow C^P = C^{+1}$

“Leading zeros, possible corrective left shift”

2. $C[0]^{+0} == 0$ and $C[0]^{+1} == 0$

(a) $grsb == 0$

i. $IE^{SIP} == PE$ or $IE^{SIP} \leq Emin \Rightarrow C^P = C^{+0}$

ii. $IE^{SIP} > PE$ and $IE^{SIP} > Emin \Rightarrow C^P = (C^{+0} \ll 1) \parallel g$

(b) $grsb == 1$ and $IE^{SIP} \leq Emin$

i. $ru^{cls==0} == 0 \Rightarrow C^P = C^{+0}$

ii. $ru^{cls==0} == 1 \Rightarrow C^P = C^{+1}$

(c) $grsb == 1$, $IE^{SIP} > Emin$, and $ru^{cls==1} == 0 \Rightarrow C^P = (C^{+0} \ll 1) \parallel g$

(d) $grsb == 1$, $IE^{SIP} > Emin$, and $ru^{cls==1} == 1$

i. $g9 == 0 \Rightarrow C^P = (C^{+0} \ll 1) \parallel (g + 1)$

ii. $g9 == 1 \Rightarrow C^P = (C^{+1} \ll 1) \parallel (g + 1)$, note $g + 1 == 0$

“Zero followed by all nines”

3. $C[0]^{+0} == 0$ and $C[0]^{+1} \neq 0$

(a) – (c) same as in Case 2

(d) $grsb == 1$, $IE^{SIP} > Emin$, and $ru^{cls==1} == 1 \Rightarrow C^P = C^{+1}$

Figure 4.12: Rounding Scheme

ponent of the rounded intermediate product, IE^{RIP} , with the maximum exponent, E_{max} . However, the following steps are taken prior to this comparison in an effort to keep the intermediate exponent in range. If the intermediate exponent of the intermediate product (IE^{IP} in Equation 4.24) minus the shift left amount (SLA in Equ-

tion 4.26) is greater than E_{max} , then SLA is increased to subsequently decrease the intermediate exponent of the shifted intermediate product (IE^{SIP} in Equation 4.25). SLA can only be increased to the extent all the leading zeros of the intermediate product are removed. If there are only enough leading zeros to bring IE^{SIP} down to at least $E_{max} + 1$, then it is possible there will be one more leading zero in the shifted intermediate product than estimated, and a left shift of one digit can occur during rounding to prevent overflow. Note the sticky counter (SC in Equation 4.27) must be decreased by the same amount SLA is increased. After adjusting SLA and SC , shifting the intermediate product, and rounding the shifted intermediate product, IE^{RIP} is compared to E_{max} . If $IE^{RIP} > E_{max}$, then overflow has occurred and the rounding mode and product sign are used to select a product based on Table 2.8. Both the overflow and inexact exceptions are signaled.

Averting underflow is similar to that described for overflow. The calculated SLA must be decreased by the amount which would lead IE^{SIP} to drop below E_{min} . The sticky counter must be increased by the same amount SLA is decreased. Dissimilar from overflow, however, is the following behavior. If $IE^{IP} < E_{min}$, then SLA is set to zero, and the intermediate product is shifted to the right to bring the IE^{SIP} into range. To accomplish the right shift, the iterative portion of the algorithm is allowed to continue beyond the processing of all the multiplier significand digits. Partial products of value zero are used in the additional iterations so as not to alter the value of the intermediate product. The number of additional iterations is equal to the amount by which E_{min} exceeds IE^{IP} . Note the sticky counter must be increased by the number of additional iterations. The number of additional iterations is at most $p + 2$ as this amount is guaranteed to place the most significant nonzero digit of any accumulated product beyond the round digit position. Thus, with a properly adjusted sticky counter, all the intermediate product data is ORed into the sticky bit.

After clearing SLA , adjusting SC , “shifting” the intermediate product, and rounding the shifted intermediate product, IE^{RIP} is compared to $Emin$. If $IE^{RIP} < Emin$ and $grsb$ of the chosen compound adder output is one, then underflow has occurred. Additionally, if $IE^{RIP} == Emin$, the MSD of the chosen compound adder output is zero, and $grsb$ of the chosen compound adder output is one, then underflow has occurred. Both the underflow and inexact exceptions are signaled.

One design consideration is whether or not to allow underflow to have a variable latency. The number of additional iterations need only be $\min(p + 2, Emin - IE^{IP})$. However, to keep the processor’s instruction issue and completion logic simple, it may be best to stall for $p + 2$ cycles always. If handling underflow with fewer fixed cycles is desired, the existing left shifter can be altered to support right shifting as well. Since in this design the shifter is before the adder (i.e., the more significant half of the intermediate product is in a redundant form), the adder would need to be widened to support a greater number of digit positions containing redundant data.

There are a number of notable implementation choices that were made when designing this DFP multiplier, including: leveraging the leading zero counts of the operands’ significands, passing NaNs through the dataflow with minimal overhead, and handling gradual underflow via minor modification to the control logic. Equations 4.26 and 4.27, and indirectly 4.25, use the leading zero counts of the significands. This is intentional as the determination of leading zeros is a common function in floating-point units [139]. Once each digit is identified as zero or nonzero, the generation of the leading zero count is the same as that for a BFP mantissa. As the accumulation of partial products is iterative, a single leading zero counter is used to determine successively the leading zero counts of C^A and C^B . If an operand is NaN, that operand’s NaN payload is used when forming the result. Instead of supporting alternative paths through the dataflow, the control logic passes C^A through the

dataflow by multiplying it by 1. If operand B is NaN, C^B is held in the less significant portion of the intermediate product register while the control logic overrides the shift left amount such that C^B is left shifted into the more significant half of the shifted intermediate product register. As for gradual underflow, the control logic extends the iterative partial product accumulation portion of the algorithm and successively adds partial products equal to zero such that the accumulated partial product is right shifted until IE^{IP} increases to E_{min} or all nonzero data are shifted into the sticky bit. This support of gradual underflow extends the latency of the multiplier from 25 cycles to a maximum of $43 = 25 + (p + 2)$ cycles.

The next subsection describes how the components in this subsection are combined to realize a DFP multiplier.

4.2.3 Implementation and Analysis

Figure 4.13 shows all the design components from the previous subsection together to realize DFP multiplication. The block-level drawing is of the bottom datapath portion of the DFP multiplier design, beginning with the $2p$ -digit intermediate product register and a sticky bit that was generated on-the-fly (see Section 4.2.2). The top datapath portion of the design, ending with the same intermediate product register, is shown in Figure 4.11. Not shown in either design drawing is the control logic, which is where the intermediate exponents, the sticky counter, the shift left amount, and the rounding control are calculated.

Referring again to Figure 4.13, the first step is to shift the intermediate product based on the shift left amount, SLA (described in Section 4.2.2), and store the $p + 2$ digit output in the shifted intermediate product register. The two additional digits are needed for the guard and round digits. Then, in support of the rounding scheme

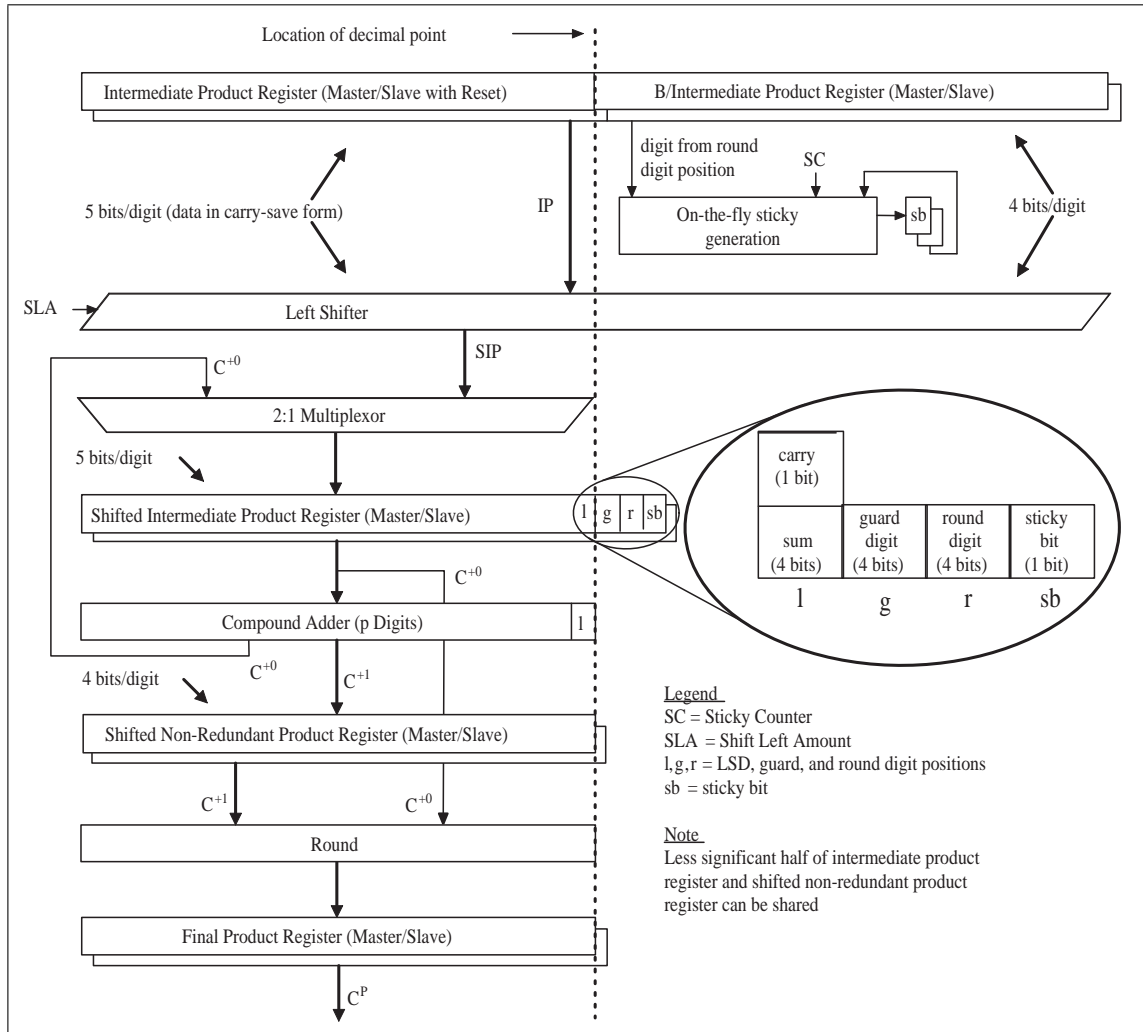


Figure 4.13: Bottom Portion of Iterative DFP Multiplier Design

presented in Section 4.2.2, a compound adder receives the data stored in the shifted intermediate product register. Since the data are either in non-redundant form or in sum and carry form (four sum bits and one carry bit), a unique compound adder is needed. For the sum portion of the addition, the $sum_i + carry_i + 0$ and the $sum_i + carry_i + 1$ is generated for each digit position, i . For the carry portion of the addition, a digit generate equal to $s_i[0] \wedge s_i[3] \wedge c_i$ and digit propagate equal to $s_i[0] \wedge (s_i[3] \vee c_i)$ are produced. The digit propagate and generate are then fed

into a carry network that generates the carries to select the appropriate digits to yield C^{+0} and C^{+1} . The compound adder is only p digits long, as described in Section 4.2.2. Once the two compound adder outputs are available in registers (i.e., C^{+0} and C^{+1}), the rounding logic produces the product significand, C^P , based on the rounding scheme in Figure 4.12.

Register transfer level models of both the presented 64-bit (16-digit) iterative DFP multiplier [41] and its predecessor iterative DFXP multiplier [39] were coded in Verilog. Both designs were synthesized using LSI Logic's gflxp 0.11um CMOS standard cell library and the Synopsys Design Compiler. To validate the correctness of the design, over 500,000 testcases covering all rounding modes and exceptions were simulated successfully on the pre-and post-synthesis design. Publicly available tests include directed-random testcases from IBM's FPgen tool [140] and directed tests available at [141].

Table 4.7 contains area and delay data for the iterative DFXP and DFP multiplier designs presented in Section 4.1.1 and Section 4.2, respectively. The values in the *FO4 Delay* column are based on the delay of an inverter driving four same-size inverters having a $55ps$ delay, in the aforementioned technology. The critical path in the iterative DFP multiplier is in the stage with the 128-bit barrel shifter, while for the iterative DFXP multiplier it is the decimal 4:2 compressor.

As shown in Table 4.7, extending the iterative DFXP multiplier design to support DFP multiplication affected the area, cycle time, latency, and initiation interval. The area of the DFP design is roughly twice that of the DFXP design. Approximately 20% of the area in the DFP design is associated with the increase in latches and associated logic. Another 20% is consumed in realizing the leading zero count, shift count, sticky counter logic, and left shift logic. And roughly 10% of the area is used for the expanded function of the adder and the rounding logic. The cycle time of the

Table 4.7: Area and Delay of Iterative Multipliers (DFXP vs. DFP)

$p = 16$	Iterative (Decimal CSAs)	
	DFXP [39]	DFP [41]
Latency (cycles)	20	25
Throughput (ops/cycle)	1/17	1/21
Cell count	59,234	117,627
Area (μm^2)	119,653	237,607
Delay (ps)	810	850
Delay ($FO4$)	14.7	15.4

DFP multiplier is 5% higher than the DFXP design, though a customized shifter may be able to reduce this gap. The latency of the DFP multiplier is five cycles longer than the DFXP multiplier, and the dispatch spacing between multiply operations increased by four cycles.

4.2.4 Summary

The iterative DFP multiplier presented in this section combines the necessary floating-point extensions of exponent processing, rounding, and exception detection and handling to the iterative DFXP multiplier design of [39]. The resultant multiplier design is compliant with IEEE 754-2008. Novel features of the multiplier include support for DFP numbers, on-the-fly generation of the sticky bit, early estimation of the shift amount, and efficient decimal rounding. All the presented design components, except the on-the-fly generation of the sticky bit, can be applied to parallel DFP multipliers, as illustrated in Section 5.2.

Chapter 5

Parallel Multiplier Designs

The multiplier designs presented in this chapter generate a reduced set of multiplicand multiples and accumulate all the partial products in parallel. As the accumulation of partial products occurs in parallel, these designs are pipelined to allow a throughput of one. This performance, as will be shown, comes at the expense of area. The reader is referred to Section 3.3 for an overview of the fundamental steps of hardware multiplication.

The work of two research teams who developed parallel DFXP multiplier designs is presented in this chapter for completeness and for comparison. The design of Lang and Nannarelli [5] appears in Section 5.1.1 and the designs of Vazquez, Antelo, and Montuschi appear in Section 5.1.2. My research [8], in collaboration with Schulte and Hickmann, on extending a parallel DFXP multiplier design of Vazquez *et al.* to support DFP multiplication as defined in IEEE 754-2008 is described in Section 5.2. A comparison is made between the iterative and parallel DFXP multiplier designs and between the iterative and parallel DFP multiplier designs. Lastly, commentary from [42] is provided that describes the preferred usage for each multiplier design.

5.1 Fixed-point Designs

The parallel DFXP multiplier designs described in this chapter feature the recoding of the multiplier operand to reduce the number of multiplicand multiples, and the use of carry-save addition for the accumulation of partial products. The design of Section 5.1.1 uses decimal CSAs for the partial product accumulation, while the design of Section 5.1.2 uses binary CSAs. The more salient and distinguishing features of these two parallel DFXP multiplier designs are presented along with area and delay data. A comparison is made between the iterative and parallel DFXP multipliers.

5.1.1 Multiplier Employing Decimal Carry-Save Adders

The parallel DFXP multiplier of Lang and Nannarelli [5] recodes the multiplier operand such that five multiplicand multiples ($-A, 2A, -2A, 5A, 10A$) need be generated and stored¹ for use in developing the partial products. The partial products are then accumulated through a reduction tree of decimal CSAs (developed in [39]). Referring to Figure 3.6 on page 63, this multiplier design recodes the multiplier operand (Figure 3.6.a), generates the partial products in parallel (Figure 3.6.b), accumulates the partial products in parallel (Figure 3.6.c), and then removes the redundancy in the intermediate product emerging from the partial product reduction tree (Figure 3.6.e) to yield the final product. This design is the first published parallel DFXP multiplier.

Algorithm

A flowchart-style drawing of the algorithm described in [5] appears in Figure 5.1. After reading in the operands, the multiplier operand is recoded according to Table 5.1 while the multiplicand multiples are being produced. That is, the multiplier operand

¹The $10A$ multiple can be generated on the fly from the original multiplicand by shifting left one digit.

is recoded in such a way that all the multiples can be realized from the multiple set $\pm A, \pm 2A, 5A, 10A$. These multiples are precomputed very quickly as doing so does not require carry propagation [105]. The negative multiples do require several gate delays to produce as the BCD-8421 format is not self-complementing. This portion of the algorithm appears in the blocks labeled “Generate Multiplicand Multiples” and “Recode Multiplier” in Figure 5.1.

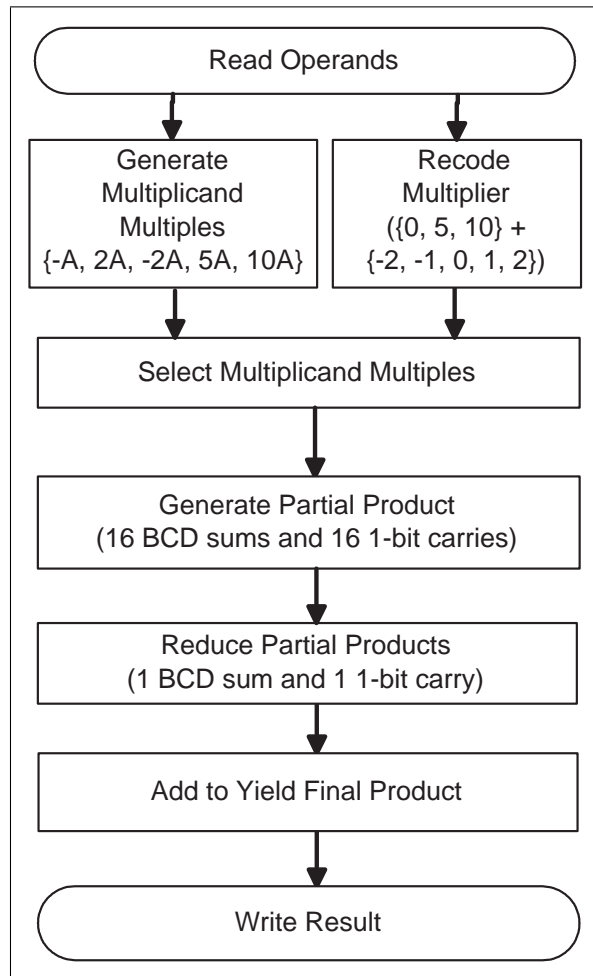


Figure 5.1: Flowchart of Parallel DFXP Multiplier Using Decimal CSAs [5]

Once the multiplicand multiples are generated and the multiplier operand is recoded, select signals are created to steer the appropriate multiples into a bank of

Table 5.1: Multiplier Operand Digit Recoding Scheme [5]

Multiplier Digit (decimal value of b_i)	Secondary Multiples		
	b'_i	+	b''_i
0	0	+	0
1	0	+	1
2	2	+	0
3	5	+	-2
4	5	+	-1
5	5	+	0
6	5	+	1
7	5	+	2
8	10	+	-2
9	10	+	-1

decimal CSAs to produce the partial products (see blocks “Select Multiplicand Multiples” and “Generate Partial Product”, respectively). The partial products then enter a partial product reduction tree comprised of decimal CSAs. This step is labeled as “Reduce Partial Products” in the flowchart figure. Emerging from the partial product reduction tree are two vectors, which are then added (shown in the “Add to Yield Final Product” block) before the result is written.

Features

As the internal format of each decimal digit is BCD (i.e., BCD-8421), the negative multiplicand multiples are produced by subtracting each digit from 9 (i.e., the diminished-radix complement) and then adding a 1 into the partial product accumulation tree at the LSD position of the non-negative secondary multiple, b'_i . Although some secondary multiples are negative, each partial product (i.e., $b'_i + b''_i > 0$) is non-

negative, and therefore, no sign extension is necessary. The $10A$ multiple is produced by left-shifting the multiplicand operand one digit.

The decimal CSAs used in the partial product reduction tree are the same as those developed for the design in [39] (see Section 4.1.1). For a $p = 16$ digit multiplier, there are 32 multiplicand multiples which enter 16 decimal CSAs. The resultant 16 partial products, represented by 16 vectors of BCD sum digits and 16 vectors of one-bit carries, are reduced via six levels of decimal CSAs as shown in Figure 5.2. Overall, 7 levels of decimal CSAs are needed. One novel aspect of this tree is the use of a counter to convert the available single-bit carries into BCD digits for accumulation further down the tree. In the worst-case column of the partial product array, there are 16 BCD sum digits and 16 carry bits to be reduced. All 16 BCD sum digits and half of the carry bits enter 8 decimal CSAs, while the remaining 8 carry bits enter the aforementioned counter to produce a single BCD sum digit.

At each level of CSAs, there are a number of single-bit carries for which there is no CSA to enter. These are handled by another instance of a counter, the output of which enters a final decimal CSA. After the last level of decimal CSAs, the intermediate product in decimal carry-save form enters a simplified BCD adder analogous to the one described in [39, 41].

Implementation and Analysis

The authors of [5] implemented their design in a HDL and synthesized it into STM 90nm CMOS standard cells using the Synopsys Design Compiler. In order to compare with the DFXP multiplier of [39], which was synthesized using a 110nm technology, a scaling factor of ~ 1.2 was used. Further, Lang *et al.* chose to fix the cycle time at the equivalent of 810ps, the cycle time of the design presented in [39], which yielded a pipeline depth of 11 cycles. In Table 5.2, the latency, throughput,

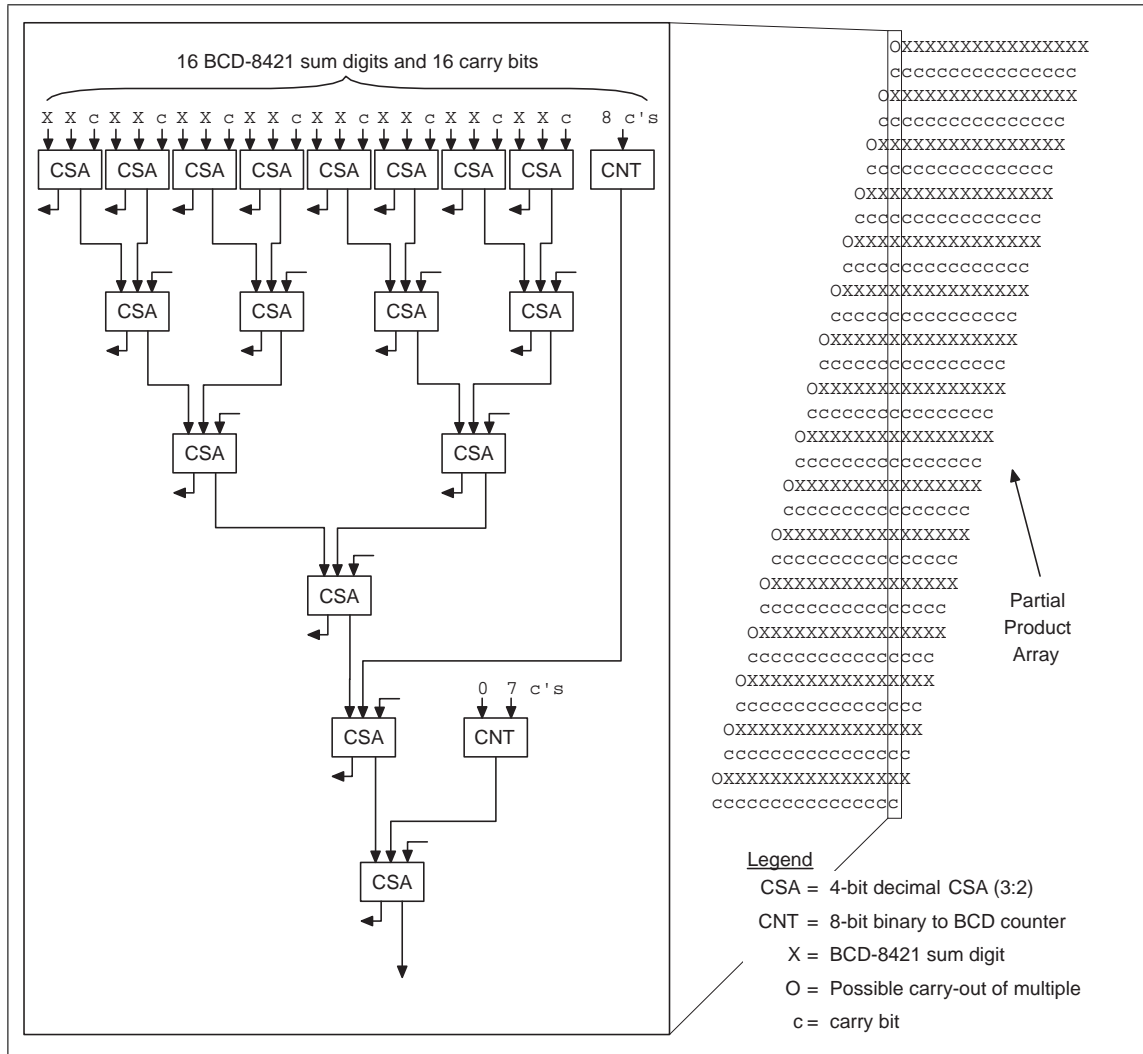


Figure 5.2: Partial Product Reduction Tree Employing Decimal CSAs [5]

area, and delay are presented for these two designs, with the numbers for the design of [5] scaled back to the technology used in [39]. From the table, one can conclude that parallel DFXP multiplication is a possibility for those situations in which area can be traded for high throughput and low latency.

Table 5.2: Area and Delay of DFXP Multipliers (Iterative vs. Parallel)

$p = 16$	Iterative	Parallel
	(Decimal CSAs) [39]	(Decimal CSAs) [5]
Latency (cycles)	20	11
Throughput (ops/cycle)	1/17	1
Cell count	59,234	180,000†
Area (um^2)	119,653	360,000‡
Delay (ps)	810	810‡
Delay ($FO4$)	14.7	14.7‡

† Approximated

‡ Scaled

Summary

The design of Lang and Nannarelli [5] recodes the multiplier operand to enable a reduced set of secondary multiples, reduces the $2p$ partial products down to a single intermediate product in decimal carry-save form through decimal CSAs, and then performs a final addition through a simplified BCD adder. This multiplier uses the same multiplier operand recoding scheme as [40] and effectively unrolls the iterative partial product reduction scheme of [39], albeit supplementing the decimal CSAs with counters to achieve a novel parallel DFXP multiplier. Although the area is significantly larger than the iterative DFXP multiplier of [39], the maximum throughput is one result per cycle and the latency is considerably better at a comparable cycle time.

5.1.2 Multiplier Employing Binary Carry-Save Adders

The parallel DFXP multipliers of Vazquez, Antelo, and Montuschi [6] utilize three different multiplier operand recodings, one employed in [40] and [5], such that only a reduced set of multiplicand multiples need be generated. The three recodings trade-off multiplicand multiple generation delay against the number of partial products to be reduced. That is, the recodings requiring a secondary multiple set in which each can be generated very quickly (i.e., without carry propagation) lead to a greater number of partial products to be reduced. The partial products are then accumulated through a reduction tree of binary CSAs. Referring to Figure 3.6 on page 63, this multiplier design recodes both the multiplicand and multiplier operands (Figure 3.6.a), generates the partial products in parallel (Figure 3.6.b), accumulates the partial products in parallel (Figure 3.6.c), recodes the intermediate product emerging from the partial product reduction tree (Figure 3.6.d), and then removes the redundancy in the intermediate product (Figure 3.6.e) to yield the final product.

Algorithm

A flowchart-style drawing of the algorithm described in [6] appears in Figure 5.3. Note that although three different multiplier operand recoding schemes are presented by Vazquez *et al.* and shown in the flowchart, a designer implements only one scheme. The flow of this multiplier is similar to that described in the previous subsection for the design of Lang *et al.* [5]. One fundamental difference, however, is the use of binary CSAs for the partial product accumulation. This is detailed in the next subsection.

The algorithm starts with the reading of the operands, after which, the multiplier operand is recoded according to the chosen scheme, as shown in Table 5.3. The *Radix-5* recoding, as the authors named it, uses a decimal carry-propagate adder to develop

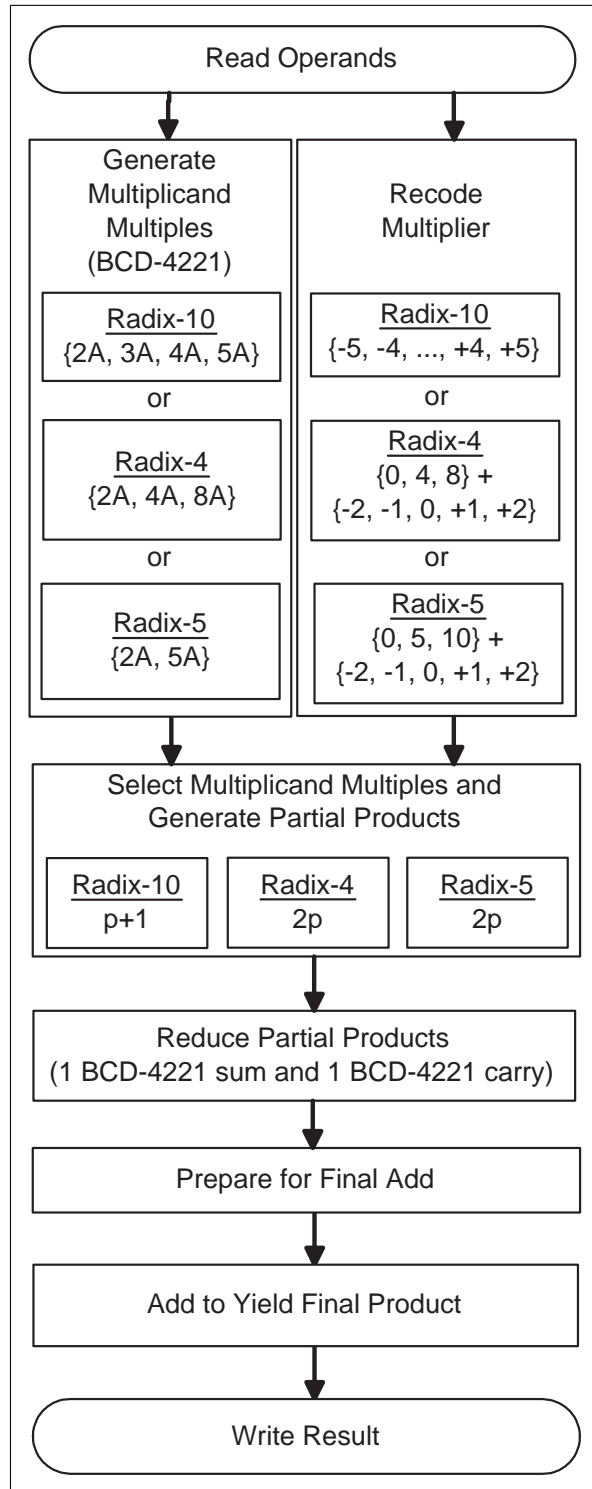


Figure 5.3: Flowchart of Parallel DFXP Multiplier Using Binary CSAs [6]

the 3A multiplicand multiple, while the multiplicand multiples needed for the other recoding schemes can be generated relatively quickly without carry propagation. In parallel and in accord with the chosen multiplier recoding scheme, the multiplicand multiples are produced. This portion of the algorithm appears in the blocks labeled “Generate Multiplicand Multiples” and “Recode Multiplier” in Figure 5.3.

Once the multiplicand multiples are generated and the multiplier operand is recoded, select signals are created to steer the appropriate multiples into a bank of XORs to selectively complement the multiples followed by a bank of binary CSAs to produce the partial products (see blocks “Select Multiplicand Multiples” and “Partial Product Generation”, respectively). There are $p + 1$ partial products with the *Radix-10* scheme and $2p$ partial products for the *Radix-5* and *Radix-4* schemes². The partial products then enter a partial product reduction tree comprised of binary CSAs, labeled as “Partial Products Reduction” in the flowchart figure. Emerging from the partial product reduction tree are two vectors, which are first converted to BCD-8421 and biased (“Prepare for Final Add” block), and then added (“Add to Yield Final Product” block) before the result is written.

Features

The most novel aspect of [6] is the use of binary CSAs for the partial product reduction. This is made possible by generating the multiplicand multiples in the BCD-4221 encoding (see Section 3.1). BCD-4221 allows the use of binary addition within each decimal digit position, since all 16 combinations of this four-bit code are valid decimal numbers. Further, when using binary CSAs, the emerging sum vector can be used directly. The only complication is in the generation of the carry

²Vazquez, *et al* chose the terms *Radix-4*, *Radix-5*, and *Radix-10* to describe the different recoding schemes. Typically in this context, the term *radix* is used to imply the number of partial products.

Table 5.3: Multiplier Operand Digit Recoding Schemes [6]

Multiplier Digit (decimal value of b_i)	Radix-10 ([5, 40]) Secondary Multiples		Radix-4 Secondary Multiples		Radix-5 Secondary Multiples	
	b'_i	+	b''_i	b'_i	+	b''_i
0	0	+	0	0	+	0
1	0	+	1	0	+	1
2	0	+	2	0	+	2
3	0	+	3	4	+	-1
4	0	+	4	4	+	0
5	10	+	-5 [†]	4	+	1
6	10	+	-4	4	+	2
7	10	+	-3	8	+	-1
8	10	+	-2	8	+	0
9	10	+	-1	8	+	1

[†] ($10 + -5$) is chosen over ($0 + 5$) to absorb any possible increment

vector emerging from the binary CSA, as these bits need to be decimal doubled. Decimal digit doubling is not as straightforward as simply left shifting by one bit as is done to realize binary doubling. Vazquez *et al.* observed that by converting the carry output of the binary CSA from BCD-4221 to BCD-5211 through a simple operation, and then left shifting by one bit, decimal doubling is achieved. A final useful benefit of having the multiplicand multiples comprised of BCD-4221 digits is that this encoding is self-complementing, which is desirable when creating negative multiplicand multiples.

For a $p = 16$ digit multiplier, there are 32 partial products for the *Radix-4* and *Radix-5* multiplier operand recoding schemes and 17 partial products for the *Radix-10* scheme. An additional partial product of $1A$ is needed for the *Radix-10* scheme when the most significant digit of the multiplier operand is greater than five. In this

case a negative partial product is chosen for this digit position, and the next more significant digit is incremented (i.e., zero to one).

The 17 partial products for the *Radix-10* scheme are reduced as shown in Figure 5.4. (To see the reduction trees for the *Radix-4* and *Radix-5* multiplier operand recoding schemes, the reader is referred to [6].) Due to the aligning of the partial products based on their respective weights, the number of decimal digits to be reduced varies from 2 to $p + 1$. For a $p = 16$ design, 7 levels of binary CSAs and three decimal doublers are traversed in the worst-case column. After the last level of binary CSAs, the intermediate product is in carry-save form as a vector of BCD-4221 sum digits and BCD-4221 carry digits, the carry digits having yet to be decimal doubled. A value of six is added to each sum digit while it is converted to BCD-8421. In parallel, the BCD-4221 carry digits are converted to BCD-5421 (see Table 3.1) such that left shifting each digit by one bit yields properly weighted digits in the BCD-8421 encoding. The two vectors of BCD-8421 digits are then added using a 128-bit wide binary quaternary tree modified to support decimal digits [114]. This adder is described in Section 3.2.

Implementation and Analysis

The authors of [6] estimated area and delay based on logical effort for their designs and the design of Lang *et al.* [5]. Their estimates of the parallel DFXP multiplier using the *Radix-10* recoding scheme show a 22% reduction in delay and a 42% reduction in area over the design presented by Lang *et al.* In Table 5.4, latency, throughput, area, and delay information is presented for the iterative DFXP multiplier of [39], the parallel DFXP multiplier of [5], and a multiplier implemented by Hickmann *et al.* [8] which is based on the design using the *Radix-10* recoding scheme as described

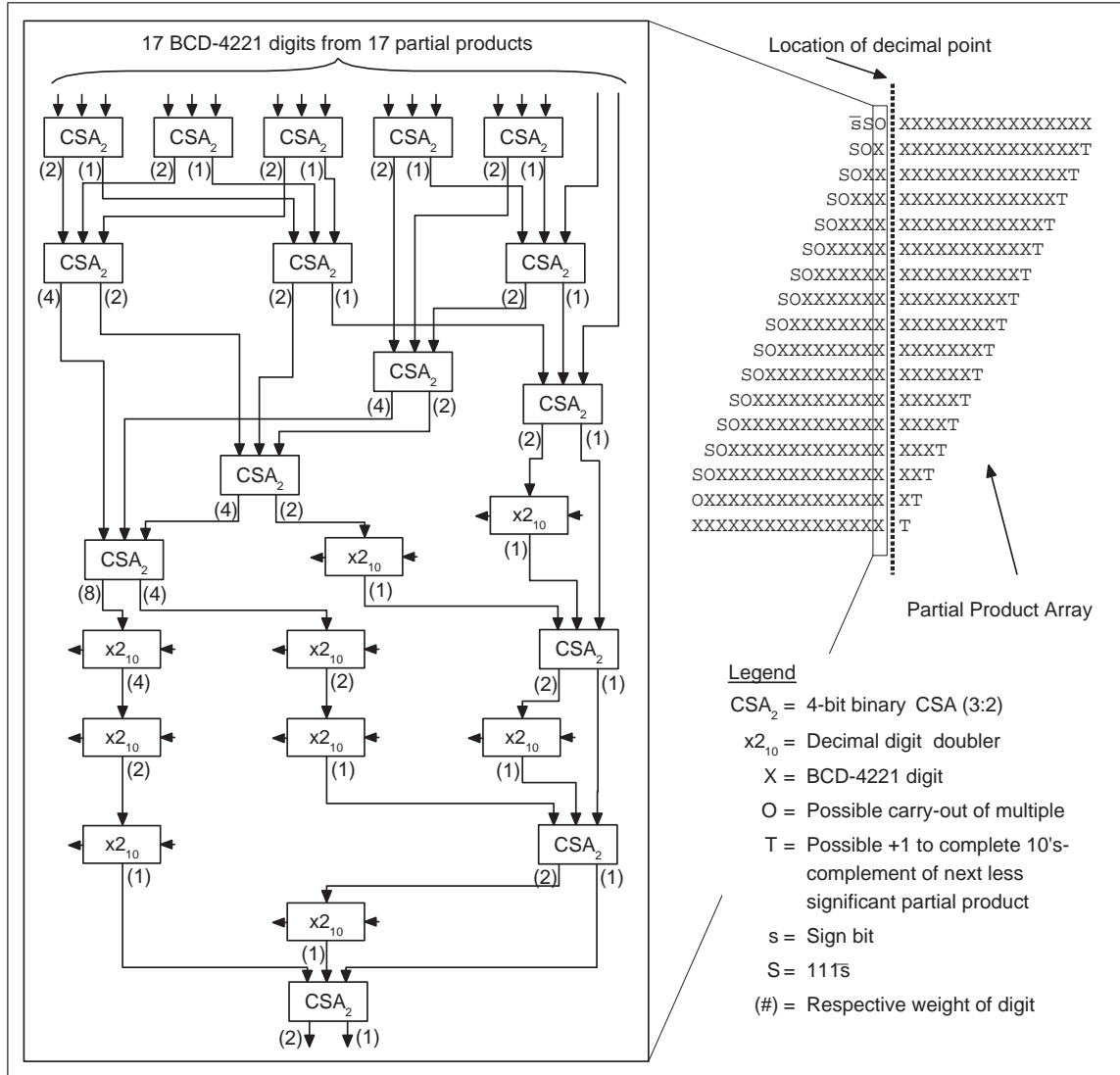


Figure 5.4: Partial Product Reduction Tree: *Radix-10* Recoding, Binary CSAs [6]

by Vazquez *et al.*. Though the area and delay numbers for the Lang *et al.* vs. the Vazquez *et al.* designs are not as different as estimated in [6], the latency difference is very significant. Note the area and cell count comparison between the two parallel designs is not entirely legitimate as their latencies are different. Further, the design of Vazquez *et al.* was estimated using logical effort which can often underestimate

²Some differences exist between the implementations, as described in Section 5.2.

the physical overhead of wiring, as they acknowledge in [6].

Table 5.4: Area and Delay of DFXP Multipliers (Iterative vs. Parallel)

$p = 16$	Iterative	Parallel	
	(Dec. CSAs) [39]	(Dec. CSAs) [5]	(Bin. CSAs) [6]
Latency (cycles)	20	11	8
Throughput (ops/cycle)	1/17	1	1
Cell count	59,234	180,000 [†]	182,791
Area (μm^2)	119,653	360,000 [‡]	369,238
Delay (ps)	810	810 [‡]	840
Delay ($FO4$)	14.7	14.7 [‡]	15.3

[†] Approximated

[‡] Scaled

Summary

The designs of Vazquez, Antelo, and Montuschi offer three different multiplier operand recoding schemes to enable a reduced set of secondary multiples, reduce the partial products down to an intermediate product in decimal carry-save form (two, four-bit digits) through binary CSAs, and then perform a final addition through a bias and correction addition scheme. The novelty of Vazquez *et al.*'s designs is in overloading binary CSAs to support the carry-save addition of decimal digits, this being facilitated by converting the BCD digits into BCD-4221 digits.

5.1.3 Summary of Parallel DFXP Designs

The parallel DFXP multipliers developed by Vazquez *et al.* have a performance and area advantage over the design developed by Lang *et al.*, in its current form. The

advantages are due to the use of binary CSAs in the partial product reduction tree. Binary CSAs are smaller and faster than decimal CSAs. And although decimal doubling on the carry digits is needed at various points during the reduction, a correction is not needed after each level of CSAs. Further, because binary CSAs are employed, this architecture can be extended to support binary multiplication [6].

For these reasons, a design developed by Vazquez *et al.* was chosen over the design by Lang *et al.* to be extended to support DFP multiplication. Namely, the *Radix-10* multiplier recoding scheme was selected. Even though this scheme uses a p -digit decimal carry-propagate adder to produce the multiplicand triple, this addition can be accomplished with less hardware and in a similar delay as it takes to reduce $2p$ partial products to roughly $p + 1$ partial products. The description of the parallel DFP multiplier design appears in the following section.

5.1.4 Combined Binary/Decimal, Fixed-point Design

As mentioned in the last subsection, the use of binary CSAs for the reduction of partial products [6] enables the partial product reduction tree to support both binary data and decimal data. As described in [6], the sum bits emerging from the binary CSA can be used directly regardless of whether the input is a binary integer or a decimal digit. However, the carry bits emerging from the binary CSA must be doubled, and this doubling is different depending on the input data type. Doubling a binary integer simply involves left shifting the bits by one. Whereas, doubling a decimal digit in BCD-4221 effectively involves the conversion of the digit into BCD-5211 followed by left shifting the bits by one. A control signal indicating if the data are binary or decimal is needed to control the behavior of the doubling circuitry.

In [7], Hickmann³, Schulte, and I describe several notable improvements over the

³lead author

combined binary *Radix-4*/decimal *Radix-5*, fixed-point multiplier design presented in [6]. Specifically, improving the delay of the combined binary/decimal doubling circuit, improving the delay of the combined binary/decimal partial product reduction tree by removing all instances of 4:2 compressors, and improving the delay of the binary datapath in the partial product reduction tree by splitting the combined binary/decimal partial product reduction tree into a dedicated binary reduction tree and a dedicated decimal reduction tree just before the first required instance of a combined binary/decimal doubler.

Algorithm

A flowchart-style drawing of the algorithm described in [7] appears in Figure 5.5. The algorithm for this design is the same effectively as that presented in Section 5.1.2. The algorithm starts with the reading of the operands, after which, the multiplier operand, when binary, is recoded according to the Booth *Radix-4* scheme [142] as shown in Table 5.5 or, when decimal, is recoded according to the *Radix-5* scheme as shown in Table 5.3. Thus, the binary multiplicand multiple set of 1A, 2A, 4A, 8A and the decimal multiplicand multiple set of 1A, 2A, 5A, 10A are needed. An example showing the recoding for a 16-bit binary/4-digit decimal multiplier operand appears in Figure 5.6. Note that two multiplicand multiples are chosen for every four binary bits, regardless of whether the operands are binary or decimal data. In parallel with the multiplier operand recoding, the multiplicand multiples are created in BCD-4221 form. Once the multiplicand multiples are generated and the multiplier operand is recoded, select signals are created to steer the appropriate multiples into a bank of XOR gates to selectively complement the multiples, followed by a bank of binary CSAs to produce the partial products. The $2p + 1$ partial products are then reduced via a reduction tree comprised of binary CSAs. Emerging from the partial product

reduction tree are two vectors which are first converted to BCD-8421, biased, and then added, before the result is written.

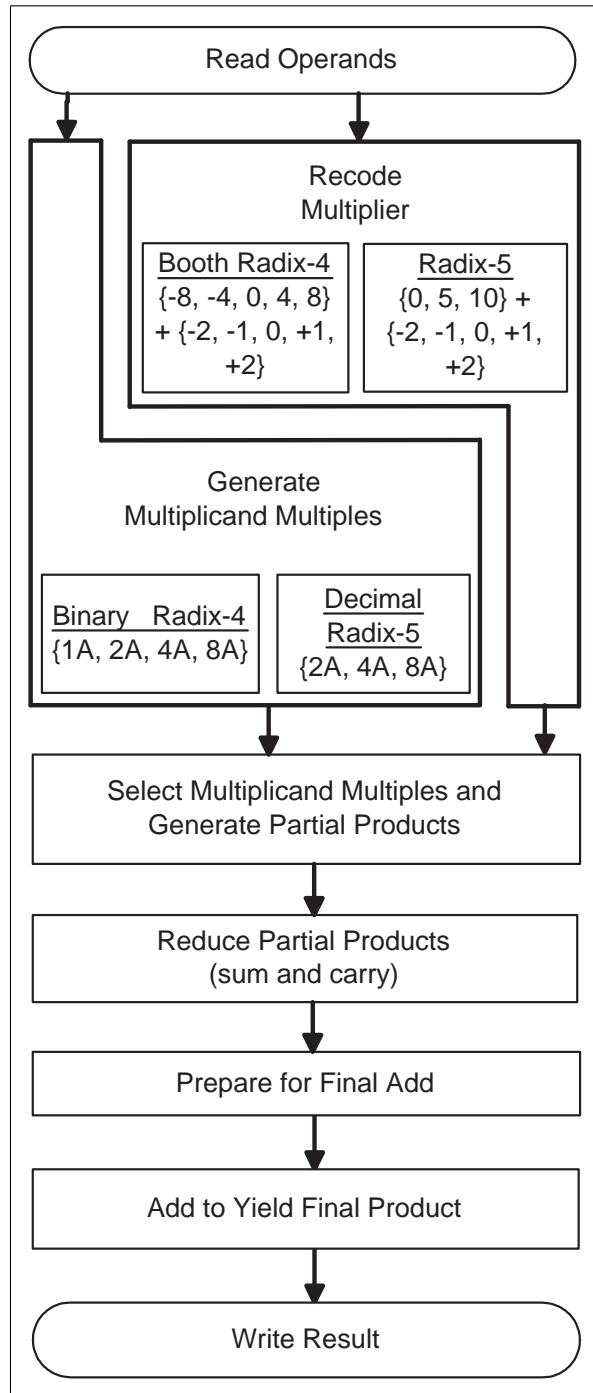


Figure 5.5: Flowchart of Parallel BFXP/DFXP Multiplier Using Binary CSAs [7]

Table 5.5: Binary Multiplier Operand Booth *Radix-4* Recoding Scheme

Multiplier Bits (overlapping triplet)	Recoded Four Bits		
	b'_i	+	b''_i
0 0 0	0	+	0
0 0 1	4	+	1
0 1 0	4	+	1
0 1 1	8	+	2
1 0 0	-8	+	-2
1 0 1	-4	+	-1
1 1 0	-4	+	-1
1 1 1	0	+	0

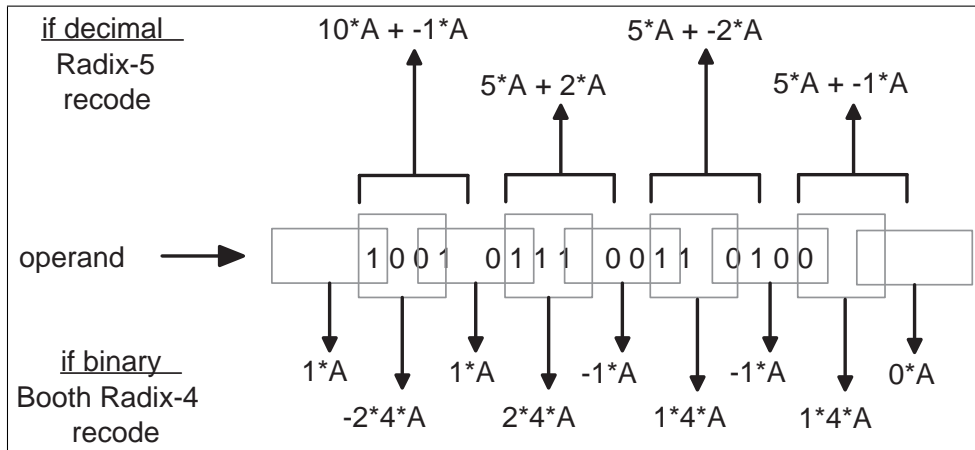


Figure 5.6: Binary/Decimal Multiplier Operand Recoding Example [7]

Features

One delay bottleneck in a combined binary/decimal partial product reduction tree based on decimal data being in BCD-4221 form, is that of the combined four-bit binary/decimal doubler. In [6], when performing binary multiplication, the doubler

simply left shifts the incoming four bits by one bit position. When performing decimal multiplication, the doubler first converts the BCD-4221 data to BCD-5211 and then left shifts the converted data. Thus, a multiplexor operation is in the critical path. By integrating the multiplexor operation into the generation of the left-shifted four-bit binary value and the left-shifted recoded BCD-4221 digit, a single set of shared outputs can be produced with no intermediate steps. Thus, the recoding, shifting, and multiplexing of the original approach [6] occurs concurrently, which significantly reduces the delay. This modification yields a 63% reduction in the delay of the combined binary/decimal doubler when optimizing for delay and a slight area advantage when optimizing for area.

To further improve the delay of combined binary/decimal partial product reduction, we eliminated the use of binary 4:2 compressors, as these used two combined binary/decimal doublers, which significantly slow down the binary datapath. Instead, we used binary 3:2 counters and combined binary/decimal doublers exclusively. Figure 5.7 shows the improved combined binary/decimal partial product reduction tree for the worst-case column containing 33 partial products. Across the partial product reduction tree, using the improved combined binary/decimal doubler and replacing the 4:2 compressors with 3:2 counters this improvement leads to a 29% reduction in area and a 3% reduction in delay compared to the original approach [6].

A final improvement has as its aim the reduction of delay in the binary datapath. By splitting the partial product reduction tree into a dedicated binary reduction tree and a dedicated decimal reduction tree just before the first required instance of a combined binary/decimal doubler, optimized data-specific doublers can be used. For the binary data, doubling is achieved by hard-wired left-shifts. And for the decimal data, doubling is achieved through simplified decimal-only doublers which do not have a multiplexor function. Figure 5.8 shows the split binary/decimal partial

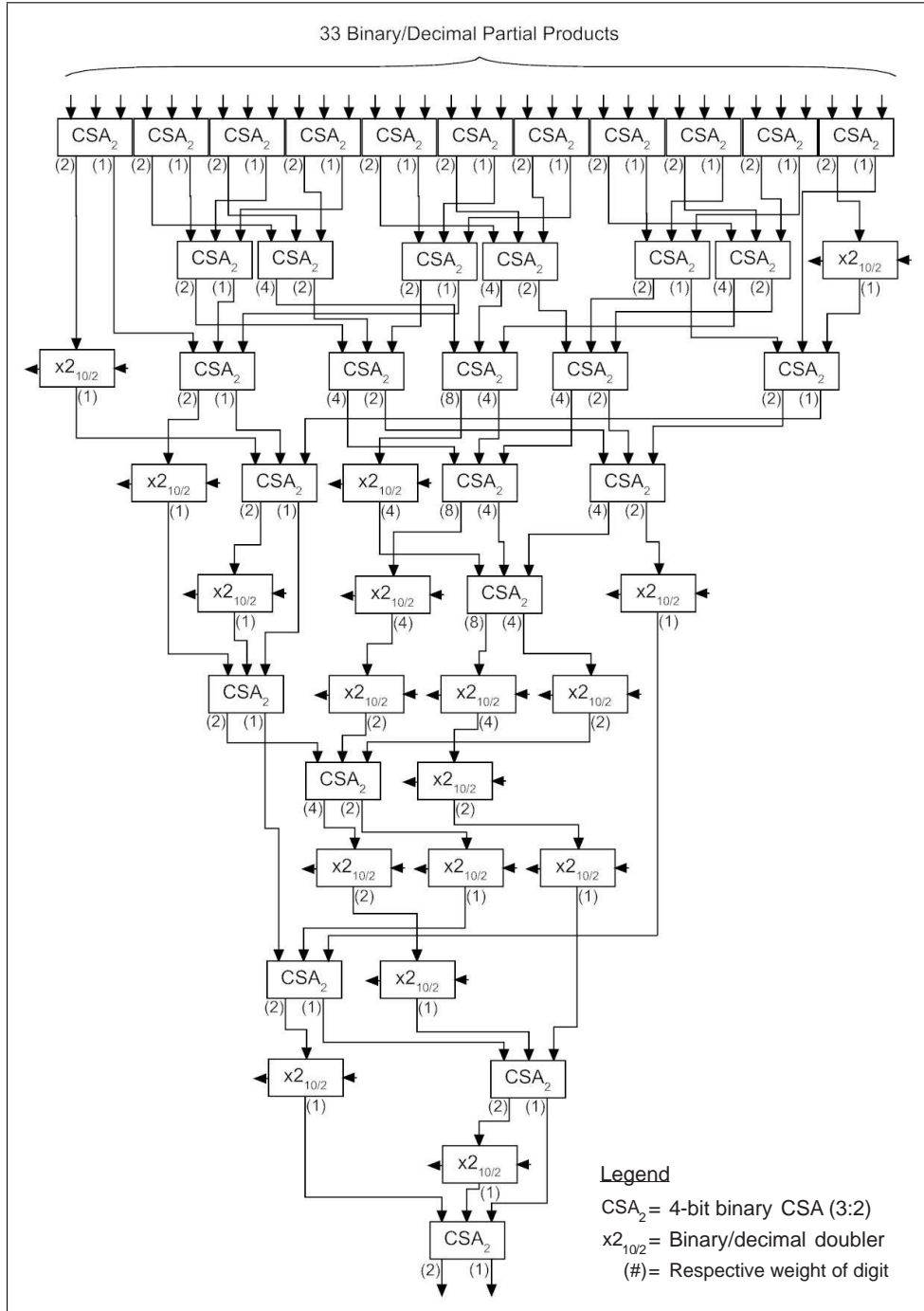


Figure 5.7: Combined Bin/Dec Partial Product Reduction Tree (33 Products) [7]

product reduction tree for the worst-case column containing 33 partial products. Across the partial product reduction tree, this improvement, in conjunction with the improved combined binary/decimal doubler and replacement of 4:2 compressors with 3:2 counters in the decimal path, leads to a 19% reduction in area and a 42% reduction in the delay of the binary path compared to the original approach [6]. The delay in the decimal path is increased by 1%.

There is an optimization to the last improvement presented in [7] which may prove useful for DFP multiplication. Namely, changing the supported operand width for binary data from 63 bits to 53 bits, which means fewer binary partial products (27) need be reduced. Therefore, a simpler partial product tree can be achieved which yields a 29% reduction in area and a 48% reduction in the delay of the binary path compared to the original approach [6]. The delay in the decimal path is increased 3%.

Implementation and Analysis

RTL models for the original combined BFXP/DFXP multiplier design by Vazquez *et al.* along with the various improved designs were coded in Verilog and verified using a suite of over 500,000 random testcases. The models were then synthesized using the Synopsys Design Compiler into TSMC's 65nm CMOS standard cells. Table 5.4 shows the cycle latency and area for the various fixed-point multiplier configurations described in [7]. Cycle latency is shown as opposed to delay because all the designs were pipelined for a cycle time of 500ps.

Summary

Several improvements over the combined BFXP/DFXP multiplier design in [6] are presented in [7] and summarized here. These improvements include a reduction

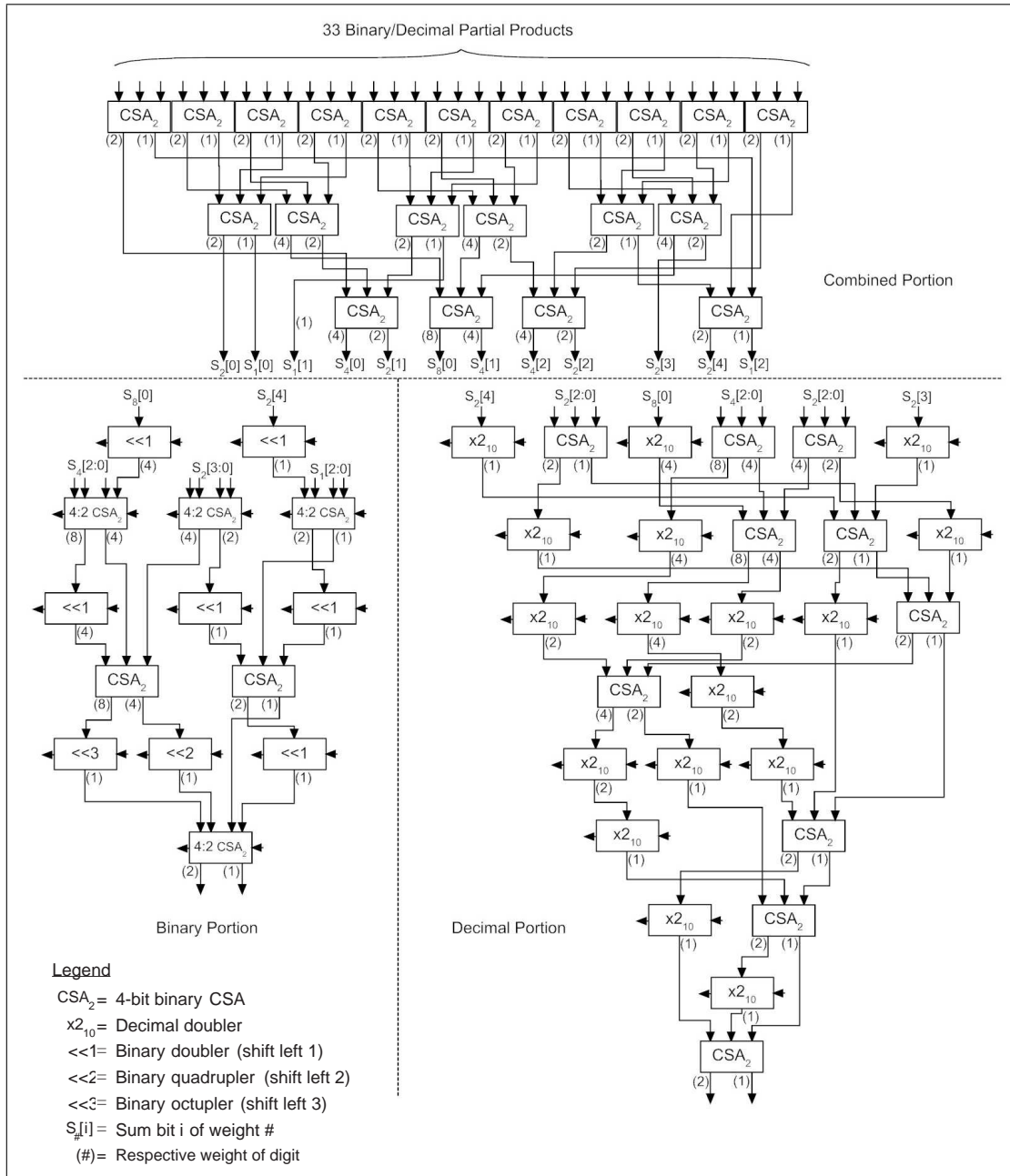


Figure 5.8: Split Bin/Dec Partial Product Reduction Tree (33 Products) [7]

in the delay of the combined binary/decimal doubling circuit, a reduction in the delay of the combined binary/decimal partial product reduction tree by removing all instances of 4:2 compressors, and a reduction in the delay of the binary datapath of

Table 5.6: Area and Delay of Various Parallel DFXP Multipliers [7]

Multiplier Design	Latency		Area	
	Bin	Dec	μm^2	Ratio
64-bit Radix-4 Binary	5	-	76,973	-
53-bit Radix-4 Binary	5	-	53,724	-
16-dig Radix-5 Decimal [6]	-	8	99,911	-
64-bit/16-dig Baseline	5	8	176,884	1.00
53-bit/16-dig Baseline	5	8	153,635	1.00
Original 64-bit/16-digit [6]	8	8	135,184	0.76
Improved 64-bit/16-digit [7]	8	8	101,229	0.57
Split 64-bit/16-digit [7]	5	8	112,952	0.64
Split 53-bit/16-digit [7]	5	8	104,789	0.68

the partial product reduction tree by splitting the combined binary/decimal partial product reduction tree into a dedicated binary reduction tree and a dedicated decimal reduction tree just before the first required instance of a combined binary/decimal doubler. Additionally, a promising design point to support combined BFP/DFP multiplication is also presented.

5.2 Floating-point Design

The DFP multiplier described in this section is based on the parallel DFXP multiplier of [6] that utilizes the *Radix-10* multiplier operand recoding scheme. It is slightly different from [6] in that the final adder uses a high-speed direct decimal carry-propagate adder [115] employing a Kogge-Stone carry network. This research, presented in [8], is believed to be the first parallel DFP multiplier design in compliance with IEEE 754-2008.

5.2.1 Algorithm

A flowchart-style drawing of the parallel DFP multiplication algorithm is shown in Figure 5.9, with the steps of the parallel DFXP multiplier surrounded by a dashed rectangle. As with the iterative DFP multiplier described in 4.2, the operation begins with the reading and decoding of the operands (block “DPD Decode Operands”). However, after the initial decoding is performed, the multiplicand is recoded from BCD-8421 into BCD-4221 while the multiplier operand’s digits are *Radix-10* recoded into $\{-5, \dots, +5\}$. The recoding of the p -digit multiplier operand into this digit set is chosen as this approach yields only $p + 1$ partial products, as opposed to $2p$ partial products with the alternative multiplier recoding schemes presented in [6].

With the multiplicand in BCD-4221 form, the double, triple, quadruple, and quintuple are generated. The recoding of the multiplier and the generation of the multiplicand multiples are shown in the block labeled “Recode Multiplier and Generate Multiples”. The recoded multiplier digits are used to select $p + 1$ partial products from the multiple set $\{C^A, 2C^A, 3C^A, 4C^A, 5C^A\}$ ⁴ and their complements. The partial products are then presented in parallel to the partial product reduction tree (see the

⁴Note, C^A is used instead of A as the discussion regards floating-point numbers. C^A represents the significand of the floating-point operand A .

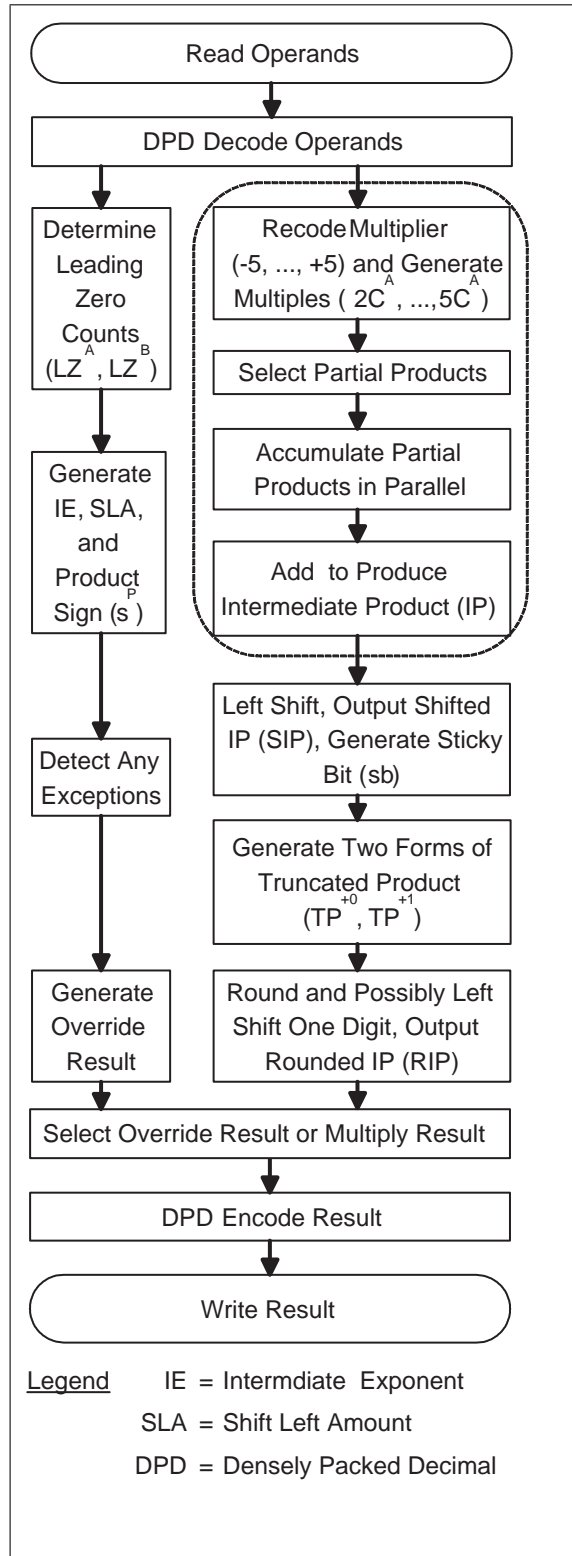


Figure 5.9: Flowchart of Parallel DFP Multiplier Using Binary CSAs [8]

“Accumulate Partial Products...” block). For a $p = 16$ design, a binary CSA tree layout for the worst-case of 17 decimal digits is shown on the left in Figure 5.4. The tree is based on the structure presented in [6], but modified to add an additional $1C^A$ partial product when the multiplier operand’s $MSD > 5$. After all the partial products are reduced into carry and sum vectors, a carry-propagate adder is used to produce a non-redundant intermediate result equal in length to the sum of the operands’ digits. This addition step appears in the block labeled “Add” in Figure 5.9.

In parallel with the multiplicand multiple generation and partial product reduction, the shift-left amount, SLA , and intermediate exponent of the intermediate product, IE^{IP} , are calculated for shifting the intermediate product, IP , to fit into p digits of precision. These calculations are performed in the same manner described for the iterative DFP multiplier (Section 4.2.2); namely, using the operands’ leading zero counts in order to estimate the number of significant digits in the result. In addition to the SLA and IE^{IP} values, the sign bit of the final result is calculated, and exception conditions are detected. These steps occur in the “Determine Leading Zero Counts” and “Generate IE, SLA, and Product Sign” blocks of Figure 5.9.

The IP emerging from the partial product accumulation tree is then left-shifted by the SLA amount, forming the shifted intermediate product, SIP . Next, the sticky bit is produced from the fractional product, FRP , which resides in the less significant half of the $2p$ -digit shifted intermediate product register (block “Left Shift...”). In parallel, the truncated product, TP , is incremented to allow the rounding logic to select between TP^{+0} and TP^{+1} , which, as described previously, is sufficient to support all rounding modes (see Section 4.2.2). The production of TP^{+0} and TP^{+1} occurs in the “Generate Two Forms...” block in Figure 5.9.

Finally, the rounding and exception logic, based on the rounding mode and exception conditions, selects between TP^{+0} , TP^{+1} , and special case values to produce

the rounded intermediate product, RIP . The RIP is then encoded in DPD and put in IEEE 754-2008 format with the appropriate flags set to produce the final product, FP . The exception handling and final product generation logic are distributed over the “Detect Any Exceptions”, “Generate Override Result”, “Round and Possibly Left Shift...”, and “Select Override Result or Multiply Result” blocks of Figure 5.9. In the following subsections, the components and functions distinct from the iterative DFP multiplier design of Section 4.2 are described.

5.2.2 Features

Figure 5.10 depicts the parallel DFP multiplier design. Note that as the register transfer level (RTL) model for this design was written without storage elements to take advantage of an auto-pipelining feature of the Synopsys Design Compiler, the actual placement of storage elements throughout the dataflow may differ slightly from those shown in the figure. More information on the RTL model and auto-pipelining appears in Section 5.2.3.

To support DFP, the intermediate product, IP , emerging from the carry-propagate adder in non-redundant form, enters a left shifter to produce the shifted intermediate product, SIP . For a $p = 16$ digit design, this shifter is 32 digits wide. In the iterative DFP design, when the intermediate exponent of the intermediate product is less than E_{min} , the control logic allows the iterative portion of the algorithm to continue, right-shifting IP in an effort to bring the IE^{IP} into range. Thus, gradual underflow is supported in hardware with no change to the dataflow circuitry. In the parallel DFP design, however, supporting gradual underflow requires a modification to the dataflow in the form of expanding the function of the left shifter to both a left and right shifter.

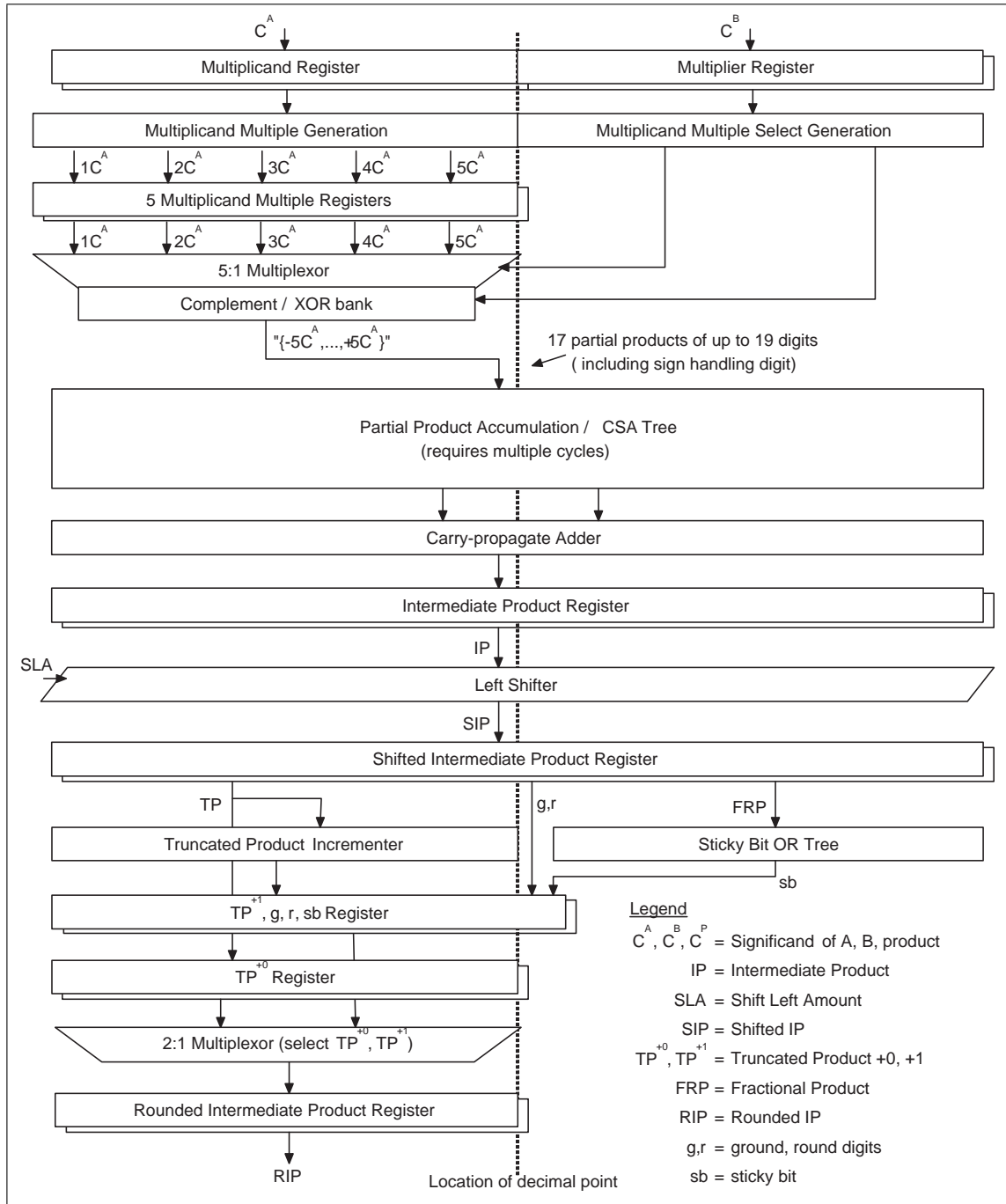


Figure 5.10: Parallel DFP Multiplier Design [8]

The shifter is controlled by the shift left amount, SLA (see Equation 4.26). The more significant half of the SIP , called the truncated product or TP , is incremented

via a 16 digit decimal incremter while the less significant half of the *SIP*, called the fractional product or *FRP*, is used to produce the sticky bit, *sb*, via a 56 bit OR tree⁵. Using the rounding scheme presented in Section 4.2.2, the rounding logic produces the rounded intermediate product, *RIP*, by selecting between the non-incremented truncated product, TP^{+0} , and its incremented value, TP^{+1} , and possibly concatenating the guard digit from the *FRP* or its incremented value.

The parallel ORing of the $p - 2$ digits *FRP* is in contrast with the iterative DFP multiplier design which developed the sticky bit on-the-fly. Thus, in the parallel DFP multiplier, the sticky counter, *SC*, is not needed. A method to pre-calculate the sticky bit in parallel with the fixed-point multiplier was considered. However, this method did not reduce the worst-case delay, and it increased the area.

Performing the shift operation before the carry-propagate adder (CPA) in the fixed-point multiplier was examined but was not implemented due to the following tradeoffs. The primary benefits of using a smaller p -digit CPA after the shifter and the potential to combine the final addition and rounding are outweighed by several negative factors. First, the calculation of the sticky bit from a redundant carry-save representation requires roughly twice as many gates to perform the calculation, increasing the area of the rounding logic. Second, a separate carry tree is required to calculate any possible carry out of the redundant representation of the FRP along with more complicated rounding logic to handle this possible carry into the round digit during result selection. It also requires the generation of TP^{+2} , in addition to TP^{+1} and TP^{+0} , to handle the case of both a round-up and carry into the round digit. Third, shifting prior to adding requires two $2p$ -digit shifters, instead of one, as the intermediate product has sum and carry vectors comprised of four-bit digits. Since the fixed-point multiplication dominates the latency of the DFP multiplication,

⁵The OR tree is not 64 bits as the two leading digits are the guard and round digits.

the small delay benefit of performing the shift earlier is considered to be outweighed by the additional area overhead of the $2p$ -digit shifter and larger sticky calculation logic. For these reasons, the shifter was placed after the CPA as shown in Figure 5.10.

Other than not supporting gradual underflow, the exception detection and handling is the same in the parallel DFP design as it is in the iterative DFP design. In the parallel DFP design, the detection of $IE^{IP} < Emin$ simply leads to the raising of an output flag to inform the system that some other mechanism is needed to calculate the subnormal result. The motivation behind this decision is the considerable savings in area and delay obtained by removing this feature from the hardware implementation. In the iterative DFP multiplier, it is straightforward to realize the necessary right shifting of the intermediate product. However, in the parallel DFP multiplier, right-shifting the intermediate product requires the replacement of the left shifter with a right-left shifter. In support of this, a shift right amount, SRA , is needed to control the shifter and to increase the intermediate exponent of the shifted intermediate product, IE^{SIP} . The equations for SRA and IE^{SIP} follow, where $IE^{IP} < Emin$.

$$SRA = \min((Emin - IE^{IP}), p + 2)$$

$$IE^{SIP} = IE^{IP} + SRA$$

Completion of the multiply operation involves the same steps as in the iterative DFP multiplier, namely, selecting between RIP and a special value, such as QNaN or infinity, and producing the final result in DPD encoded form. A QNaN result is warranted when the multiply operation is $0 \times \text{inf}$ or if one or both operands is NaN. If an operand is NaN, that operand's NaN payload is used when forming the result. Instead of supporting alternative paths through the dataflow, the control logic passes the NaN value through the dataflow by multiplying it by 1 (i.e., coercing the other

operand). The rounding logic ensures the passed value does not get incremented or shifted.

5.2.3 Implementation and Analysis

Register transfer level models of both the presented 64-bit (16-digit) parallel DFP multiplier [8] and its predecessor parallel DFXP multiplier [6] were coded in Verilog. Both designs were synthesized using LSI Logic's gflxp 0.11um CMOS standard cell library and the Synopsys Design Compiler. To validate the correctness of the designs, over 500,000 testcases were simulated successfully on the DFXP model (pre- and post-synthesis versions) and over 500,000 testcases covering all rounding modes and exceptions were simulated successfully on the DFP model (pre- and post-synthesis versions). Publicly available tests include directed-random testcases from IBM's FP-gen tool [140] and directed tests at [141]. Table 5.7 contains area and delay estimates for the DFP multiplier design presented and its predecessor DFXP multiplier design. The values in the *FO4 Delay* column are based on the delay of an inverter driving four same-size inverters being *55ps*, in the aforementioned technology. The critical path in the parallel DFP multiplier [8] is in the stage with the 128-bit shifter, while for the parallel DFXP multiplier it is within the partial product reduction tree.

The entries in Table 5.7 for the parallel DFP multiplier were chosen from one of multiple synthesis jobs using Synopsys Design Compiler's auto-pipelining feature. This feature provides area and delay results for design implementations of various pipeline depths. With this information, one can more readily examine the costs associated with a desired design point. Table 5.8 shows the area and delay information for various pipeline depths as computed from multiples synthesis runs using this auto-pipelining feature. One slight drawback of auto-pipelining is that there are no latches

Table 5.7: Area and Delay of Parallel Multipliers (DFXP vs. DFP)

$p = 16$	Parallel (Binary CSAs)	
	DFXP [6]	DFP [8]
Latency (cycles)	8	12
Throughput (ops/cycle)	1	1
Cell count	182,791	433,957
Area (μm^2)	369,238	876,593
Delay (ps)	840	820
Delay ($FO4$)	15.3	14.9

Table 5.8: Area and Delay vs. Pipeline Depth of Parallel DFP Multiplier

$p = 16$ Pipeline Depth	Delay		Area	
	ps	FO4	Cells	μm^2
0 (combinational)	4600	83.64	323,488	653,445
1 (latched outputs)	4470	81.27	350,042	707,084
2	2470	44.91	357,807	722,770
3	1880	34.18	381,630	770,892
4	1560	28.36	359,544	726,278
5	1310	23.82	371,620	750,672
6	1180	21.45	392,807	793,470
7	1090	19.82	405,797	819,790
8	1050	19.09	389,687	787,167
9	940	17.09	401,609	811,250
10	890	16.18	412,725	833,704
11	870	15.82	436,114	880,950
12	820	14.91	433,957	876,593

on the inputs. Thus, the area numbers are slightly optimistic when compared to the iterative DFP multiplier's numbers.

5.2.4 Summary

The parallel DFP multiplier presented in this subsection combines the DFP extensions developed for the iterative DFXP multiplier design of [39] to a slightly altered parallel DFXP multiplier design of [6]. The resultant parallel DFP multiplier is compliant with IEEE 754-2008. In the next section, this parallel DFP multiplier [8] is compared with an iterative DFP multiplier [41].

Table 5.9: Area and Delay of Multipliers (Iterative vs. Parallel, DFXP vs. DFP)

$p = 16$	Iterative (Decimal CSAs)		Parallel (Binary CSAs)	
	DFXP [39]	DFP [41]	DFXP [6]	DFP [8]
Latency (cycles)	20	25	8	12
Throughput (ops/cycle)	1/17	1/21	1	1
Cell count	59,234	117,627	182,791	433,957
Area (μm^2)	119,653	237,607	369,238	876,593
Delay (ps)	810	850	840	820
Delay ($FO4$)	14.7	15.4	15.3	14.9

5.3 Analysis of Iterative and Parallel Designs

Having presented multiple DFXP and DFP hardware multipliers, this section aims to compare the iterative DFXP design of Section 4.1.1, the iterative DFP design of Section 4.2, the parallel DFXP design of Section 5.1.2, and the parallel DFP design of Section 5.2. As mentioned in each multiplier’s respective section, register transfer level models were coded in Verilog for all four 64-bit (16-digit) iterative and parallel DFP multipliers [8, 41], and their predecessor DFXP multipliers [6, 39]. To make the comparison between the iterative and parallel designs as balanced as possible, the iterative design was converted so as *not* to support gradual underflow. This alteration has no effect on the critical path and virtually no change in area. The designs were synthesized using LSI Logic’s gflxp 0.11 μm CMOS standard cell library and Synopsys Design Compiler Y-2006.06–SP1. To validate the correctness of the design, over 500,000 testcases covering all rounding modes and exceptions were simulated successfully on the designs, both pre- and post-synthesis. Publicly available tests used for validation include directed pseudo-random testcases from IBM’s FPgen tool [143] and directed tests available at [141].

Table 5.9 contains area and delay estimates for the DFP multiplier designs and

their predecessor DFXP multiplier designs. The values in the *FO₄ Delay* column are based on the delay of an inverter driving four same-size inverters having a 55ps delay, in the aforementioned technology. The parallel DFXP and DFP multipliers have eight and twelve pipeline stages, respectively, to achieve critical path delays that are comparable to the iterative multiplier designs. The critical path in the iterative DFP multiplier is in the stage with the 128-bit barrel shifter, while for the iterative DFXP multiplier it is the decimal 4:2 compressor. As for the parallel DFP multiplier, the critical path is the 128-bit shifter, while for the fixed-point portion it is within the partial product reduction tree. The critical paths identified in the parallel multipliers are with respect to the implementations shown in Table 5.8.

According to Table 5.9, the amount of logic necessary to extend the parallel DFXP multiplier to support DFP is significantly more than the amount necessary to extend the iterative DFXP multiplier to support DFP. Though one might expect the deltas in area to be relatively similar, there are several reasons why they are quite different. First, as the throughput of the parallel multiplier is one, there is a significant amount of information which needs to be moved from pipe stage to pipe stage. Information such as the shift amount, intermediate exponent, exponent-related data (for exception detection and handling), rounding mode, and the sticky and sign bit. Second, there is logic which exists in the parallel DFP multiplier which does not exist in the iterative DFP multiplier. Namely, the carry-propagate adder is twice as wide in the parallel design, the shifter is a circular shifter as opposed to a left shifter, a shift right amount must be developed to support gradual underflow, and there is an OR tree to generate the sticky bit. Third, there may be some inefficiency in the auto-pipelining feature of the synthesis tool, which was only applied to the parallel multiplier.

As can be seen in Table 5.9, the iterative DFP multiplier is significantly smaller. Further, the iterative DFP multiplier may achieve a higher practical implementa-

tion frequency as the latch overhead may be prohibitive when the partial product reduction tree of the parallel DFP multiplier is deeply pipelined. Thus, in situations when area is constrained or when cycle time is extremely aggressive, the iterative DFP multiplier may be an attractive implementation. However, the parallel DFP multiplier has less latency for a single multiply operation and is able to produce a new result every cycle. Therefore, in situations when area can be traded for latency and throughput, the parallel DFP multiplier may be an attractive implementation. As for power considerations, the fewer overall devices in the iterative multiplier, and more importantly the fewer storage elements, will result in less leakage in standby mode. This benefit is mitigated in operational mode as the higher latency and lower throughput may result in more power consumed for a given workload. A more thorough examination of the iterative and parallel DFP multiplier designs is presented in [42].

Chapter 6

Conclusion

This chapter presents a summary of my research performed in collaboration with others, my thoughts on related future research, and closing remarks.

6.1 Summary

Of my research contained in this dissertation, I start with a description of an iterative DFXP multiplier design. The algorithm and implementation of this design establish several notable decimal-specific features, such as decimal carry-save addition, and serve as a baseline for comparing area and performance of future multiplier designs. Then, I present a second iterative DFXP multiplier, which in contrast to the aforementioned design, develops the partial products on-the-fly. The algorithm and implementation of this design illustrate the benefit of operand recoding to reduce the number of multiplicand multiples. From this point, my research focuses on DFP multiplication.

I then present my research on adapting the iterative DFXP multiplier design based on decimal carry-save addition to support DFP multiplication as defined in IEEE 754-2008. The control portion of this design, which is used in subsequent research efforts, is derived and described. Further, a comparison is made between this iterative DFP multiplier and its predecessor fixed-point design with respect to area, delay, and latency.

The research of two teams into parallel DFXP multiplication are described for a thorough examination of the topic of decimal multiplication, and because I extend one of these designs to support DFP multiplication. I compare the area, delay, latency, and throughput of these two parallel DFXP multiplier designs and my earlier iterative DFXP multipliers. My research is then presented on extending the parallel DFXP multiplier based on binary carry-save addition to support DFP multiplication as defined in IEEE 754-2008. Finally, I compare the iterative and parallel DFXP multipliers and the iterative and parallel DFP multipliers with respect to area, delay, latency, and throughput. I offer several design considerations and trade-offs to be

weighed by those seeking to implement hardware support for decimal multiplication.

The iterative DFXP multiplier designs consume significantly less area than their parallel counterparts. Further, the iterative designs can achieve a higher practical operating frequency. This is because the introduction of additional pipeline stages into the parallel multiplier designs, particularly into the partial product accumulation tree, is much more costly in terms of latch count, and therefore, area. Thus, the iterative multipliers may be better suited for implementations in which area is deemed more important than throughput. However, because a mechanism has been developed to overload binary carry-save adders to support decimal addition, combined binary/decimal parallel multipliers can be implemented to mitigate the area overhead of a parallel design. Using a combined binary/decimal multiplier is most suitable for workloads which are not expected to compete for use of the binary and decimal multiplier resource. The same may be said for a combined binary/decimal adder.

The iterative DFXP multiplier designs consume significantly less wiring resource than their parallel counterparts. Thus, the iterative multipliers may be better suited for technologies which offer limited wiring layers or in which wiring is particularly expensive in terms of area and/or delay. As only one partial product in the iterative designs is produced at a time, the same wiring tracks can be used to send each partial product to the adder for accumulation. This is in contrast to the current parallel decimal multiplier designs which produce multiple partial products in parallel, along with negative partial products, and then distribute these vectors to multiple locations in the partial product accumulation tree. Some ideas to address this problem are discussed in Section 6.2.

As defined in IEEE 754-2008, the features and requirements of DFP arithmetic are similar to BFP arithmetic. At the implementation level, however, the following facts

are significant: 1) DFP operands and results are not normalized, and 2) decimal digit boundaries are every four or more bits, depending on the encoding, and not every bit. I explored the effect of these differences in developing both iterative and parallel DFP multiplier designs. It is clear that hardware unique to decimal is needed for exception detection and handling, as well as for rounding. Further, and fortunately, because of the large amount of similarity between DFP logic and BFP logic, a significant amount of hardware could be shared in a combined BFP/DFP execution unit.

6.2 Future Research

With the IEEE 754-2008 [20] now a standard and the recent announcements of DFP hardware support in IBM's server [23] and mainframe [25] microprocessors, it is anticipated other microprocessor manufacturers will consider DFP hardware implementations. Thus, there are several areas in which further research will be beneficial to those making the decision to add DFP hardware and to those deciding whether DFP hardware should continue to be offered or how those offerings should be modified.

For those considering DFP hardware implementations, the marketplace may very well provide the answer. If a sizable number of customers purchase systems with hardware support for DFP operations specifically to modify, compile, and run their applications, or if independent software vendors begin developing applications to take advantage of the DFP hardware, the manufacturers wishing to distinguish their processor may want to add DFP hardware instructions. Now that two platforms offer DFP hardware instructions, benchmarks and applications can be modified to take advantage of these instructions in order to examine the actual speedup. For those already offering hardware implementations, this scenario of more applications and improved benchmarks enables them to profile the performance of these applications to determine which operations should be added or have their latency reduced to speed up the program's operation.

This suggests additional research in the area of benchmark development and potential speedup may be needed. In commercial computing, where the need for DFP hardware is less contested or die space may be more readily available, further research is needed to determine which decimal operations yield the greatest speedup – specifically, which functions should have dedicated hardware, which functions should

be implemented in microcode or be hardware-assisted, and which functions should remain in software. Also, the distribution of data and results may be different from those of BFP workloads. Thus, as applications become more prevalent and benchmarks mature, the frequency of subnormals, NaNs, and exceptions need to be studied to determine whether these should be handled in hardware, microcode, or software.

From an architectural point of view, there are numerous research opportunities as well, such as implementing fused multiply-add, tagging the register file data, and combining DFP and BFP hardware into a single execution unit. Some BFP units support fused, multiply-add ($Y = (A \times C) + B$), as some program profiles indicate the result of a multiply operation is then often used in an addition operation. The need and benefit of this operation needs to be explored as the number and type of decimal applications increase. Tagging the register file data with information about its data may improve the latency of some operations and subsequently, program throughput. As an example of how tagging may be worthwhile, consider the cost associated with aligning operands for addition due to determining the difference between the exponents and the number of leading zeros in the larger operand. If the number of leading zeros in each operand were available at the time the operands are read from the register file, the addition operation could complete sooner. The determination of the number of leading zeros can be performed when the data are fetched from memory and/or when each result is produced. In addition to the leading zero count, additional information could be placed in each operand's tag field (e.g., zero, NaN, subnormal, infinity). This is similar to what is done in many BFP units.

As a combined binary/decimal multiplier design now exists and many combined binary/decimal adder designs have been developed, research to develop a combined BFP/DFP unit is appropriate at this point. A significant percentage of the area in a BFP unit is used to realize the parallel accumulation of partial products in

the multiply operation and to realize operand alignment and addition in the addition/subtraction operations. Thus, it may very well be possible for BFP instructions to have similar latencies and throughput in a combined BFP/DFP unit as they do in a dedicated unit and with significantly less area than if implemented separately. Research along these lines into extending the major dataflow components in a BFP design to support the various DFP operations would be beneficial.

There are also a number of worthwhile research pursuits specific to decimal multiplication. One idea is to apply the use of binary carry-save adders to the iterative DFXP multiplier. Another idea is to extend the combined BFXP/DFXP multiplier (using binary CSAs in the partial product accumulation tree) to support the floating-point requirements of IEEE 754-2008 for both radices. Also, as mentioned in the last section, current parallel multiplier designs require significant wiring resource to distribute the partial products to the partial product accumulation tree. To reduce the number of wiring tracks, research is needed to examine ways to generate the partial products locally, while still using a parallel reduction technique. Further, applying the research on parallel DFP multiplication to division and square root, perhaps utilizing Goldschmidt's concepts, may prove interesting.

We are nearing 50 years of modern computing with binary arithmetic, and there are still a large number of publications, patents, and presentations emerging from research in this field. I believe we have only recently begun to satisfy the current needs and arouse the interests of both commercial and personal users in decimal computer arithmetic. And this need/interest will only increase as the following cycle gains momentum: application developers write more programs calling for DFP instructions → researchers improve DFP hardware solutions → more platforms offer improved DFP support → application developers write more programs calling for DFP instructions. Only the market will decide, but as continued research narrows the performance gap

between BFP and DFP arithmetic, we may very well see DFP applications displacing BFP applications in certain computing markets.

6.3 Closing

My contribution to advancing the body of knowledge includes two iterative DFXP multiplication algorithms and hardware designs, the adaptation of one of these designs to support DFP, and the extension of another research team's parallel DFXP multiplier design to support DFP. Both designs which support DFP multiplication are in compliance with IEEE 754-2008. This dissertation serves as a useful resource for those seeking to implement area-efficient implementations of DFXP and DFP multiplication, to extend a given DFXP multiplication design or other decimal operations to support DFP, and to pursue additional research in the area of DFXP and DFP multiplication.

Bibliography

- [1] M. F. Cowlshaw, “Decimal Arithmetic FAQ.” World Wide Web.
<http://speleotrove.com/decimal/decifaq1.html>.
- [2] I. C. Society, “Computing History Timeline.” World Wide Web.
[http://www.computer.org/portal/cms_docs_computer/computer/timeline/-
timeline.pdf](http://www.computer.org/portal/cms_docs_computer/computer/timeline/-timeline.pdf).
- [3] M. F. Cowlshaw, “Densely Packed Decimal Encoding,” *IEE Proceedings – Computers and Digital Techniques*, vol. 149, pp. 102–104, May 2002.
- [4] A. Svoboda, “Decimal Adder with Signed Digit Arithmetic,” *IEEE Transaction on Computers*, vol. C, pp. 212–215, March 1969.
- [5] T. Lang and A. Nannarelli, “A Radix-10 Combinational Multiplier,” in *Asilomar Conference on Signals, Systems, and Computers*, pp. 313–317, Oct.–Nov. 2006.
- [6] A. Vazquez, E. Antelo, and P. Montuschi, “A New Family of High-Performance Parallel Decimal Multipliers,” in *18th IEEE Symposium on Computer Arithmetic*, pp. 195–204, IEEE Computer Society, June 2007.
- [7] B. J. Hickmann, M. A. Erle, and M. J. Schulte, “Improved Combined Binary/Decimal Fixed-Point Multipliers,” *IEEE*, October 2008.

- [8] B. J. Hickmann, A. Krioukov, M. A. Erle, and M. J. Schulte, “A Parallel IEEE P754 Decimal Floating-Point Multiplier,” in *25th International Conference on Computer Design*, pp. 296–303, IEEE, IEEE Computer Society, October 2007.
- [9] Free Software Foundation, “GNU C Compiler (GCC) 4.3 Release.” World Wide Web. <http://gcc.gnu.org/gcc-4.3>.
- [10] JTC 1/SC 22/WG 4, *ISO/IEC 1989: Information technology – Programming languages – COBOL*. New York: American National Standards Institute, first ed., December 2002. 859 pages.
- [11] P. Crismer, “Eiffel Decimal Arithmetic Library.” World Wide Web. <http://eiffelzone.com/esd/eda/index.html>, version 1.08.
- [12] Sun Microsystems, “BigDecimal Java Class.” World Wide Web. <http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html>.
- [13] D. Currie, “Lua decNumber Library.” World Wide Web. <http://luaforge.net/projects/ldecnumber/>, version 21.
- [14] A. Gough (Maintainer), “PERL BigInt Library.” World Wide Web. http://dev.perl.org/perl6/pdd/pdd14_bignum.html, version 1.5.
- [15] F. Batista, “Decimal Data Type.” World Wide Web. <http://www.python.org/dev/peps/pep-0327>, version 62268.
- [16] TC X3J18, *ANSI X3.274-1996: American National Standard for Information Technology - Programming Language REXX*. New York: American National Standards Institute, February 1996. 167 pages.
- [17] S. Kobayashi, “Ruby BigDecimal Class.” World Wide Web. <http://www.ruby-doc.org/stdlib/libdoc/bigdecimal/rdoc>, version 16-April-2007.

- [18] Floating-Point Working Group, *ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*. New York: The Institute of Electrical and Electronics Engineers, August 1985. 17 pages.
- [19] Floating-Point Working Group, *ANSI/IEEE Std 854-1987: IEEE Standard for Radix-Independent Floating-Point Arithmetic*. New York: The Institute of Electrical and Electronics Engineers, October 1987. 16 pages.
- [20] IEEE Working Group of the Microprocessor Standards Subcommittee, *IEEE Standard for Floating-Point Arithmetic*. New York: The Institute of Electrical and Electronics Engineers, 2008.
- [21] G. E. Moore, “Cramming More Components onto Integrated Circuits,” *Electronics*, vol. 38, pp. 114–117, April 1965.
- [22] M. Cornea, C. Anderson, and C. Tsen, “Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic,” in *First International Conference on Software and Data Technologies*, (Setúbal, Portugal), pp. 12–20, INSTICC Press, September 2006.
- [23] L. Eisen, J. W. W. III, H.-W. Tast, N. Mding, 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. 663–684, November 2007.
- [24] A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic, “Decimal Floating-Point in Z9: An Implementation and Testing Perspective,” *IBM Journal of Research and Development*, vol. 51, pp. 217–228, January 2007.

- [25] E. M. Schwarz, J. S. Kapernick, and M. F. Cowlshaw, “Decimal Floating-Point Support on the IBM System z10 Processor,” *IBM Journal of Research and Development*, vol. 53, no. 1/2, 2009.
- [26] M. A. Erle, M. J. Schulte, and J. M. Linebarger, “Potential Speedup Using Decimal Floating-Point Hardware,” in *Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1073–1077, November 2002.
- [27] M. J. Schulte, N. Lindberg, and A. Laxminarain, “Performance Evaluation of Decimal Floating-Point Arithmetic,” in *Proceedings of the 6th IBM Austin Center for Advanced Studies Conference*, (Austin, TX), February 2005.
- [28] L.-K. Wang, C. Tsen, M. J. Schulte, and D. Jhalani, “Benchmarks and Performance Analysis for Decimal Floating-Point Applications,” in *25th International Conference on Computer Design*, pp. 164–170, IEEE, October 2007.
- [29] K. Quinn, “Ever Had Problems Rounding Off Figures? This Stock Exchange Has,” *Wall Street Journal*, November 8 1983.
- [30] M. Blair, S. Obenski, and P. Bridickas, “Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia,” Tech. Rep. GAO/IMTEC-92-26, United States General Accounting Office, Washington, D.C. 20548, February 1992.
- [31] A. Tsang and M. Olschanowsky, “A Study of Database 2 Customer Queries,” IBM Technical Report 03.413, IBM, San Jose, CA, April 1991.
- [32] Intel, “Intel Decimal Floating-Point Math Library.” World Wide Web. <http://softwarecommunity.intel.com/articles/eng/3687.htm>, version 1.0.

- [33] M. F. Cowlshaw, “IBM decNumber Library.” World Wide Web. <http://www.alphaworks.ibm.com/tech/decnumber>, version 3.56.
- [34] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, “The IBM z900 Decimal Arithmetic Unit,” in *Asilomar Conference on Signals, Systems, and Computers*, vol. 2, pp. 1335–1339, November 2001.
- [35] L.-K. Wang and M. J. Schulte, “Decimal Floating-Point Division Using Newton-Raphson Iteration,” in *15th IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 84–95, IEEE Computer Society Press, September 2004.
- [36] M. Mittal, A. Peleg, and U. Weiser, “MMX Technology Architecture Overview,” *Intel Technology Journal*, vol. Q3, pp. 1–12, 1997.
- [37] M. Cornea and J. Crawford, “IEEE 754R Decimal Floating-Point Arithmetic: Reliable and Efficient Implementation on Intel© Architecture Platforms.” World Wide Web, Intel Technology Journal, February 2007. <http://www.intel.com/technology/itj/2007/v11i1/s2-decimal/1-sidebar.htm>.
- [38] M. Bhat, J. Crawford, R. Morin, and K. Shiv, “Performance Characterization of Decimal Arithmetic in Commercial Java Workloads,” in *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 54–61, April 2007.
- [39] M. A. Erle and M. J. Schulte, “Decimal Multiplication Via Carry-Save Addition,” in *14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 348–358, June 2003.

- [40] M. A. Erle, E. M. Schwarz, and M. J. Schulte, “Decimal Multiplication with Efficient Partial Product Generation,” in *17th IEEE Symposium on Computer Arithmetic*, pp. 21–28, IEEE Computer Society, June 2005.
- [41] M. A. Erle, M. J. Schulte, and B. J. Hickmann, “Decimal Floating-Point Multiplication Via Carry-Save Addition,” in *18th IEEE Symposium on Computer Arithmetic*, pp. 46–55, IEEE Computer Society, June 2007.
- [42] M. A. Erle, B. J. Hickmann, and M. J. Schulte, “Decimal Floating-Point Multiplication,” *accepted for publication in IEEE Transactions on Computers*, 2008.
- [43] L.-K. Wang and M. J. Schulte, “Decimal Floating-Point Square Root Using Newton-Raphson Iteration,” in *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 309–315, July 2005.
- [44] G. Ifrah, *A Universal History of Computing: From Prehistory to Computers*. New York, NY: John Wiley and Sons, Inc., 2000.
- [45] F. Bacon, *The Advancement of Learning, Book 6, Chapter 1*. 1605.
- [46] G. Boole, *An Investigation of the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities*. London: Walton and Maberley, 1854.
- [47] C. E. Shannon, “A Symbolic Analysis of Relay and Switching Circuits,” *Transactions American Institute of Electrical Engineers*, vol. 57, pp. 713–723, March 1938. Included in Part B.

- [48] H. H. Goldstine and A. Goldstine, “The Electronic Numerical Integrator and Computer (ENIAC),” *IEEE Annals of the History of Computing*, vol. 18, no. 1, pp. 10–16, 1996.
- [49] R. Head, “Univac: a Philadelphia Story,” *IEEE Annals of the History of Computing*, vol. 23, no. 3, pp. 60–63, 2001.
- [50] D. E. Knuth, “The IBM 650: An Appreciation from the Field,” *IEEE Annals of the History of Computing*, vol. 8, no. 1, pp. 50–55, 1986.
- [51] N. Stern, *From ENIAC to UNIVAC – An Appraisal of the Eckert-Mauchly Computers*. Bedford, MA: Digital Press, 1981.
- [52] M. V. Wilkes, “Arithmetic on the EDSAC,” *IEEE Annals of the History of Computing*, vol. 19, no. 1, pp. 13–15, 1997.
- [53] M. R. Williams, “The Origins, Uses, and Fate of the EDVAC,” *IEEE Annals of the History of Computing*, vol. 15, no. 1, pp. 22–38, 1993.
- [54] T. Leser and M. Romanelli, “Programming and Coding for ORDVAC.” World Wide Web, October 1956. http://www.bitsavers.org/pdf/ordvac/-ORDVAC_programming_Oct56.pdf.
- [55] G. Gray, “Unisys History Newsletter (Volume 1, Number 2).” World Wide Web, December 1992. <http://www.cc.gatech.edu/gvu/people/randy.carpenter/-folklore/v1n2.html>.
- [56] B. Randell, “From Analytical Engine to Electronic Digital Computer: The Contributions of Ludgate, Torres, and Bush,” *IEEE Annals of the History of Computing*, vol. 4, no. 4, pp. 327–341, 1982.

- [57] NationMaster Encyclopedia, “Z1 (Computer).” World Wide Web. [http://www.nationmaster.com/encyclopedia/Z1-\(computer\)](http://www.nationmaster.com/encyclopedia/Z1-(computer)).
- [58] J. V. Atanasoff, “Advent of Electronic Digital Computing,” *IEEE Annals of the History of Computing*, vol. 6, no. 3, pp. 229–282, 1984.
- [59] J. Copeland, “Colossus: Its Origins and Originators,” *IEEE Annals of the History of Computing*, vol. 26, no. 4, pp. 38–45, 2004.
- [60] R. Campbell, *Makin’ Numbers: Howard Aiken and the Computer*, ch. “Aiken’s First Machine: the IBM ASCC/Harvard Mark I”, pp. 31–63. Cambridge, MA: MIT Press, 1999.
- [61] ScienCentral, Inc. and The American Institute of Physics, “Miracle Month - The Invention of the First Transistor.” World Wide Web, 1999. <http://www.pbs.org/transistor/background1/events/miraclemo.html>.
- [62] W. Shockley, M. Sparks, and G. K. Teal, “p-n Junction Transistors,” *Physical Review*, vol. 83, pp. 151–162, July 1951.
- [63] ScienCentral, Inc. and The American Institute of Physics, “The First Silicon Transistor.” World Wide Web, 1999. <http://www.pbs.org/transistor/science/events/silicont1.html>.
- [64] Texas Instruments, “The Chip that Jack Built.” World Wide Web, 1995. <http://www.ti.com/corp/docs/kilbyctr/jackbuilt.shtml>.
- [65] I. P. S. of Japan, “Historic Computers in Japan: [NEC] NEAC 2201.” World Wide Web. <http://museum.ipsj.or.jp/en/computer/dawn/0018.html>.

- [66] C. Cole, “The Remington Rand Univac LARC.” World Wide Web. <http://www.computer-history.info/Page4.dir/pages/-LARC.dir/LARC.Cole.html>.
- [67] W. B. *et al.*, *Planning a Computer System: Project Stretch*. New York: McGraw-Hill Book Company, 1962. <http://ed-thelen.org/comp-hist/IBM-7030-Planning-McJones.pdf>.
- [68] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, “Architecture of the IBM System/360,” *IBM Journal of Research and Development*, vol. 8, pp. 87–101, April 1964.
- [69] J. E. Thornton, “The CDC 6600 Project,” *IEEE Annals of the History of Computing*, vol. 2, no. 4, pp. 338–348, 1980.
- [70] ISO/IEC JTC 1/SC 22/WG 9, *ISO/IEC 8652:1995: Ada Reference Manual: Language and Standard Libraries*. Switzerland: International Organization for Standardization, June 2001. 560 pages.
- [71] JTC 1 (ECMA TC 39/TG 2), *ISO/IEC 23270: Information Technology – C# Language Specification*. Switzerland: International Organization for Standardization, first ed., April 2003. 471 pages.
- [72] Microsoft, “Visual Basic.” World Wide Web. <http://msdn.microsoft.com/en-us/vbasic/default.aspx>, version 9.0.
- [73] TC X3J1, *ANSI X3.53-1976: American National Standard Programming Language PL/I*. New York: American National Standards Institute, February 1976. 403 pages.

- [74] ISO/IEC JTC 1/SC 32, *ISO/IEC 9075:1992: Information Technology – Database Languages – SQL*. Switzerland: International Organization for Standardization (ISO), March 1992. 626 pages.
- [75] A. M. P. V. Biron, “XML Schema Part 2: Datatypes Second Edition.” World Wide Web, October 2004. <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028>.
- [76] M. F. Cowlishaw, “General Decimal Arithmetic Specification.” <http://speleotrove.com/decimal/decarith.html>, July 2008. Draft 1.68.
- [77] Intel, “Reference Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic.” World Wide Web. http://cache-www.intel.com/cd/00/00/29/43/294339_294339.pdf.
- [78] JTC 1/SC 22/WG 14, “TR 24732: Extensions for the Programming Language C to Support Decimal Floating Point Arithmetic.” World Wide Web, 2007. <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1312.pdf>.
- [79] JTC 1/SC 22/WG 14, “TR 24733: C++ Decimal Floating Point Arithmetic Extensions.” World Wide Web, 2008. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2732.pdf>.
- [80] Free Software Foundation, “GNU C Compiler (GCC) 4.2 Release.” World Wide Web. <http://gcc.gnu.org/gcc-4.2>.
- [81] IBM, “IBM C and C++ Compilers.” World Wide Web. <http://www.ibm.com/software/awdtools/xlcpp>.
- [82] IBM, “IBM Enterprise PL/I V3.7.” World Wide Web. http://www.ibm.com/common/ssi/rep_ca/6/897/ENUS207-266/index.html.

- [83] IBM, “IBM DB2 V9.5.” World Wide Web. http://www.ibm.com/common/ssi/rep_ca/1/897/ENUS207-261/index.html.
- [84] IBM, “IBM High Level Assembler Release 6.” World Wide Web. <http://www.ibm.com/software/awdtools/hlasm/library.html>.
- [85] P. Shaw, “DFPAL.” World Wide Web. <http://speleotrove.com/decimal/#dfpal>.
- [86] M. Koechl and P. Hartman and O. Rutz and P. Shah, “Decimal Floating Point Computations in SAP NetWeaver 7.10.” World Wide Web, September 2007. <http://www.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP101104>.
- [87] M. S. Cohen, T. E. Hull, and V. C. Hamacher, “CADAC: A Controlled-Precision Decimal Arithmetic Unit,” *IEEE Transactions on Computers*, vol. C-32, pp. 370–377, April 1983.
- [88] T. E. Hull, M. S. Cohen, and C. B. Hall, “Specifications for a Variable-Precision Arithmetic Coprocessor,” in *10th Symposium on Computer Arithmetic*, pp. 127–131, IEEE, IEEE Computer Society, May 1991.
- [89] G. Bohlender and T. Teufel, “BAP-SC: A Decimal Floating-Point Processor for Optimal Arithmetic,” in *Computer Arithmetic: Scientific Computation and Programming Languages*, pp. 31–58, Stuttgart, Germany: B. G. Teubner, 1987.
- [90] *IA-32 Intel Architecture Software Developer’s Manual*, vol. 2: Instruction Set Reference, ch. 3: Instruction Set Reference. Intel, 2001.
- [91] Motorola, “Motorola M68000 Family Programmers Reference Manual.” World Wide Web, 1992. http://www.freescale.com/files/archives/doc/ref_manual/-M68000PRM.pdf.

- [92] G. Kane, *PA-RISC 2.0 Architecture*, ch. 7: Instruction Descriptions. Prentice Hall, 1996.
- [93] *ESA/390 Principles of Operation*, ch. 8: Decimal Arithmetic Instructions. IBM, 2001.
- [94] E. O. Carbares, “IBM System z10 Enterprise Class Mainframe Server Features and Benefits.” World Wide Web, February 2008. <http://www.ibm.com/systems/z/hardware/z10ec/features.html>.
- [95] SilMinds, “Decimal Floating Point Arithmetic IP Cores Family.” World Wide Web, 2008. <http://www.silminds.com/resources/SilMinds-DFPA-IP-Cores-Family.pdf>.
- [96] R. Eissa, A. Mohamed, R. Samy, T. Eldeeb, Y. Farouk, M. Elkhoully, and H. Fahmy, “A Decimal Fully Parallel and Pipelined Floating Point Multiplier,” in *Asilomar Conference on Signals, Systems, and Computers*, 2008.
- [97] K. A. Duke, “Decimal Floating-Point Processor,” *IBM Technical Disclosure Bulletin*, vol. 12, p. 862, November 1969.
- [98] F. N. Ris, “A Unified Decimal Floating-Point Architecture for the Support of High-Level Languages,” *ACM SIGNUM Newsletter*, vol. 11, pp. 18–23, October 1976.
- [99] T. C. Chen and I. T. Ho, “Storage-Efficient Representation of Decimal Data,” *Communications of the ACM*, vol. 18, pp. 49–52, January 1975.
- [100] P. Johnstone and F. E. Petry, “Higher Radix Floating Point Representations,” in *9th IEEE Symposium on Computer Arithmetic*, pp. 128–135, IEEE, IEEE Computer Society, September 1989.

- [101] IEEE P1596.5 Working Group, *ANSI/IEEE Std 1596.5-1993: IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*. New York: The Institute of Electrical and Electronics Engineers, April 1994.
- [102] D. Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic,” *ACM Computing Surveys*, vol. 23, pp. 5–48, March 1991.
- [103] D. W. Matula, “The IEEE Standard for Floating-Point Systems.” personal communication. draft section of “Finite Precision Number Systems and Arithmetic” by P. Kornerup and D. W. Matula, available upon request.
- [104] M. F. Cowlshaw, E. M. Schwarz, R. M. Smith, and C. F. Webb, “A Decimal Floating-Point Specification,” in *15th IEEE Symposium on Computer Arithmetic*, pp. 147–154, IEEE Computer Society, July 2001.
- [105] R. K. Richards, *Arithmetic Operations in Digital Computers*. New Jersey: D. Van Nostrand Company, Inc., 1955.
- [106] B. Parhami, “Carry-Free Addition of Recoded Binary Signed-Digit Numbers,” *IEEE Transaction on Computers*, vol. 37, pp. 1470–1476, November 1988.
- [107] B. Shirazi, D. Y. Y. Yun, and C. N. Zhang, “RBCD: Redundant Binary Coded Decimal Adder,” *IEE Proceedings*, vol. 136, part E, pp. 156–160, March 1989.
- [108] J. L. Anderson, “Binary or BCD Adder with Precorrected Result,” *U.S. Patent*, October 1979. #4,172,288.
- [109] L. P. Flora, “Fast BCD/Binary Adder,” *U.S. Patent*, April 1991. #5,007,010.

- [110] H. Fischer and W. Rohsaint, “Circuit Arrangement for Adding or Subtracting Operands in BCD-Code or Binary-Code,” *U.S. Patent*, September 1992. #5,146,423.
- [111] U. Grupe, “Decimal Adder,” *U.S. Patent*, January 1976. #3,935,438.
- [112] J. Thompson, N. Karra, and M. J. Schulte, “A 64-bit Decimal Floating-Point Adder,” in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pp. 297–298, February 2004.
- [113] R. D. Kenney, M. J. Schulte, and M. A. Erle, “A High-Frequency Decimal Multiplier,” in *International Conference on Computer Design: VLSI in Computers and Processors*, pp. 26–29, IEEE, October 2004.
- [114] A. Vazquez and E. Antelo, “Conditional Speculative Decimal Addition,” in *7th Conference on Real Numbers and Computers*, pp. 47–57, July 2006.
- [115] M. S. Schmookler and A. W. Weinberger, “High Speed Decimal Addition,” *IEEE Transactions on Computers*, vol. C, pp. 862–867, August 1971.
- [116] I. Koren, *Computer Arithmetic Algorithms*. New Jersey: Prentice-Hall, Inc., 1993.
- [117] A. Cauchy, “Calculs Numériques – Sur les moyens d’éviter les erreurs dans les calculs numériques,” in *Oeuvres Complètes D’Augustin Cauchy*, vol. 5 of 1, pp. 431–442, Gauthier-Villars, 1840.
- [118] R. D. Kenney and M. J. Schulte, “High-Speed Multioperand Decimal Adders,” *IEEE Transactions on Computers*, vol. 54, pp. 953–963, August 2005.

- [119] A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 389–400, September 1961.
- [120] G. Metzger and J. E. Robertson, "Elimination of Carry Propagation in Digital Computers," *Proceedings of the International Conference on Information Processing*, pp. 389–396, June 1959.
- [121] E. M. Schwarz, *High-Performance Energy-Efficient Microprocessor Design*, ch. 8. Binary Floating-Point Unit Design: The Fused Multiply-Add Dataflow, pp. 189–208. Dordrecht, The Netherlands: Springer, 2006.
- [122] S. R. Carlough and E. M. Schwarz, "Decimal Multiplication Using Digit Recoding," *U.S. Patent*, November 2006. #7,136,893.
- [123] R. H. Larson, "Medium Speed Multiply," *IBM Technical Disclosure Bulletin*, p. 2055, December 1973.
- [124] R. H. Larson, "High Speed Multiply Using Four Input Carry Save Adder," *IBM Technical Disclosure Bulletin*, pp. 2053–2054, December 1973.
- [125] T. Ueda, "Decimal Multiplying Assembly and Multiply Module," *U.S. Patent*, January 1995. #5,379,245.
- [126] R. T. Jackson, C. Kurtz, and R. Moore, "Decimal Multiplication and Decimal-to-binary Conversion Utilizing a Three-input Adder," *IBM Technical Disclosure Bulletin*, vol. 14, p. 543, July 1971.
- [127] S. H. Angelov and S. V. Hristova, "Coded Decimal Multiplication By Successive Additions," *U.S. Patent*, May 1972. #3,644,724.

- [128] R. L. Hoffman and T. L. Schardt, “Packed Decimal Multiply Algorithm,” *IBM Technical Disclosure Bulletin*, vol. 18, pp. 1562–1563, October 1975.
- [129] S. Singh and A. Weinberger, “High-speed Binary and Decimal Multiply by Array Assist,” *IBM Technical Disclosure Bulletin*, vol. 18, pp. 4105–4106, May 1976.
- [130] T. Ohtsuki, Y. Oshima, S. Ishikawa, K. Yabe, and M. Fukuta, “Apparatus for Decimal Multiplication,” *U.S. Patent*, June 1987. #4,677,583.
- [131] A. Yamaoka, K. Wada, and K. Kuriyama, “Decimal Multiplier Device and Method Therefor [sic],” *U.S. Patent*, May 1988. #4,745,569.
- [132] J. J. Bradley, B. L. Stoffers, T. R. S. Jr., and M. A. Widen, “Simplified Decimal Multiplication by Stripping Leading Zeros,” *U.S. Patent*, June 1986. #4,615,016.
- [133] C. S. Wallace, “A Suggestion for a Fast Multiplier,” *IEEE Transactions on Computers*, vol. 13, no. 2, 1964.
- [134] L. Dadda, “Some Schemes for Parallel Multipliers,” *Alta Frequenza*, vol. 34, pp. 349–356, 1965.
- [135] J. R. Pivnichny, “High Speed Decimal Multipliers,” *IBM Technical Disclosure Bulletin*, vol. 24, pp. 2612–2617, October 1981.
- [136] B. Parhami, *Computer Arithmetic Algorithms and Hardware Design*. New York: Oxford University Press, Inc., 2000.
- [137] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, “SIS: A System for Sequential Circuit Synthesis,” tech. rep., 1992.

- [138] N. T. Quach, N. Takagi, and M. J. Flynn, “Systematic IEEE Rounding Method for High-Speed Floating-Point Multipliers,” *IEEE Transactions on VLSI Systems*, vol. 12, pp. 511–521, May 2004.
- [139] M. S. Schmookler and K. J. Nowka, “Leading Zero Anticipation and Detection – A Comparison of Methods,” in *15th IEEE Symposium on Computer Arithmetic*, pp. 7–12, IEEE Computer Society, July 2001.
- [140] IBM Floating-Point Test Generator, “Floating-Point Test Suite for IEEE 754R Standard.” World Wide Web. <http://-www.haifa.il.ibm.com/projects/-verification/fpgen/ieeets.html>.
- [141] IBM, “General Decimal Arithmetic Testcases.” World Wide Web. <http://speleotrove.com/decimal/-dectest.html>.
- [142] A. D. Booth, “A Signed Binary Multiplication Technique,” *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.
- [143] R. M. M. Aharoni, R. Maharik, and A. Ziv, “Solving Constraints on the Intermediate Result of Decimal Floating-Point Operations,” in *18th IEEE Symposium on Computer Arithmetic*, pp. 38–45, IEEE Computer Society, June 2007.

Appendix A

Glossary

A term often has multiple meanings across different disciplines and occasionally has multiple meanings within a discipline. The following list of terms and definitions¹ is provided to clarify the meaning of most of the technical terms used in this document.

The reasoning behind the usage of radix, base, power, and exponent warrants discussion. The term base can be used to describe the operand that is raised to a particular power and the number system (e.g., the decimal number system is *base ten*). To prevent confusion, radix is used when referring to the number system. Further, although the term exponent is commonly the quantity to which the base is raised, power is used instead to prevent confusion with the exponent portion of the operand. The raising of a base by a particular power is still referred to as exponentiation, however, as this is the common name and should not cause confusion.

actual exponent - The exponent component of an IEEE DFP entity without any bias; the integer portion of a real number describing the order of the significand's LSD.

¹Wikipedia.org was used as a reference.

addend - The operand to be added to the augend.

addition - The binary operation whereby a sum is developed whose quantity, prior to rounding, is equivalent to the augend incremented by 1 addend times.

additive inverse - The negative of A when A is in signed-magnitude form, represented by $-A$, such that $A + -A = 0$; the unary operation whereby a difference is developed whose quantity is equivalent to the subtraction of 0 and the subtrahend; the complement of A when A is in radix-complement form, represented by \bar{A} , such that $A + \bar{A} = S(a_n) - 1$, where A has n digits and $S()$ is the significance function.

AND - The binary operation whereby a result is developed by the bit-wise conjunction of two operands; e.g., if $A = 1100$ and $B = 1010$, then $A \cdot B = 1000$.

augend - The operand to which the addend is to be added.

base - The operand that is to be raised by a power.

biased exponent - See **exponent**.

binary coded decimal - A number system wherein a decimal digit is represented by four binary digits of weight 2^3 , 2^2 , 2^1 , and 2^0 , respectively, from left to right.

cohort - A set of floating-point entities all of which describe the same quantity.

combination field - The fixed-length portion of an IEEE DFP entity in storage that indicates ∞ , NaN, or a normal number, and, if a normal number, contains the leading two bits of the exponent and the leading digit of the significand.

complement - The additive inverse of an integer number stored in radix complement form or diminished-radix complement form.

conformer - The operand that is to be quantized according to the template.

correspondent - The result of quantization; $C = A \times B$, where C is the correspondent, A is the conformer, and B is the template whose quantum is to be matched (i.e., $S(b_0)$).

decimal floating-point - The set of representations of approximations of radix ten real numbers; the arithmetic involving radix ten real numbers.

delet - A ten bit vector containing three Densely Packed Decimal digits.

densely packed decimal - An encoding scheme in which three BCD digits are compressed into a ten-bit vector.

difference - The result of subtraction; $D = A - B$, where D is the difference, A is the minuend, and B is the subtrahend.

digit-overlap form - A representation of a quantity as (A, B) , wherein $A+B$ equals the quantity, the first vector is comprised of digits, and the second vector, whose order is one higher than the first vector, is comprised of full or partial digits; e.g., $99 \times 9 = 891$ can be represented as $(88, 11)$, where 88 is an order higher than 11.

diminished-radix complement form - A representation of integer numbers wherein positive quantities are simply the integer numbers and negative quantities are the integer numbers added to one less than the radix raised to one more than the order of the MSD; i.e., $(S(a_n) - 1) - A = \bar{A}$, where A has n digits and $S()$ is the significance function.

dividend - The operand that is to be divided by the divisor.

division - The binary operation whereby a quotient is developed whose quantity, prior to rounding, is equivalent to the number of times the divisor can be subtracted from dividend.

divisor - The operand by which the dividend is to be divided.

entity - A generic description for the contents of a vector, register, or memory address, which may or may not represent a number.

exact floating-point number - A representable number in the intended format is the infinitely precise result of the calculation.

exception - An error condition that alters the intended flow of control.

XS3 - A number system wherein a decimal digit is represented by four binary digits of weight 2^3 , 2^2 , 2^1 , and 2^0 , respectively, from left to right and biased by a positive three.

exponent - The number describing the order of the LSD of the integer portion of the significand; the fixed-length, component of an IEEE DFP entity, which, together with two bits from the combination component forms a binary, natural number describing the order of the LSD of the integer portion of the significand and biased to make the range of orders nonnegative.

exponent bias - A constant added to the order of a floating-point number to make the adjusted range of orders nonnegative; a constant added to the order such that the order of the smallest significand is nonnegative for a given IEEE DFP format.

exponentiation - The binary operation whereby a yield is developed whose quantity, prior to rounding, is equivalent to the base multiplied by itself power times.

exponent range - The number of different orders the significand's LSD can have for a given IEEE DFP format; the difference between the maximum exponent and the minimum exponent.

finite floating-point number - A floating-point entity that is either zero or a sub-normal or normal number.

fixed-point number - A representation of a subset of the real numbers and rounded irrational numbers as a fixed-length vector with a fixed amount of digits to the right of the radix point.

floating-point number - A representation of a subset of the real numbers, rounded real numbers, and rounded irrational numbers in a fixed-length format comprised of a sign bit, a natural number significand, and a natural number exponent in the form $-1^{sign} \times significand \times radix^{exponent}$.

following exponent - The fixed-length portion of an IEEE DFP entity in storage, which, together with two bits from the combination field, comprises the exponent component.

format - The arrangement of data within a fixed-length vector.

full tuple set - All the single-digit multiples of the multiplicand (1-tuple through $(radix - 1)$ -tuple).

inexact floating-point number - A representable number in the intended format that is closest to the infinitely precise result of a calculation, based on the rounding mode of the calculation.

infinite floating-point number - A floating-point entity that represents all numbers larger in magnitude than the largest representable number.

intended format - The set of fixed-length, IEEE DFP entities onto which a result must be mapped.

least significant bit - The lowest order bit in a digit or a vector; the furthest-right bit in a digit or a vector.

least significant digit - The lowest order digit in a vector; the furthest-right digit in a vector.

logical negation - The unary operation whereby the truth value is inverted; equivalent to performing the arithmetic operation of diminished-radix complement in radix two (i.e., ones' complement).

maximum exponent - The largest, unbiased order of a significand's LSD for a given IEEE DFP format.

minimum exponent - The smallest, unbiased order of a significand's LSD for a given IEEE DFP format.

minterm - A product term in a Boolean equation comprised of one or more variables, each appearing at most once in either its true or complement form.

minuend - The operand from which the subtrahend is to be subtracted.

most significant bit - The highest order bit in a digit or a vector; the furthest-left bit in a digit or a vector.

most significant digit - The highest order digit in a vector; the furthest-left digit in a vector.

multiplication - The binary operation whereby a product is developed whose quantity, prior to rounding, is equivalent to the multiplicand added to itself multiplier times.

multiplicative inverse - The reciprocal of A , represented by A^{-1} , such that $A \times A^{-1} = 1$; zero does not have a reciprocal.

multiplicand - The operand that is to be multiplied by the multiplier; operand A in a fixed-point multiply and C^A in a floating-point multiply; the value which is added to itself multiple times as specified by the value of the multiplier value.

multiplier - The operand by which the multiplicand is to be multiplied; operand B in a fixed-point multiply and C^B in a floating-point multiply; the value which dictates the number of times the multiplicand is added to itself.

negative - The additive inverse of a quantity in signed-magnitude form.

negative floating-point number - A floating-point representation whose sign bit is set and is either zero or a subnormal, normal, or finite number.

normal floating-point number - A floating-point representation that is a non-zero number whose exponent is within the range of its format.

NOT - The unary operation whereby a result is developed by bit-wise logical negation of the operand; e.g., if $A = 10$, then $\bar{A} = 01$.

not-a-number - A cohort designed to provide test and diagnostic capability in a floating-point system at the expense of a small reduction in the number of distinct representable numbers.

null addition - The unary operation whereby a sum is developed whose quantity is equivalent to the addition of 0 and the addend.

operand - An input vector to an algorithm, logic equation, or circuit.

predicate - An operation that affirms or denies a proposition about its operand.

OR - The binary operation whereby a result is developed by the bit-wise disjunction of two operands; e.g., if $A = 1100$ and $B = 1010$, then $A + B = 1110$.

order - An integer number describing a digit's positional significance as a power of the radix; the distance a digit is from the radix point (left of the radix point is positive, right of the radix point is negative); e.g., the 5 in 500_D and the 1 in 100_B are both order 2; a vector A is of a higher order than a vector B if the $S(a_0) > S(b_0)$.

overloaded decimal representation - In this representation, each four-bit grouping of binary bits corresponds to a decimal digit and has the same weights as BCD-8421 code, however, its value can exceed ten.

partial product - The intermediate cumulative sum of repeated additions obtained during execution of a multiplication algorithm.

partial yield - The intermediate successive product of repeated multiplications obtained during execution of an exponentiation algorithm.

power - The operand by which the base is to be raised; the number of times the base is repetitively multiplied.

precision - The number of digits in the significand component for a given IEEE DFP format.

predicate - A proposition that is affirmed or denied.

product - The result of multiplication; $P = A \times B$, where P is the product, A is the multiplicand, and B is the multiplier.

quantum - The positional significance of a vector's LSD; e.g., 501.907 has a quantum of 10^{-3} ; in the IEEE DFP number system, the quantum is equal to ten raised to the exponent minus the bias of the exponent.

quantization - The binary operation whereby a correspondent is developed whose quantity is the conformer's significand properly adjusted and rounded if necessary such that its quantum matches the quantum of the template.

quiet not-a-number - A subset of the cohort of NaNs that, as an operand, propagates through an operation but does not signal an exception.

quotient - The result of division; $Q = A / B$, where Q is the quotient, A is the dividend, and B is the divisor.

radix - The base of a number system.

radix complement form - A representation of integer numbers wherein positive quantities are simply the integer number and negative quantities are the integer numbers added to the radix raised to the next higher order of the MSD; i.e., $S(a_n) - A = \bar{A} + 1$, where A has n digits.

reciprocal - The multiplicative inverse of an operand.

minimum remainder - The binary operation whereby a smallest magnitude quantity is developed to which can be added an integer multiple of the divisor to produce the dividend; e.g., $8[/]3 = 1$.

partial tuple set - A subset of all the single-digit multiples of the multiplicand (1-tuple through $(radix - 1)$ -tuple).

representation - The set of IEEE DFP floating-point entities that all describe the same class of entities.

result - An output vector from an algorithm, logic equation, or circuit.

secondary multiple set - A partial tuple set from which all the single-digit multiples of the multiplicand (1-tuple through $(radix - 1)$ -tuple) can be obtained through the sum or difference of at most two of its elements.

self-complementing - When the additive inverse can be achieved through logical negation.

sign bit - See **sign indicator field**.

sign indicator field - The fixed-length portion of an IEEE DFP entity in storage that indicates if a DFP number is negative or nonnegative; a single-bit component of a DFP number wherein 1 indicates a negative quantity and 0 indicates a nonnegative quantity.

signaling not-a-number - A subset of the cohort of NaNs that, as an operand, is converted to a quiet NaN and signals an exception, unless exceptions are disabled or the operation is one that does not signal exceptions.

signed-magnitude form - A representation of integer numbers wherein only natural numbers are used along with a bit or digit to indicate if the quantity is positive or negative.

significance - The weight of a digit.

significand - The portion of an IEEE DFP quantity which describes the coefficient of the number.

subnormal floating-point number - A floating-point representation that is a non-zero, finite number whose order is less than the minimum exponent of its format.

subtraction - The binary operation whereby a difference is developed whose quantity, prior to rounding, is equivalent to the subtrahend decremented by 1 minuend times.

subtrahend - The operand that is to be subtracted from the minuend.

sum - The result of addition; $S = A + B$, where S is the sum, A is the augend, and B is the addend.

sum of products - A boolean equation in which the output is expressed as the OR (sum) of AND terms (products).

template - The operand whose quantum is to be matched when the conformer is quantized.

tertiary multiple set - A partial tuple set from which all the single-digit multiples of the multiplicand (1-tuple through $(radix - 1)$ -tuple) can be obtained through the sum or difference of at most three of its elements.

trailing significand - The fixed-length portion of an IEEE DFP entity in storage, which, together with one digit from the combination field, comprises the significand component.

tuple - A multiplicand multiple; when a specific multiple needs to be addressed, the quantity of the multiple is a hyphenated prefix of the word “tuple” or a prefix of the variable name, e.g., A 's septuple is referred to as the 7-tuple of A or $7A$.

unbiased exponent - See **actual exponent**.

unordered - A relation indicating the comparison was illogical or not in order, i.e., at least one of the operands was a NaN.

vector - A one-dimensional array of one or more digits in which multiple digits are arranged in decreasing order from left to right.

weight - A positive, rational number describing a digit's positional significance equal to $radix^{order}$; i.e., the significance of the digit.

weighted compressor - An electronic circuit that accepts two or more digits of the same order, said digits comprised of two or more bits of different weights, and produces a minimal length output in digit-overlap form; e.g., a two-digit BCD-weighted compressor accepts two, four-bit BCD inputs and produces a four-bit BCD output of the same order as the inputs and a one-bit BCD output of the next higher order.

XOR - The binary operation whereby a result is developed by the bit-wise exclusive disjunction of two operands; e.g., if $A = 1100$ and $B = 1010$, then $A \oplus B = 0110$.

yield - The result of exponentiation; $Y = A ^ B$, where Y is the yield, A is the base, and B is the power.

Appendix B

Notation

This appendix describes the mathematical and logical notation used throughout this dissertation when discussing the data, the algorithms, and the logic equations.

DFP operands and results can be represented using the following equation, comprised of three components:

$$FP = -1^s \times C \times 10^E \quad (\text{B.1})$$

where s is the single-bit sign indicator (0 or 1), C is the fixed-length, natural number significand, and E is the fixed-length, natural number exponent, which is a biased form of the actual exponent. For reasons explained in Section 2.4, four fields are used to store the three components. Figure B.1 graphically lists the four fields used to store all DFP entities. (The figure is not to scale.)

Table B.1 lists the components and fields of a DFP entity and their symbols and names. Specific information, such as the data formats, the encoding of the significand, and the range of the exponent is described in Section 2.4.2.

Letters are used as variable names for FP numbers and their components (e.g.,

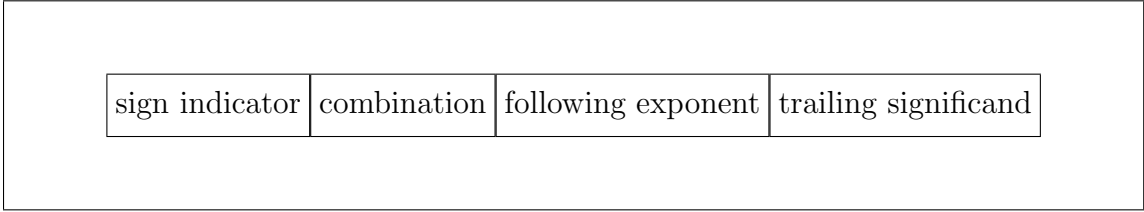


Figure B.1: DFP Storage Fields

Table B.1: Notation and Nomenclature of DFP Entity Components and Fields

Symbol	Name
i	sign indicator field/component
G	combination field
F	following exponent field
E	biased exponent - two MSD bits from G concatenated with F
T	trailing significand field
C	significand - one MSD digit from G concatenated with T

significands, exponents). When dealing with FP numbers, A , B , and C are used for operands, and the remaining letters are available for results. The letter chosen for a result is a matter of convenience and is often related to the operation. For example, P for the product of a multiplication. The use of A , B , and C is primarily to emphasize the importance the ordering of the operands has in the operation's algorithm. For unary operations such as the negative, the operand is labeled A . For binary operations such as addition and quantize, the operands are labeled A and B . And for ternary operations such as fused multiply-add, the operands are labeled A , B , and C .

Further, A is used for the operand on which a particular operation is performed or the operand upon which another operand acts. For example, A is used for the augend, minuend, multiplicand, dividend, conformer, and base. B is used for the active second operand. Example uses of B include the addend, subtrahend, multiplier, divisor,

template, and power. C is used for the third operand, which may be a passive or active operand.

To distinguish different forms of data, an upper case variable denotes FP numbers or words comprised of digits, a lower case variable denotes a decimal digit or bit, and a lower case variable with a bracketed number denotes a bit of a digit. To indicate a specific digit, a subscript may follow a lower case variable. A subscript following an upper case variable is used to indicate number or words which are part of an iterative equation. Finally, superscripts denote some unique aspect about the variable, such as incremented by one. As examples, $p[0]$ corresponds to bit 0 of the product P , $p_i[j]$ corresponds to the j^{th} bit of the i^{th} digit of the product P , and $p_i^{+1}[j]$ corresponds to the j^{th} bit of the i^{th} digit of the incremented product P . Examples of data notation are found in Table B.2.

Symbol	Meaning
P	number or multiple-digit word
P^{+0}, P^{+1}	non-incremented word P and incremented word P
P_i	P after i^{th} iteration
$p_i[j]$	j^{th} bit of the i^{th} digit of P
$p_i[0 : 3]$	bits 0 to 3 of the i^{th} digit of P

Table B.2: Notation of Operands and Data

Table B.3 contains a listing of the unary arithmetic operation symbols, how they are read, and the operation they represent. The complement operation is the diminished-radix complement, which in the general case, is distinctly different than logical negation. The radix complement is expressed as $(\bar{A} + 1)$.

Table B.4 contains a listing of the binary arithmetic operation symbols, how they are read, the operation they represent, and the name of their operands and results.

Table B.3: Unary Arithmetic Operations and Symbols

Symbol	Read	Operation	Operand	Result
$+a[i]$	positive	null addition	addend	sum
$-a[i]$	negative	additive inversion (signed-magnitude)	subtrahend	difference
$\overline{a[i]}$	complement of	additive inversion (diminished-radix)	true value	complement
common exponentiation operations				
$a[i]^{\frac{1}{2}}$	square root of	square root extraction	radicand	radical
$a[i]^{-1}$	reciprocal of	multiplicative inversion	recipricand	reciprocal

Table B.4: Binary Arithmetic Operations, Symbols, and Operand Names

Symbol	Read	Operation	Operands	Result
+	plus	addition	augend, addend	sum
-	minus	subtraction	subtrahend, minuend	difference
\times or \cdot	times	multiplication	multiplicand, multiplier	product
/	divided by	division	dividend, divisor	quotient
^	raised to	exponentiation	base, power	yield
\propto	proportional to	quantization	conformer, template	correspondent

The result name for the exponentiation operation and the symbol, operand names, and result name for quantization were developed for this dissertation. To reduce the length of an equation, multiplication can be implied by placing two variables next to each other without the \times symbol in between (e.g., $A \times B = AB$). Additionally, exponentiation can be implied by placing two variables next to each other without the $^$ symbol in between and superscripting the power operand (e.g., $A^B = A^B$).

Ternary arithmetic operations are expressed by placing a brace over the operands

and symbols. For example, the fused-multiply add operation is represented by $\overbrace{A \times B + C}$.

Table B.5: Logic Operations and Symbols

Symbol	Read	Operation
$\overline{a[i]}$	NOT	negation (unary)
$a[i] \vee b[j]$	OR	inclusive-disjunction or disjunction
$\overline{a[i] \vee b[j]}$	NOR	inverse disjunction
$a[i] \wedge b[j]$	AND	conjunction
$\overline{a[i] \wedge b[j]}$	NAND	inverse conjunction
$a[i] \oplus b[j]$	XOR	exclusive-disjunction
$\overline{a[i] \oplus b[j]}$	XNOR	correlation

Table B.6: Truth Tables of Logic Operations

$a[i]$	$b[j]$	NOT $a[i]$	AND	NAND	OR	NOR	XOR	XNOR
0	0	1	0	1	0	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	1	0
1	1	0	1	0	1	0	0	1

Table B.5 contains a listing of the logic symbols, how they are read, and the operation they represent. To reduce the length of an equation, AND can be implied by placing two variables next to each other without the \wedge symbol in between (e.g., $a[0] \wedge b[0] = a[0]b[0]$). Truth tables for each of the logic operations are included in Table B.6.

The precedence and associativity of the operators are listed in Table B.7 in order of decreasing precedence from top to bottom. Parentheses are used to alter the

Table B.7: Operator Precedence

Symbol	Operation	Associativity
\wedge	exponentiation	right to left
\times or \cdot , $/$	multiplication, division	left to right
$+a[i]$, $-a[i]$	null addition, additive inversion	right to left
$+$, $-$	addition, subtraction	left to right
\propto	quantization	right to left
\vee (logical), $\overline{a[i] \vee b[j]}$	disjunction	left to right
\wedge , $\overline{a[i] \wedge b[j]}$	conjunction	left to right
\oplus , $\overline{a[i] \oplus b[j]}$	exclusive-disjunction, correlation	left to right
\overline{A} (logical)	negation	left to right

precedence or associativity, or to improve readability.

Appendix C

Vita

Upon graduating from Parkland High School in 1984, I enlisted in the United States Army. During my two years in the military, I completed a number of skills assessment and interest inventory surveys to assist me in choosing my field of study in college. The results of these assessments indicated an aptitude/preference for engineering. I applied for admission to the College of Engineering at The Pennsylvania State University, and started in January of 1987.

I attended Penn State year-round and completed my degree in three and one half years. In the fall semester of my final year, I was offered employment with IBM. I was also accepted into the doctoral program in physics at Lehigh University but decided to obtain work experience before furthering my education. Therefore, in September, 1990, I joined a team of engineers responsible for verifying the architecture of a personal computer microprocessor at the IBM Williston Laboratory in Vermont.

Although I was obtaining significant knowledge of microprocessor architecture and verification, after a short while I developed a strong desire to be a VLSI circuit designer. To position myself to enter circuit design, I entered the master's program at The University of Vermont. With IBM as my sponsor, I studied VLSI design and

manufacturing, computer architecture, and compiler design on a part-time basis.

Also around this time (1992), IBM asked me to lead an effort to reduce the company's dependence on scan-based manufacturing testing of its microprocessors. I accepted this challenge as it afforded me the opportunity to learn about manufacturing test and behavioral model development and broaden my skills as a leader. In a little over three years, I completed this effort, culminating in the successful fault simulation of a dual-architecture microprocessor. That same year, 1996, I completed my studies and received a master's degree in electrical engineering from The University of Vermont.

Soon thereafter, IBM focused its attention on significantly increasing the operational frequency of its server microprocessors. An opportunity presented itself in the form of a fledgling processor development team in need of designers. I seized the chance to move into a circuit design position at the IBM Advanced Semiconductor Technology Center in New York.

I worked as a circuit designer for nearly four years, designing a portion of its BFP execution unit. This microprocessor, Power4, was made generally available with an operating frequency in excess of 1.0 GHz. Due to its high frequency, the project presented significant challenges. In spite of the success of these products and the satisfaction I gained while working on this program, I once again found myself with the desire to continue my education.

In the fall of 2000, I approached my management team with a request to pursue my doctorate. IBM agreed to sponsor my studies full-time at Lehigh University, and I entered the computer engineering program in the fall of 2001. I completed all my course work and passed my qualification exam in two years. Shortly thereafter, I went back to work full-time, assisting with the circuit design of a server processor and, subsequently, the logic design of a mainframe processor. What is noteworthy

about these processors, which were announced in the IBM p590 and z10 machines, is that both of these supported DFP in hardware. Over the last year and a half, I have been designing and developing circuitry surrounding the instruction cache related to instruction fetch and decode.

While working full-time the past four years, I have performed and published additional research, culminating with this dissertation.