

**DESIGN AND SYNTHESIS OF
EFFICIENT MAC ARCHITECTURES FOR
HIGH SPEED DECIMAL PROCESSOR**

Thesis submitted by

REKHA K. JAMES

for the award of the degree of

DOCTOR OF PHILOSOPHY

Under the guidance of

K. POULOSE JACOB, Ph.D. and SREELA SASI, Ph.D.



**DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF TECHNOLOGY**

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

KOCHI – 682 022

INDIA

JANUARY 2010

**DESIGN AND SYNTHESIS OF
EFFICIENT MAC ARCHITECTURES FOR
HIGH SPEED DECIMAL PROCESSOR**

Ph.D. thesis in the field of Architectures for Decimal Computations

Author

REKHA K. JAMES
Research Scholar
Department of Computer Science
Cochin University of Science and Technology
Kochi-682022, Kerala, India
Email: rekhajames@cusat.ac.in

Research Advisors
K. POULOSE JACOB, Ph.D.
Professor and Head
Department of Computer Science
Cochin University of Science and Technology
Kochi-682022, Kerala, India
Email: kpj@cusat.ac.in

SREELA SASI, Ph.D.
Associate Professor
Department of Computer & Information Science
Gannon University
PA, USA
Email: sasi001@gannon.edu

January 2010

Dedicated

to

My Family

Certificate

*This is to certify that the thesis entitled “**DESIGN AND SYNTHESIS OF EFFICIENT MAC ARCHITECTURES FOR HIGH SPEED DECIMAL PROCESSOR**” is a bonafide record of the research work carried out by Ms. Rekha K. James under my supervision in the Department of Computer Science, Cochin University of Science and Technology, Kochi with Dr. Sreela Sasi, Associate Professor, Gannon University, PA, USA as co-guide. The results presented in this thesis or parts of it have not been presented for the award of any other degree.*



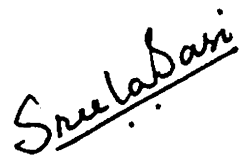
4th January 2010

K. POULOSE JACOB, Ph.D.

(Supervising Guide)
Professor and Head
Department of Computer Science
Cochin University of Science and Technology
Kochi-682022, Kerala

CERTIFICATE

*This is to certify that the thesis entitled “**DESIGN AND SYNTHESIS OF EFFICIENT MAC ARCHITECTURES FOR HIGH SPEED DECIMAL PROCESSOR**” is a bonafide record of the research work carried out by Ms. Rekha K. James under the supervision and guidance of Dr. K. Poulouse Jacob, Professor and Head, Department of Computer Science, Cochin University of Science and Technology, Kochi with myself as co-guide. The results presented in this thesis or parts of it have not been presented for the award of any other degree.*



4th January 2010

SREELA SASI, Ph.D.

Associate Professor

Department of Computer & Information Science

Gannon University, PA, USA

Declaration

*I hereby declare that the work presented in this thesis entitled **“DESIGN AND SYNTHESIS OF EFFICIENT MAC ARCHITECTURES FOR HIGH SPEED DECIMAL PROCESSOR”** is based on the original research work carried out by me under the supervision and guidance of **Dr. K. Poullose Jacob**, Professor and Head, Department of Computer Science, Cochin University of Science and Technology with **Dr. Sreela Sasi**, Associate Professor, Gannon University, PA, USA as co-guide. The results presented in this thesis or parts of it have not been presented for the award of any other degree.*

Kochi - 682022

4th January 2010



REKHA K. JAMES

Acknowledgement

I am deeply indebted and grateful to many people who supported me during the research work and preparation of the thesis.

First and foremost, I give special thanks and glory to the God Almighty for giving me the wisdom and health to complete this endeavour.

I would like to express sincere gratitude and appreciation to my supervising guide Dr. K. Poulose Jacob, Professor and Head, Department of Computer Science, Cochin University of Science and Technology for his constant encouragement, support and guidance. His sincerity, positive and supportive attitude, calmness and scholarly advice have been a steady source of inspiration to me.

My deepest gratitude and respect also goes to Dr. Sreela Sasi, Associate Professor, Department of Computer and Information Science, Gannon University, PA, USA for her guidance and assistance as co-supervisor. Her creative comments and suggestions from the initial conception till the completion of this work are highly appreciated. I am greatly indebted to her for the financial assistance which enabled me to register for several international conferences, and also for the strenuous effort in reviewing the research papers and the thesis.

I am thankful to Dr. R. Gopikakumari, Head, and all my colleagues in Division of Electronics Engineering, School of Engineering, Cochin University of Science and Technology for their encouragement and support. I am very grateful to Dr. Binu Paul, Dr. Mridula S., Dr. Shahana T. K., Dr. Sheena Mathew for their cooperation, support and care which helped me to pursue the research. I acknowledge the contribution of the technical and non-technical staff in the Department of Computer Science, Cochin University of Science and Technology.

I owe heartfelt thanks to my parents and my mother-in-law for their motivation, encouragement and understanding when it was mostly required. I specially mention my husband George Raphael, and my daughters Shilpa and Alka for their love, understanding, support and encouragement that helped me to fulfil my dream.

REKHA K. JAMES

ABSTRACT

Most of the commercial and financial data are stored in decimal form. Recently, support for decimal arithmetic has received increased attention due to the growing importance in financial analysis, banking, tax calculation, currency conversion, insurance, telephone billing and accounting. Performing decimal arithmetic with systems that do not support decimal computations may give a result with representation error, conversion error, and/or rounding error. In this world of precision, such errors are no more tolerable. The errors can be eliminated and better accuracy can be achieved if decimal computations are done using Decimal Floating Point (DFP) units. But the floating-point arithmetic units in today's general-purpose microprocessors are based on the binary number system, and the decimal computations are done using binary arithmetic. Only few common decimal numbers can be exactly represented in Binary Floating Point (BFP). In many cases, the law requires that results generated from financial calculations performed on a computer should exactly match with manual calculations. Currently many applications involving fractional decimal data perform decimal computations either in software or with a combination of software and hardware. The performance can be dramatically improved by complete hardware DFP units and this leads to the design of processors that include DFP hardware.

However, the hardware implementation for decimal operations has been limited due to the increase in cost and complexity. VLSI implementations using same modular building blocks can decrease system design and manufacturing cost. A multiplexer realization is a natural choice from the viewpoint of cost and speed. By suitable selection of variables or

functions as control inputs, the number of modules and/or delay can be reduced for realizing logic functions.

Although DFP arithmetic is well-suited for the financial computations, it occupies more area than binary leading to more power when implemented in hardware. Low power designs with high performance are given prime importance, since power has become an important design consideration. While efforts are being made to reduce power dissipation due to leakage currents, alternate circuit design considerations are also gaining importance. In recent years, reversible logic has emerged as one of the most important approaches for power optimization. So, reversible logic is in demand for high-speed power aware circuits.

This thesis focuses on the design and synthesis of efficient decimal MAC (Multiply ACcumulate) architecture for high speed decimal processors based on IEEE Standard for Floating-point Arithmetic (IEEE 754-2008). The research goal is to design and synthesize decimal MAC architectures to achieve higher performance. The main objectives of the research are to:

- Design a new fixed point iterative double digit decimal multiplier to improve performance compared to single digit decimal multiplier
- Develop a novel algorithm for fixed point iterative decimal multiplier to increase speed
- Design an improved Binary Coded Decimal (BCD) multiplier for digit by digit multiplication of significand digits of Decimal Floating Point (DFP) inputs
- Design modified parallel fixed point decimal multipliers to attain high speeds

- Design floating point decimal multipliers using modified iterative and parallel fixed point decimal multipliers
- Design different implementations of DFP adders aiming at reducing the delay
- Implement DFP MAC with fused multiply-add architectures using the modified multiplier designs and DFP adder units for high performance decimal processors
- Design optimized reversible logic implementation of BCD adders in terms of number of reversible gates and garbage outputs
- Design reversible decimal adders suitable for multi-digit BCD addition using Fredkin gates (FRG) and Toffoli gates (TG)
- Develop a new algorithm to implement any given logic function using only multiplexers to decrease system design and manufacturing cost

Efficient design methods and architectures are developed for a high performance DFP MAC unit as part of this research. The major achievements of the research are the following.

- The speed of fixed point decimal multiplier in DFP unit is increased by using double digit decimal multiplier.
- A novel RPS algorithm for fixed point iterative decimal multiplier is developed to increase speed. In this approach, partial products generated using BCD digit multipliers are accumulated from the least significant end in a column manner.
- An improved BCD multiplier is designed with reduced area and delay for digit by digit multiplication of significant digits of DFP inputs. Improved

BCD digit multipliers are used in the iterative decimal fixed point multiplier that employs novel RPS algorithm.

- Modified parallel decimal fixed point multiplier design is done using both row and column partial product accumulations. The column accumulation approach gives a decrease of area and delay over the row accumulation method.
- The iterative DFP multipliers are designed using floating point extensions of the iterative decimal fixed point multiplier using double digit decimal multiplier and RPS algorithm. A delay reduction is achieved using RPS algorithm because of the initiation of rounding process during the fixed point multiplication.
- The parallel DFP multiplier is designed with the floating point extensions of the parallel fixed point multiplier design and is compared with the iterative designs using double digit decimal multiplier and using RPS algorithm.
- Floating point MAC unit implements the fused multiply-add operation with a single final rounding after add operation. The fused multiply add unit uses parallel and iterative multipliers and a floating point adder unit. The DFP adder is implemented using ripple carry BCD adders, kogge stone adders and reduced delay BCD adders.
- The modified reversible BCD adder implementations presented are highly optimized in terms of number of reversible gates and garbage outputs.
- The performance comparison of carry select and hybrid BCD adders with conventional BCD adder is also done. It shows that the hybrid BCD adder attains speed up over carry select and conventional BCD adders, for any

input length in a reversible implementation; unlike in classical logic gate implementation.

- The performance comparison of VLSI implementations of different BCD adders reveals that the implementations using Toffoli gates are superior in terms of quantum cost, garbage count and gate count, compared to Fredkin gate implementations.
- The new exhaustive branching algorithm is developed to obtain reduced hardware and/or delay for synthesizing logic functions using multiplexers.

Contents

	<i>Page No:</i>
LIST OF FIGURES	v
LIST OF TABLES	xi
LIST OF ABBREVIATIONS	xv
1. INTRODUCTION	1
1.1 Decimal arithmetic	3
1.2 Early Computer Arithmetic Systems	7
1.3 Decimal Encodings	10
1.3.1 Trailing Significant Field	12
1.3.1.1 Densely Packed Decimal Encoding	13
1.3.2 Combination Field	15
1.4 Overview of Research	16
1.5 Layout of the Thesis	18
2. DECIMAL FIXED POINT MULTIPLICATION	21
2.1 Decimal Fixed point multiplication	23
2.2 Iterative DFxP Multipliers	23
2.2.1 Double Digit Decimal Multiplication (DDDM)	26
2.2.1.1 Secondary Multiple Generation Block	26
2.2.1.2 Multiplier Shift Register	29
2.2.1.3 Multiplexer Block	29
2.2.1.4 Carry Save Adder Block	31
2.2.1.5 Partial Product Register	33

2.2.1.6	Decimal Carry Propagate Adder	33
2.2.2	Decimal Fixed Point Multiplication using RPS Algorithm	36
2.2.2.1	BCD Digit Multiplier	36
2.2.2.2	Binary Multiplier	37
2.2.2.3	6-bit Binary to BCD Converter	39
2.2.2.4	Hex/Decimal Multiplier	42
2.2.2.5	Multi-operand Decimal Adders	44
2.2.2.6	RPS Algorithm	46
2.3	Parallel DFXP Multipliers	56
2.3.1	Row Accumulation	56
2.3.2	Column Accumulation	60
2.4	Summary	62
3.	DECIMAL FLOATING POINT MULTIPLIERS AND MAC UNIT	65
3.1	Decimal Floating Point Multipliers	67
3.2	DFP Multiplication using RPS algorithm	68
3.3	DFP Multiplication using DDDM	74
3.4	DFP Multiplication using Parallel DFXP Multiplier	75
3.5	DFP MAC Unit	75
3.5.1	DFP Adders	77
3.5.1.1	Ripple Carry BCD Adder	79
3.5.1.2	Kogge-Stone Adder	80
3.5.1.3	Reduced Delay BCD Adder	81
3.6	Summary	84
4.	REVERSIBLE CIRCUITS FOR DECIMAL ADDERS	87
4.1	Reversible Logic	89

4.2	Reversible Gates	90
4.3	Reversible Full Adders	93
4.4	Reversible Decimal Adders	99
4.5	Reversible Fast Decimal Adders	104
4.5.1	Quick Decimal Adder	104
4.5.1.1	Parity Preserving Reversible Quick Decimal Adder	107
4.5.2	Carry Select BCD Adder	110
4.5.2.1	Parity Preserving Reversible Carry Select BCD Adder	112
4.5.3	Hybrid BCD Adder	113
4.5.3.1	Hybrid Reversible BCD Adder	115
4.5.4	Toffoli Gate Implementation	116
4.6	New Reversible RPS Gate	124
4.6.1	Fully Reversible RPS Gate	125
4.6.2	4-bit Binary to BCD Converter using RPS Gate	126
4.6.3	4-bit Binary to BCD Converter using other existing Reversible Gates	128
4.6.4	4-bit Binary to BCD Converter using HNG Gate	128
4.6.5	Partially Reversible RPS Gate	129
4.6.6	Reversible Implementations of BCD Adder	131
4.6.6.1	BCD Adder using HNG-RPS Gates	132
4.6.6.2	BCD Adder using RPS Gates	133
4.6.6.3	BCD Adder using HNG Gates	134
4.7	Reversible Error Correcting Code Generation and Detection	134
4.8	Summary	140
5.	LOGIC SYNTHESIS USING MULTIPLEXERS	143
5.1	Delay Reduced Combinational Logic Synthesis using Multiplexers	145
5.2	N-ary Exhaustive Branching Technique	146

5.3	Exhaustive Branching Technique	149
5.4	Summary	152
6.	SIMULATION RESULTS AND ANALYSIS	155
6.1	Simulation Results and Analysis	157
6.2	Simulation Results of Decimal Fixed Point Multipliers	157
6.2.1	DDDM: Synthesis and analysis	157
6.2.2	Simulation Results of BCD Digit Multiplier	169
6.2.3	Simulation Results of DFXP Multiplication using RPS Algorithm	170
6.2.4	Simulation Results for Parallel Decimal Multipliers	172
6.3	Simulation Results for Decimal Floating Point Units	174
6.3.1	Simulation Results for Decimal Floating Point Multipliers	174
6.3.2	Simulation Results of Decimal Floating Point MAC Unit	181
6.4	Performance Comparison of Reversible Circuits for Decimal Adders	183
6.5	Logic Synthesis Simulation using Multiplexers	194
6.6	Summary	201
7.	CONCLUSION AND FUTURE WORK	205
7.1	Conclusion	207
7.2	Suggestions for Future Work	211
	REFERENCES	213
	LIST OF PUBLICATIONS OF THE AUTHOR	
	APPENDIX	
	INDEX	

LIST OF FIGURES

		Page No:
Figure 1.1	Survey on Commercial databases	3
Figure 1.2	Decimal floating-point format	11
Figure 2.1	Block Schematic of Decimal Fixed Point Multiplication	23
Figure 2.2	Block Diagram of the Fixed Point DDDM	27
Figure 2.3	Secondary Multiple Generation	29
Figure 2.4	Multiplexer Block	30
Figure 2.5	Decimal Incrementer	34
Figure 2.6	Block Diagram for the Partitioned DDDM for 34-digits	35
Figure 2.7	Conventional Paper and Pencil View of Digit Multiplication	36
Figure 2.8	3×3 Multiplication	38
Figure 2.9	4×3 Multiplication of BCD inputs	38
Figure 2.10	4×4 Multiplication of BCD inputs	38
Figure 2.11	4×4 Multiplication of BCD inputs generating 8-bit BCD output	39
Figure 2.12	The principle of 6-bit binary to BCD conversion	40
Figure 2.13	Addition of first and last lower BCD digits	40
Figure 2.14	Addition of other two lower BCD digits to get a BCD Sum	41
Figure 2.15	6-bit Binary to BCD converter	41
Figure 2.16	BCD Digit Multiplier	42
Figure 2.17	4×4 Hex Multiplication	43
Figure 2.18	Hex/Decimal multiplier	44
Figure 2.19	Partial product generation and accumulation in different cycles	49
Figure 2.20	One-digit, 15-operand Non-speculative Adder	50
Figure 2.21	One-digit, 5-operand Non-speculative Adder	51
Figure 2.22	DFxP multiplier using RPS algorithm	52
Figure 2.23	Decimal Fixed Point multiplier controller block	52
Figure 2.24	Register array for storing output of BCD-digit multiplication	53
Figure 2.25	Selecting inputs for multi-operand BCD addition	54
Figure 2.26	BCD adder array	55

Figure 2.27	Partial Product accumulation using carry save adders	57
Figure 2.28	Partial Product accumulation of 7-digit \times 7-digit multiplier	58
Figure 2.29	Decimal 4:2 Compressor Block for a 7-digit \times 7-digit multiplier	59
Figure 2.30	Column Adder array for 16-operand BCD addition	60
Figure 2.31	Adder array for accumulating partial products column wise	61
Figure 3.1	Decimal Floating Point (DFP) Multiplier	68
Figure 3.2	Block Schematic of DFxP Multiplier using RPS and Rounding Unit	73
Figure 3.3	Block diagram of a Decimal Floating Point MAC unit	76
Figure 3.4	Block diagram of a Decimal Floating Point Adder unit	78
Figure 3.5	Concatenated BCD Adders	79
Figure 3.6	Kogge –Stone Adder	80
Figure 3.7	Adder and Analyzer Unit	82
Figure 3.8	Adder, Analyzer and Carry Network	82
Figure 3.9	Reduced Delay BCD Adder	83
Figure 4.1	3 \times 3 New Gate (NG)	90
Figure 4.2	Toffoli Gate (TG)	91
Figure 4.3	3 \times 3 New Toffoli Gate (NTG)	91
Figure 4.4	4 \times 4 TS Gate (TSG)	92
Figure 4.5	Fredkin Gate (FRG)	92
Figure 4.6	2 \times 2 Feynman Gate (FG)	93
Figure 4.7	3 \times 3 Feynman double Gate (F2G)	93
Figure 4.8	A reversible full adder with NG and NTG	94
Figure 4.9	Reversible full adder using NG	95
Figure 4.10	Reversible full adder using NTG	95
Figure 4.11	Reversible full adder using TG	96
Figure 4.12	Reversible full adder using TSG	96
Figure 4.13	Reversible Full adder using Fredkin Gates	97
Figure 4.14	BCD Adder	99
Figure 4.15	Reversible Implementation of 6-correction Circuit	100

Figure 4.16	Reversible Implementation of Special Adder	101
Figure 4.17	Reversible Implementation of BCD Adder	102
Figure 4.18	Modified Implementation of BCD Adder	102
Figure 4.19	4×4 HN Gate (HNG)	103
Figure 4.20	BCD adder for Quick Addition of Decimals (QAD)	105
Figure 4.21	4-Digit Quick Decimal Adder	106
Figure 4.22	Half adder using Fredkin Gates	107
Figure 4.23	Full adder using Fredkin Gates	108
Figure 4.24	Generation of 'L' bit using Fredkin Gates	109
Figure 4.25	Generation of 'K' bit using Fredkin Gates	109
Figure 4.26	Carry Select BCD adder	111
Figure 4.27	Generation of 'K' bit using k_0 and k_1	112
Figure 4.28	Hybrid N-digit Decimal Adder	114
Figure 4.29	Half Adder	117
Figure 4.30	Full Adder	117
Figure 4.31	4-bit Binary Adder using Toffoli Gates	118
Figure 4.32	6-Correction Circuit using Toffoli Gates	119
Figure 4.33	Modified Special Adder of Conventional BCD Adder	119
Figure 4.34	Toffoli Gate implementation of Conventional BCD Adder	120
Figure 4.35	4-bit Binary Adder for QAD	121
Figure 4.36	K-bit Generation using Toffoli Gates	122
Figure 4.37	Toffoli implementation of Special Adder for QAD	123
Figure 4.38	Toffoli Gate Implementation of QAD	123
Figure 4.39	K-bit generation of Carry select BCD Adder	124
Figure 4.40	4 × 4 Fully reversible RPS gate	126
Figure 4.41	4-bit Binary to BCD converter using Fully Reversible RPS gate	127
Figure 4.42	4-bit Binary to BCD converter using HNG and NG	128
Figure 4.43	4-bit Binary to BCD converter using only HNG gates	129
Figure 4.44	4 × 4 Partially Reversible RPS Gate	130

Figure 4.45	Reversible Implementation of Conventional BCD adder	133
Figure 4.46	Conventional BCD adder using RPS gates	133
Figure 4.47	Conventional BCD adder using HNG	134
Figure 4.48	Reversible 4 x 4 HCG	136
Figure 4.49	Reversible 4 x 4 PPHCG	136
Figure 4.50	Reversible (7, 4) Hamming code generator using HCG	137
Figure 4.51	Reversible (7, 4) Hamming code generator using F2G	137
Figure 4.52	(7, 4) Hamming Code Generator using parity preserving gates	138
Figure 4.53	(7, 4) Hamming Code Generator using F2G	138
Figure 4.54	Reversible (7, 4) Hamming code error detector using HCG	139
Figure 4.55	Reversible (7, 4) Hamming code error detector using F2G	140
Figure 5.1	1-control line multiplexer module	147
Figure 5.2	Implementation of a 9-variable function using exhaustive branching technique	149
Figure 6.1	Area for different blocks of Double Digit Decimal Multiplier (7-digit × 7-digit)	158
Figure 6.2	Delay for different blocks of Double Digit Decimal Multiplier (7-digit × 7-digit)	158
Figure 6.3	Area for different blocks of Double Digit Decimal Multiplier (16-digit × 16-digit)	160
Figure 6.4	Delay for different blocks of Double Digit Decimal Multiplier (16-digit × 16-digit)	160
Figure 6.5	Area for different blocks of Double Digit Decimal Multiplier (34-digit × 34-digit)	161
Figure 6.6	Delay for different blocks of Double Digit Decimal Multiplier (34-digit × 34-digit)	162
Figure 6.7	Area for various blocks of Double Digit Decimal Multipliers (7, 16, & 34-digits)	162
Figure 6.8	Delay for various blocks of DDDM (7, 16 & 34 digits)	163

Figure 6.9	Area comparison of DDDM and SDDM	165
Figure 6.10	Delay comparison of DDDM and SDDM	165
Figure 6.11	Delay Break up of different components of DFP Multiplier using RPS Algorithm	176
Figure 6.12	Delay Break up of DFxP Multiplier (using RPS) and Rounding Unit	177
Figure 6.13	Delay break up of 32-bit DFP Multiplier using RPS Algorithm	177
Figure 6.14	Delay analysis of Conventional, Carry Select and Hybrid BCD Adders using classical logic gates	185
Figure 6.15	Analysis of area-delay product of Conventional, Carry Select and Hybrid BCD Adders using classical logic gates	185
Figure 6.16	Speed up factor for Carry select and Hybrid BCD adders vs. conventional BCD adder for classical logic gates	186
Figure 6.17	Delay analysis of reversible BCD adders using Fredkin gates	187
Figure 6.18	Speed up factor for reversible implementations of fast BCD adders vs. Conventional BCD adder	188
Figure 6.19	Optimum block size of Hybrid reversible BCD adder for different input lengths	189
Figure 6.20	Exhaustive branched network implementation for $F = \sum (4, 7, 9, 10, 12, 13, 14, 15)$	197
Figure 6.21	Tree implementation for $F = \sum (4, 7, 9, 10, 12, 13, 14, 15)$	197
Figure 6.22	Exhaustive branched network implementation for $F = \sum (3, 5, 7, 9, 11, 15)$	198
Figure 6.23	Tree implementation for $F = \sum (3, 5, 7, 9, 11, 15)$	198
Figure 6.24	Exhaustive branched network implementation for $F = \sum (3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$	199
Figure 6.25	Tree implementation for $F = \sum (3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$	199

LIST OF TABLES

		Page No:
Table 1.1	Error caused in decimal computations done using binary arithmetic	4
Table 1.2	Early Computer Systems	7
Table 1.3	Contemporary Processor Support of Decimal Arithmetic	9
Table 1.4	Decimal Format Parameters	12
Table 1.5	Compression: (abcd)(efgh)(ijklm) to (pqr)(stu)(v)(wxy)	14
Table 1.6	Expansion: (pqr)(stu)(v)(wxy) to (abcd)(efgh)(ijklm)	14
Table 1.7	Encoding format for 5 bits of Combination field	15
Table 2.1	Recoding of Digits of b_i	30
Table 2.2	Sum and carry correction values for 5-operand BCD adder	51
Table 3.1	Selection of the Value to be added for Correction	83
Table 4.1	Comparison of Reversible Full Adders	98
Table 4.2	Truth Table of Fully Reversible RPS Gate	126
Table 4.3	Truth Table of 4-Bit Binary to BCD Converter	127
Table 4.4	Truth Table of the Partial Reversible RPS Gate	130
Table 4.5	Truth Table of the 4 X 4 HCG	136
Table 4.6	Truth Table of the PPHC Gate	136
Table 6.1	Area and Delay for various blocks of Double Digit Decimal Multiplier (7-digit \times 7-digit)	157
Table 6.2	Area and Delay for various blocks of Double Digit Decimal Multiplier (16-digit \times 16-digit)	159
Table 6.3	Area and Delay for various stages of Double Digit Decimal Multiplier (34-digit \times 34-digit)	161
Table 6.4	Area and Delay for various blocks of 34-digit partitioned Double Digit Decimal Multiplier	163
Table 6.5	Comparisons for different designs of 34-digit Decimal Double Digit Multipliers	164

Table 6.6	Comparison of DDDM and SDDM	164
Table 6.7	Simulation results of Double Digit Decimal Multipliers (7-digit) on FPGAs	166
Table 6.8	Simulation results of Double Digit Decimal Multipliers (16-digit) on FPGAs	167
Table 6.9	Simulation results of Double Digit Decimal Multipliers (34-digit) on FPGAs	168
Table 6.10	Comparison of area and delay of BCD digit multipliers	169
Table 6.11	Area and Delay for different stages of Decimal Fixed Point Multiplier using RPS Algorithm (7-digit \times 7-digit)	171
Table 6.12	Area and Delay for different stages of Decimal Fixed Point Parallel Multipliers	173
Table 6.13	Comparison of DFxP Parallel Multipliers for different lengths	173
Table 6.14	Area and Delay for different stages of Decimal Fixed Point Multiplier (7-digit \times 7-digit) Column Accumulation	174
Table 6.15	Area and Delay for different stages of DFP Multiplier using RPS algorithm (32-bit)	175
Table 6.16	Area and Delay of Rounding Unit and DFxP Multiplier using RPS algorithm (7-digit \times 7-digit)	175
Table 6.17	Comparison of DFxP multipliers using RPS algorithm with existing one (7-digit \times 7-digit)	178
Table 6.18	Comparison of DFP Multipliers using RPS algorithm with existing one (32-bit \times 32-bit)	179
Table 6.19	Comparison of Iterative DFP Multipliers for 32-bit input	180
Table 6.20	Comparison of DFP Multipliers (using parallel, DDDM & of Erle) for 32-bit Input	180
Table 6.21	Comparison of DFP Adders (16-digit)	181
Table 6.22	Comparison of DFP MAC units (32-bit)	182
Table 6.23	Comparative Analysis of the Reversible BCD Adders	184

Table 6.24	Comparative analysis of Reversible BCD Adders Implemented using Toffoli and Fredkin Gates	190
Table 6.25	Comparison of Reversible 4-Bit Binary to BCD Converters	191
Table 6.26	Comparative Analysis of Reversible BCD Adders for logical complexity	192
Table 6.27	Comparison of Reversible Hamming code Generation and Detection Circuits	194
Table 6.28	4-bit binary minterm table	195
Table 6.29	The reduced minterm table for $x_3 = 0$	195
Table 6.30	The reduced minterm table for $x_3 = 1$	196
Table 6.31	The reduced minterm table for $(x_2 \oplus x_1) = 1$	196
Table 6.32	The reduced minterm table for $(x_2 \oplus x_1) = 0$	196
Table 6.33	Comparison in terms of delay and hardware for standard implementation, tree implementation and exhaustive branched network implementation	200

LIST OF ABBREVIATIONS

ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuit
BAP-SC	Bit-slice Arithmetic Processor for Scientific Computing
BCD	Binary Coded Decimal
BFP	Binary Floating Point
BID	Binary Integer Decimal
CADAC	Controlled-Precision Decimal Arithmetic Unit
CC	Carry Counter
CMOS	Complementary Metal Oxide Semiconductor
CPA	Carry Propagate Adder
CPU	Central Processing Unit
CSA	Carry Save Adder
DCA	Decimal Carry-save Adder
DDDM	Double Digit Decimal Multiplier
DFP	Decimal Floating Point
DFxP	Decimal Fixed Point
DG	Digit Generate
DP	Digit Propagate
DPD	Densely Packed Decimal
DSP	Digital Signal Processing
FA	Full Adder
FG	Feynman Gate
F2G	Feynman double Gate
FPGA	Field Programmable Gate Array
FRG	FRedkin Gate
GC	Garbage Count

HA	Half Adder
HCG	Hamming Code Gate
HNG	HN Gate
LSB	Least Significant Bit
LSD	Least Significant Digit
LUT	Look Up Table
MAC	Multiply ACcumulate
MSB	Most Significant Bit
MSD	Most Significant Digit
MSR	Multiplier shift Register
MUX	Multiplexer
NaN	Not a Number
NG	New Gate
NTG	New Toffoli Gate
PPHCG	Parity Preserving Hamming Code Gate
PPRM	Positive Polarity Reed Muller
QAD	Quick Addition of Decimals
QC	Quantum Cost
RBCD	Redundant Binary Coded Decimal
RC	Reversible Circuit
RM	Reed Muller
RTE	Round To Even
SEC	Single Error Correction
SDDM	Single Digit Decimal Multiplier
SMG	Secondary Multiplier Generation
TC	Temporary Carry
TG	Toffoli Gate
TPR	Temporary Product Register

TS	Temporary Sum
TSG	TS Gate
ULM	Universal Logic Module
UNIVAC	UNIVersal Automatic Computer
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Chapter 1

Introduction

This chapter explains the motivation behind the decimal computer arithmetic. It brings out the importance of decimal arithmetic over binary arithmetic for financial, commercial and all other applications where data is given in decimal form. The chapter also explores the history of computer arithmetic systems. Different decimal encodings for IEEE 754- 2008 standards are discussed as well. The chapter also gives an overview of the research work detailed in the thesis.

1.1 Decimal arithmetic

Most of world's commercial and financial data are stored in decimal form. A survey done on commercial databases show that most data are decimal as shown in Figure 1.1 [A. Tsang and M. Olschanowsky, 1991].

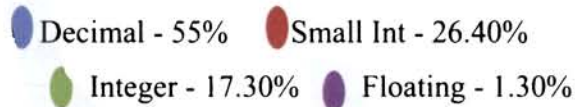
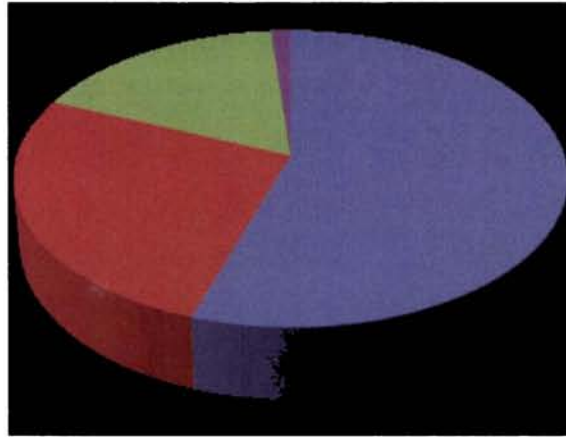


Figure 1.1: Survey on Commercial databases

Currently, decimal computations are done using binary arithmetic by general purpose computers. Many early computers (such as ENIAC and IBM 650) were decimal machines [IBM Decimal arithmetic –FAQ]. However, in the 1950s, most computers turned to binary representations of numbers as this made useful reductions in the complexity of arithmetic units. This reduction in turn led to greater reliability at lower costs. This is because binary data can be stored efficiently and processed quickly on two-state computers.

Binary arithmetic is appropriate for mathematical analysis and requires less hardware to implement the same function. Even then there are compelling reasons to consider decimal arithmetic, particularly for business computations. The reasons include human's natural affinity for decimal arithmetic and the inexact mapping between some decimal and binary values. Binary floating-point (BFP) values cannot represent all decimal numbers. For example a value of 0.1 requires an infinitely recurring binary pattern of zeros and ones while, a decimal number system can represent 0.1 exactly, as one tenth (that is, 10^{-1}). When an average user performs a calculation such as addition of 0.1 and 0.9, the expected result is 1.0. The user would find it very confusing to be presented with an answer of 0.999999. Errors caused in binary results of repeatedly dividing 9 by ten is a good example of the problem of using binary arithmetic for decimal computations as shown in Table 1.1[IBM, Decimal arithmetic - FAQ]

Table 1.1: Error caused in decimal computations done using binary arithmetic

Decimal	Binary
0.9	0.9
0.09	0.0899999996
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	8.99999996E-10

Binary calculations can apparently make some predictable results erroneous. Such errors may accumulate unnoticed and then surface after repeated operations. Hence, binary arithmetic is not suitable for financial, commercial, and human-centric applications or for any calculations where the results achieved are required to match those which are calculated by hand. Due to these reasons, calculations are to be carried out using decimal arithmetic for data which are in decimal form. Moreover, in many cases, the law requires that results generated from financial calculations performed on a computer exactly match with manual calculations. The legal requirements (for example, in Euro regulations) demand the working precision in decimal digits and rounding method to decimal digits to be used for calculations. All these requirements can only be met by radix 10 arithmetic which preserves precision. Recently, decimal arithmetic has received increased attention due to this growing importance in financial analysis, banking, tax calculation, currency conversion, insurance, telephone billing and accounting. In engineering, exact measurements are often kept in a decimal form and processing such values in binary can lead to inaccuracies. This was the cause of the Patriot missile failure in 1991, when a missile failed to track and intercept an incoming Scud missile. The error was caused by multiplying a time (measured in tenths of a second) by 0.1 (approximated in binary floating-point) to calculate seconds [M. Blair *et al.* 1992]. This makes it difficult to develop and test applications that use exact real-world data using binary floating point arithmetic. But there are some disadvantages of using decimal arithmetic over binary. Decimal numbers are traditionally held in a Binary Coded Decimal (BCD) form that uses more storage than a purely binary representation. Calculations in decimal can therefore require more circuitry

than pure binary calculations, and will typically be slower. Currently, binary floating-point is usually implemented by the hardware in a computer, whereas decimal floating-point is implemented in software. This means that decimal computations are typically about 100 to 1000 times slower than binary operations [IBM Decimal arithmetic –FAQ].

Hardware support for decimal operations has been limited. This is because decimal arithmetic operations are more complex and occupy more area leading to more power and less speed compared to binary arithmetic when implemented in hardware. But the scenario is set to change with the cost of die space continually dropping and the significant speedup achievable in hardware. Yet there is little in the way of hardware assist that perform operations on data stored in decimal form.

The relevance of research in the area of decimal computer arithmetic, is emphasized by the following observations. There are a number of established computer languages now supporting Decimal floating point (DFP) arithmetic, including C/C++, COBOL, Java, PERL, Python etc [M. A. Erle, 2008]. Further the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985) and the IEEE Standard for Radix-Independent Floating-Point Arithmetic (IEEE 854-1987) have been merged, expanded and approved as a new standard named the IEEE Standard for Floating-Point Arithmetic or IEEE 754-2008 standard. Finally the decreasing cost of die space [G. E. Moore, 1965] allows new features to be added such as the introduction of decimal arithmetic.

1.2 Early Computer Arithmetic Systems

The earliest electronic computers, for example ENIAC [H. H. Goldstine and A. Goldstine, 1996], UNIVAC [R. Head, 2001], and IBM 650 [D. E. Knuth, 1986], performed arithmetic functions in base ten [N. Stern, 1981]. But the EDSAC [M. V. Wilkes, 1997], EDVAC [M. R. Williams, 1993], and the ORDVAC [T. Leser and M. Romanelli, 1956] used binary base. 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)). Table 1.2 provides list of early computer systems [M. A. Erle, 2008].

Table 1.2: Early Computer Systems

Year	System	Base
1913	analytical engine [B. Randell, 1982]	decimal
1938	Z1	binary
1939	ABC [J. V. Atanasoff, 1984]	binary
1943	Colossus [J. Copeland, 2004]	binary
1944	IBM ASCC/MARK I [R. Campbell, 1999]	binary
1945	ENIAC [H. H. Goldstine and A. Goldstine, 1996]	decimal
1949	EDSAC [M. V. Wilkes, 1997]	binary
1951	UNIVAC I [R. Head, 2001]	decimal
1952	EDVAC [M. R. Williams, 1993]	binary
1952	ORDVAC [T. Leser and M. Romanelli, 1956]	binary
1953	IBM 650 [D. E. Knuth, 1986]	decimal
1956	UNIVAC [R. Head, 2001]	decimal
1959	NEC NEAC 2201	decimal
1960	UNIVAC LARC	decimal
1961	IBM 7030 (Stretch) [W. B. <i>et al.</i> , 1962]	binary
1964	IBM System/360 [G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, 1964]	binary
1964	CDC 6600 [J. E. Thornton, 1980]	binary

Some of the more recent computer systems providing hardware support of decimal arithmetic described in [M. A. Erle, 2008] are given below. A Controlled-Precision Decimal Arithmetic Unit (CADAC) is described in [M. S. Cohen, T. E. Hull, and V. C. Hamacher, 1983]. In [T. E. Hull, M. S. Cohen, and C. B. Hall, 1991], this work is extended with a software system that supports variable-precision decimal arithmetic. Even though consideration is given to the hardware for exception handling, it does not give any drastic improvement in the actual hardware implementation. In [G. Bohlender and T. Teufel, 1987], a Bit-slice Arithmetic Processor for Scientific Computing (BAP-SC) is described that implements DFP ADD, SUBTRACT, MULTIPLY, and DIVIDE and provide significant acceleration over pure software solutions. Several commercial microprocessors offer some fixed-point BCD arithmetic instructions. The Intel x86 processor series offers eight decimal instructions, the Motorola 68k processor series provides five decimal instructions, and the HP PA-RISC processor series offers two decimal instructions [G. Kane, 1996]. 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. 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 is found in IBM's mainframes, such as the S/390 and System z900 [F. Y. Busab *et al.*, 2001]. Support for DFP arithmetic can be found in IBM's System z9 [A. Y. Duale *et al.*, 2007] and System z10 [E. M. Schwarz, J. S. Kapernick, and M. F. Cowlshaw, 2009] mainframes, and in IBM's Power6 [L. Eisen *et al.*, 2007] server. The S/390 processor offers decimal fixed point ADD, SUBTRACT, MULTIPLY, and DIVIDE via a

dedicated decimal adder employing the direct decimal addition scheme. The MULTIPLY, and DIVIDE instructions involve iterative additions and subtractions. The System z900 processor offers the same 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. The fixed-point arithmetic unit on the z9 processor supports binary and decimal fixed-point operations. Recently, SilMinds corporation [SilMinds, 2008] has made available its synthesizable VHDL and Verilog code which performs DFP ADD, SUBTRACT, MULTIPLY [R. Eissa *et al.*, 2008], FUSED MULTIPLY-ADD, DIVIDE, and SQUARE ROOT in conformance with IEEE 754-2008. The customers can use these off-the-shelf codes when developing commercial processors or Application Specific Integrated Circuits (ASICs). Table 1.3 contains a list of contemporary processor support for decimal arithmetic [M. A. Erle, 2008].

Table 1.3: Contemporary Processor Support of Decimal Arithmetic

Processor	Support
Intel x86 family	instructions to correct binary +, -, ×, /
Motorola 68k family	instructions to correct binary +, -
HP PA-RISC family	instructions to correct binary +, -
Early IBM mainframes	firmware-assisted Decimal fixed point
IBM System z9	In compliance with IEEE 754-2008 firmware-assisted DFP
IBM Power6	In compliance with IEEE 754-2008 complete DFP hardware unit
IBM System z10	In compliance with IEEE 754-2008 extension of IBM Power6 DFP unit
SilMinds DFP cores	In compliance with IEEE 754-2008 partial implementation (+, -, ×, ×+, /, $x^{1/2}$)

The new hardware support is much faster than software, but is still somewhat slower than binary floating-point hardware. However, the programming and conversion overheads and other costs of using binary arithmetic suggest that hardware decimal arithmetic is now the more economical option for most applications.

1.3 Decimal Encodings

In the early days of electronic computers a great variety of both fixed-point and floating-point decimal encodings were used. Over the years, most of these encodings were abandoned, but the form of decimal encoding that endured is the dual-integer encoding. Dual-integer encodings describe a decimal number using two integers: a significand and an exponent. The value of a number encoded with these two parameters is $\text{significand} \times 10^{\text{exponent}}$. These encodings allow for a range of positive and negative values together with values of ± 0 , $\pm \text{Infinity}$, and Not-a-Number (NaN). Standard specifications for decimal representations are recently added to the IEEE 754-2008, which was officially approved in June 2008. The specifications include the choice of precision, exponent base and range, significand representation and encoding. These are approved for IEEE 754-2008 based on the considerations and reasoning presented by [Cowlshaw *et al.*, 2001] and the Decimal Subcommittee of the committee revising IEEE 754-1985.

Three formats of decimal floating point numbers suggested were:

- A decimal32 number, which is encoded in four consecutive bytes (32 bits)
- A decimal64 number, which is encoded in eight consecutive bytes (64 bits)

- A decimal128 number, which is encoded in 16 consecutive bytes (128 bits)

In IEEE 754-2008, the value of a finite DFP number with an integer significand is

$$v = (-1)^s \times C \times 10^q$$

where ‘S’ is the sign, ‘q’ is the unbiased exponent, and ‘C’ is the significand. The precision or the length of the significand is denoted as ‘p’, which is equal to 7, 16, or 34 digits, for decimal32, decimal64, or decimal128, respectively. Figure 1.2 shows the Decimal Floating Point format.

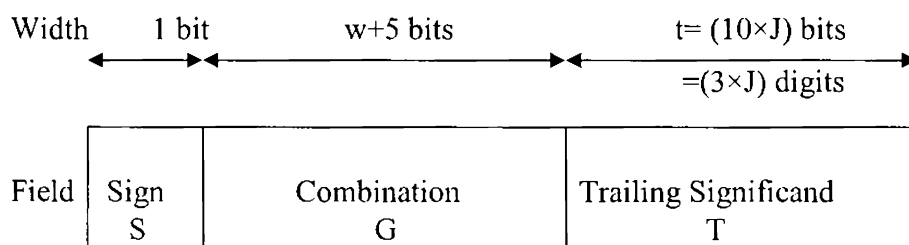


Figure 1.2: Decimal floating-point format

The 1-bit Sign Field, S indicates the sign of a number. The (w+5)-bit Combination Field, G provides the most significant digit (MSD) of the significand and a non-negative biased exponent, E such that $E = q + \text{bias}$. The exponent is almost always encoded in binary. The G Field also indicates special values, such as Not-a-Number (NaN) and infinity (∞). The remaining digits of the significand are specified in the t-bit Trailing Significand Field, T.

1.3.1 Trailing Significant Field

The number of bits in T field is an integer multiple of 10, indicated as $10 \times J$ where J is an integer that can be 2, 5 or 11 in the decimal32, decimal64 or decimal128 formats respectively.

IEEE 754-2008 specifies two encodings for the Trailing Significant Field. The first encodes its significand using a decimal encoding, also known as the Densely Packed Decimal (DPD) encoding based on the encoding of three BCD digits into ten bits [M. F. Cowlishaw, 2002]. The other encoding uses a binary integer significand, and is commonly referred to as the Binary Integer Decimal (BID) encoding. IEEE 754-2008 refers to the BID encoding as the binary encoding of DFP numbers and it refers to the DPD encoding as the decimal encoding of DFP numbers. This research makes use of DPD encoding.

Table 1.4 gives the important parameters for each decimal format. In this table, widths are given in bits, and *emax* and *emin* indicate the minimum and maximum unbiased exponents, respectively, in each format.

Table 1.4: Decimal Format Parameters

Type of Format	Decimal32	Decimal64	Decimal128
Storage width	32	64	128
Trailing Significand field width (t)	20	50	110
Number of significand digits (precision : p)	7 digits	16 digits	34 digits
Combination field width (w+5)	11	13	17
Exponent bias	101	398	6176
emax	+96	+384	+6144
emin	-95	-383	-6143

1.3.1.1 Densely Packed Decimal Encoding

Densely Packed Decimal (DPD) encoding, encodes three decimal digits in 10 bits, giving a 20% more efficient encoding than simple BCD (one digit in 4 bits). The specific encoding preserves much of the identity of the three decimal digits, and allows simple processing. The primary advantage of the encoding over a pure binary representation in ten bits is that no arithmetic is needed for conversion to or from BCD. In hardware, encoding or decoding can be achieved with only 2–3 gate delays. In addition, the encoding has other advantages. For example, the least-significant bit of each digit is retained as such. Compression of one or two decimal digits (into the optimal four or seven bits respectively) is achieved as a subset of the 3-digit encoding. The DPD encoding, considers each of the three digits as either being small (0–7, requiring 3 bits to distinguish) or large (8 or 9, requiring one bit).

The possible combinations of these ranges are then:

- All digits are small (51.2% of the possibilities). This requires 3+3+3 bits for the digits, leaving 1 bit to indicate this combination.
- Two digits are small (38.4%). This requires 3+3+1 bits for the digits, leaving 3 to indicate this combination.
- One digit is small (9.6%). This requires 3+1+1 bits for the digits, leaving 5 to indicate this combination.
- No digits are small (0.8%). This requires 1+1+1 bits for the digits, leaving 7 (only 5 are needed) for the indication.

The Tables 1.5 and 1.6 describe the encoding (compression from BCD) and decoding (expansion to BCD); the letters a–k and m represent the 12 bits of three BCD digits, and p–y represent the 10 bits of the encoded digits.

Table 1.5: Compression:(abcd)(efgh)(ijkm) to (pqr)(stu)(v)(wxy)

aei	pqr stu v wxy	Comments
000	bcd fgh 0 jkm	All digits are small
001	bcd fgh 1 00m	Right digit is large [keeps 0-9 unchanged]
010	bcd jkh 1 01m	Middle digit is large
100	jdk fgh 1 10m	Left digit is large
110	jdk 00h 1 11m	Right digit is small [L & M are large]
101	fgd 01h 1 11m	Middle digit is small [L & R are large]
011	bcd 10h 1 11m	Left digit is small [M & R are large]
111	00d 11h 1 11m	All digits are large; two bits are unused

Table 1.6: Expansion: (pqr)(stu)(v)(wxy) to (abcd)(efgh)(ijkm)

vwkst	abcd efgh ijk m
0....	0pqr 0stu 0wxy
100..	0pqr 0stu 100y
101..	0pqr 100u 0sty
110..	100r 0stu 0pqy
11100	100r 100u 0pqy
11101	100r 0pqu 100y
11110	0pqr 100u 100y
11111	100r 100u 100y

1.3.2 Combination Field

In IEEE 754-2008 standard there are 7 significant digits for the 32-bit format, 16 digits for the 64-bit format and 34 digits for the 128-bit format. In all three cases there is one digit more than a number divisible by 3. The most significant digit (MSD) of the significand is intermingled with two bits of the binary written exponent into five bits of 'Combination Field'. Also the non-numerical values Infinity and Not-a-Number (NaN) are encoded into this field. Table 1.7 shows the format of coding 5 bits of the combination field. The exponents bits a, b are the most significant exponent bits and can take the values 00, 01 and 10. The rest of the exponent bits are represented as 'w' bits of the combination field. The exponent is stored as a binary number. Since the first two bits of the exponent are not allowed to be 1 simultaneously its greatest allowed integer value is $(2^n - 1) - 2^{(n-2)} = 3 \times 2^{(n-2)} - 1$ where n is the number of bits in the exponent.

Table 1.7: Encoding format for 5 bits of Combination field

Type of value of the number	Combination field	Exponent bits	Significand's MSD
Finite	a b c d e	a b	0 c d e
Finite	1 1 a b e	a b	1 0 0 e
Infinite	1 1 1 1 0
NaN	1 1 1 1 1

Further details of IEEE 754-2008 standard are listed in Appendix.

1.4 Overview of Research

This thesis focuses on the design and synthesis of efficient decimal MAC (Multiply Accumulate) architecture for high performance decimal processors based on IEEE Standard for Floating-point Arithmetic (IEEE 754-2008). The design makes use of Binary Coded Decimal (BCD) coding for decimal representation. The decimal MAC unit has a DFP multiplier fused with a DFP adder module.

The DFP multiplier has to be an efficient, high speed multiplier, for the MAC unit to achieve high performance. The DFP multiplier can be designed using iterative and parallel approaches. This thesis presents two novel techniques for iterative DFP multiplication. The first approach has a Decimal Fixed Point (DFxP) multiplier using a novel Double Digit Decimal Multiplication (DDDM) technique that performs two digit multiplications simultaneously. The second approach does DFxP multiplication using a novel RPS algorithm. In this approach partial products for column accumulation are generated from the least significant end in an iterative manner using BCD digit multipliers. A novel design for BCD digit multiplication that reduces the critical path delay and area is also presented in this thesis. The thesis also presents a parallel DFP multiplier having a parallel DFxP multiplier for significant digit multiplication. Parallel designs are adopted when latency and throughput are considered more important than area. All these floating point multipliers presented incorporate exponent processing, rounding and exception detection capabilities. Different designs of DFP adders are also implemented aiming at reducing the delay.

In recent years, reversible logic has emerged as one of the most important approaches for power optimization with its significance in quantum computing and nanotechnology. There is considerable difference in the synthesis of logic circuits using classical logic gates and reversible gates. Design of reversible circuits for decimal arithmetic provides a low power approach for MAC implementation.

An improved design for a reversible logic implementation of BCD adder that is highly optimized in terms of number of reversible gates and garbage count (GC) is

presented. This research also presents a reversible fault tolerant implementation for quick addition of decimals (QAD) suitable for multi-digit BCD addition. The design makes use of reversible conservative Fredkin gates (FRG) only and presents a performance analysis of carry select and hybrid decimal adders. Toffoli Gate (TG) implementations of conventional BCD adders, adders for QAD, and carry select BCD adders for multi-digit addition are also presented. This thesis presents two new universal 4×4 'reversible RPS gates' that can function as a reversible 4-bit Binary to BCD converter with a garbage count of zero. The reversible implementations of BCD adder using fully reversible RPS gates, using combination of HNG-RPS gates and using HNG gates are presented as well. The reversible circuits presented forms the initial step in the building of complex reversible systems, which can execute more complicated operations for a reversible Decimal ALU.

Low power circuits designed in reversible logic for Hamming code generation and error detection circuit is also presented. Reversible logic is suitable for implementing error correcting code generation and detection circuits. The design is done using a new 4×4 reversible gate, Hamming Code Gate (HCG). A parity preserving HCG (PPHCG) that preserves the input parity at the output bits is used for achieving fault tolerance for the Hamming code generation and error detection circuits.

This thesis also presents an approach to obtain reduced hardware and/or delay for logic functions using multiplexers. Replication of single control line multiplexer is used as the only design unit for defining any logic function specified by minterms. A new algorithm is presented that does exhaustive branching to reduce the number of levels and/or modules required to implement any logic function. The algorithm identifies a single variable or a function at the control input of the multiplexer which leads to an implementation with reduced number of levels and/or hardware. A VLSI design using same modular building blocks can decrease system design and manufacturing cost for MAC implementation.

1.5 Layout of the Thesis

The layout of the thesis is as follows:

- Chapter 2 describes the different iterative and parallel decimal fixed point multipliers. It includes the double digit fixed point decimal multiplication for iterative approach. The use of a novel RPS algorithm for the fixed point decimal multiplication is explained in the next session. The chapter also presents a BCD digit multiplier design with reduced critical path delay and area. The design can be used in iterative and parallel decimal fixed point multipliers. Different parallel DFxP multipliers are also introduced. All these fixed point multipliers are used to multiply the significands of the floating point inputs in a DFP multiplier.
- Chapter 3 presents the different schemes of decimal floating point multiplication, decimal floating point adders and floating point fused multiply-add units.
- Chapter 4 presents reversible logic implementation of improved decimal adders suitable for low power designs.
- Chapter 5 presents new exhaustive branching algorithm to obtain reduced hardware and/or delay for logic functions using multiplexers.
- Chapter 6 presents the simulation results of various designs described in Chapters 2, 3, 4 and 5. The performances of the new systems are compared with that of existing systems and the results are tabulated.

- Chapter 7 sums up the thesis by drawing conclusions from the results, and suggests possible extensions of the research work for further investigation. References are listed following this chapter along with the details of publications by the author.

Chapter 2

Decimal Fixed Point Multiplication

Decimal floating point multiplication is an integral part of financial, and commercial computations. The multiplication of significant digits of floating point numbers are done by Decimal Fixed Point multipliers. Decimal Fixed Point multipliers are typically implemented using an iterative approach because of their complexity. This chapter presents a novel Double Digit Decimal Multiplication (DDDM) technique for iterative fixed point multiplication that performs 2-digit multiplications simultaneously. A novel RPS design for iterative decimal fixed point multiplication is also presented in this chapter. In this design the partial products are generated using BCD digit multipliers, and are accumulated based on a novel RPS algorithm. A novel design for BCD digit multiplication that reduces the critical path delay and area is presented as well in this chapter. The design is extended to a Hex/Decimal multiplier that gives either a decimal output or a binary output depending on the requirement. Parallel multipliers are used at the expense of area to attain high speeds. This chapter also presents three approaches for parallel fixed point decimal multiplication.

2.1 Decimal Fixed Point Multiplication

A Decimal Fixed Point (DFxP) multiplier multiplies an n -digit multiplicand, A , by an n -digit multiplier, B , producing a $2n$ -digit product, P . The block diagram of a DFxP multiplier is shown in Figure 2.1. The two main components in the DFxP multiplier design are: generation of partial products and accumulation of partial products. The generation of partial products can be done by three different ways: digit by digit multiplication, word by digit multiplication or word by word multiplication. In digit by digit multiplication, each digit of the multiplicand is multiplied by each digit of the multiplier by a decimal digit multiplier. In word by digit multiplication, the multiplicand as such is multiplied by each digit of the multiplier. In word by word multiplication the multiplicand is multiplied by the multiplier as a whole. The accumulation of partial products can be done in two ways: Row accumulation or Column accumulation.

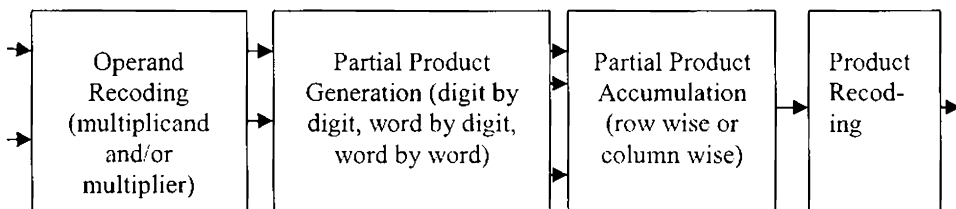


Figure 2.1: Block Schematic of Decimal Fixed Point Multiplication

2.2 Iterative DFxP Multipliers

DFxP multipliers are typically implemented using an iterative approach because of their complexity. Several iterative designs for DFxP multiplication have been proposed by [Larson, 1973], [Hoffman and

Schardt, 1975] and [Bradley *et al.*, 1986] in seventies and eighties. Such designs either successively add the current multiplicand as many times as the value of multiplier digit, or add a multiple of the multiplicand to the previously accumulated partial product. In an iterative DFxP multiplier presented in [Ohtsuki *et al.*, 1987], decimal partial products are generated by creating two partial products for each multiplier digit. Multiplying two n -digit Binary Coded Decimal (BCD) numbers requires n iterations, where all iterations consist of two binary carry-save additions and three decimal corrections. After n iterations, the carry and sum are added using a decimal carry propagate adder to produce the final product. The multiplier presented in [Ueda, 1995] generates the partial products by a costly retrieval of product of BCD digits from look-up tables (LUT). Several existing designs for DFxP multiplication generate and store multiples of the multiplicand before partial product generation. The multiplier digits are then used to select the appropriate multiple as the partial product [Busaba *et al.*, 2001]. The primary multiples $2A$ through $9A$ are calculated initially, and stored along with A to reduce the delay. The enormous area and delay required for generating all the eight multiples is a disadvantage of this approach. Eight additional registers are also needed to store these multiples.

An alternative method is to find a reduced set of secondary multiples. For example, if $2A$, $5A$, and $8A$ are computed and stored along with A , all the other multiples can be obtained with a single addition of two members from the set. Another reduced set of multiples, comprising A , $2A$, $4A$, and $8A$ has a one-to-one correspondence with the weighted bits of a BCD digit. The disadvantage is that certain missing multiples can be generated only by the addition of three multiples from the reduced set. For example, the generation

of $7A$ requires the addition of three multiples: A , $2A$, and $4A$. Although the secondary multiple approach reduces the area and register count, it introduces the overhead of potentially one more addition for each iteration. The multiplier design proposed by [Erle and Schulte, 2003] uses decimal carry-save addition to reduce this overhead. It gives a DFXP multiplication algorithm suitable for high-performance with short cycle times. The multiplier in [Kenney, Schulte and Erle, 2004] stores intermediate product digits in a less restrictive, redundant format called the overloaded decimal representation that reduces the delay of the iterative portion of the DFXP multiplier. The partial product generated is then added to an intermediate product register that holds the previously accumulated partial product.

This chapter presents two new iterative designs for DFXP multiplication to improve latency and speed. The throughput of these designs is less than one multiply per cycle because of the iterative method. The iterative nature implies a high degree of hardware re-use and these designs are most applicable to systems in which area is more important than throughput. The first approach presents a novel Double Digit Decimal Multiplication (DDDM) technique for iterative DFXP multiplication that performs 2-digit multiplications simultaneously. A novel RPS design for iterative DFXP multiplication is presented as the second approach. In this design the partial products are generated using Binary Coded Decimal (BCD) digit multipliers. (In this research the decimal digits are coded in BCD. BCD encoding of decimal digits is the simplest and most popular code for decimal data.) A new design for BCD digit multiplication that reduces the critical path delay and area is presented. The design is then extended to a Hex/Decimal multiplier that gives either a decimal output or a binary output depending on the requirement.

2.2.1 Double Digit Decimal Multiplication (DDDM)

The DDDM method for DFXP multiplier iterates over the digits of the multiplier operand and successively produces partial products by a word by digit multiplication. The method uses a reduced set of secondary multiples comprising of $2A$, $4A$ and $5A$ for partial product generation. The novel approach is that the multiplier generates multiplicand multiples for 2-digits of the multiplier simultaneously in each iteration cycle. The DDDM method uses decimal carry save addition, to reduce the iterative delay, for row accumulation of partial products as in [Erle and Schulte, 2003]. This DFXP multiplication algorithm is suitable for high-performance with less number of cycles and short cycle times.

The DDDM technique is explained with the help of a block diagram shown in Figure 2.2. The main blocks of the multiplier are the ‘Secondary Multiple Generation Block’, ‘Multiplier Shift Register’, ‘Multiplexer Block’, ‘Carry save Adder Block’ (comprising of Decimal Carry Save adder and Decimal 4:2 compressor), ‘Temporary Product Registers’, ‘Partial Product Shift Register’, and the ‘Decimal Carry Propagate Adder’.

2.2.1.1 Secondary Multiple Generation Block

The ‘Secondary Multiple Generation’ block generates the reduced set of secondary multiples of the multiplicand A . For an n -digit multiplication it generates $2A$, $4A$ and $5A$ multiples of length $(n+1)$ digits. The $2A$, $4A$, and $5A$ secondary multiple set is selected since they can be generated with least delay compared to other secondary multiple sets. This is because, doubling ($2A$) and quintupling ($5A$) of BCD numbers do not require carry propagation beyond the next digit.

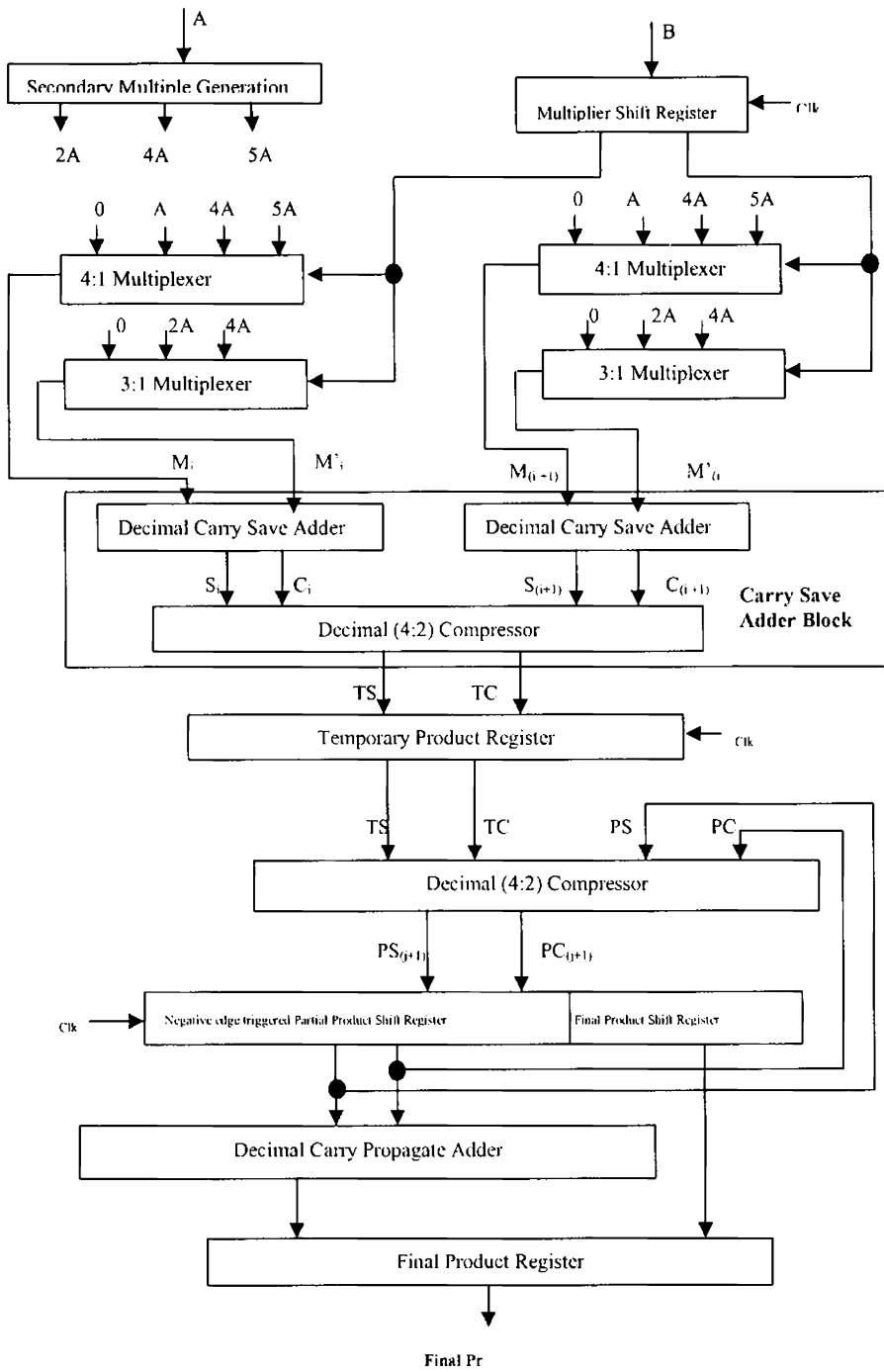


Figure 2.2: Block Diagram of the Fixed Point Double Digit Decimal Multiplication

When any BCD digit is doubled, its Least Significant Bit (LSB) initially becomes zero. Thus, if a carry-out of one occurs (for digit values in the range 5-9), it does not propagate beyond the next digit's, LSB which is always zero. The 'Secondary Multiple Generation' block is purely a combinational block with a maximum delay of 6 gates [Erle and Schulte, 2003].

Let a_i ($0 \leq i < n$) be the n digits of the multiplicand A ,

a_{2i} ($0 \leq i \leq n$) be the $n+1$ digits of $2A$ and

a_{5i} ($0 \leq i < n$) be the $n+1$ digits of $5A$.

The equations for generating a_{2i} of $2A$ are as follows (bit 0 is the LSB):

$$a_{2i}(0) = (a_{i-1}(2) \cdot a_{i-1}(1) \cdot \overline{a_{i-1}(0)}) + (a_{i-1}(2) \cdot a_{i-1}(0)) + a_{i-1}(3) \quad (2.1)$$

$$a_{2i}(1) = (\overline{a_i(3)} \cdot \overline{a_i(2)} \cdot a_i(0)) + (a_i(2) \cdot a_i(1) \cdot \overline{a_i(0)}) + (a_i(3) \cdot \overline{a_i(0)}) \quad (2.2)$$

$$a_{2i}(2) = (a_i(1) \cdot a_i(0)) + (\overline{a_i(2)} \cdot a_i(1)) + (a_i(3) \cdot \overline{a_i(0)}) \quad (2.3)$$

$$a_{2i}(3) = (a_i(2) \cdot \overline{a_i(1)} \cdot \overline{a_i(0)}) + (a_i(3) \cdot a_i(0)) \quad (2.4)$$

where $a_{2i}(j)$ indicates the j^{th} bit of the i^{th} digit of $2A$

$a_{i-1}(j)$ indicates the j^{th} bit of the previous digit of a_i

Quintupling a digit whose value is odd will give a 2-digit result with lower significant digit as five ('0101'). But, when digit is even, the 2-digit result will have the lower significant digit as zero. Further, any digit whose value is ≥ 2 will produce a carry-out in the range 1-4. Thus, when a carry-out does occur, it will not propagate beyond the next significant digit, which has a maximum value of five before adding the carry. The equations for generating a_{5i} of $5A$ are as follows (bit 0 is the LSB):

$$a_{5i}(0) = (a_i(0) \cdot \overline{a_{i-1}(3)} \cdot \overline{a_{i-1}(1)}) + (\overline{a_i(0)} \cdot a_{i-1}(1)) + (a_i(0) \cdot a_{i-1}(3)) \quad (2.5)$$

$$a_{5i}(1) = (\overline{a_i(0)} \cdot a_{i-1}(2)) + (a_i(0) \cdot \overline{a_{i-1}(2)} \cdot a_{i-1}(1)) + (a_{i-1}(2) \cdot \overline{a_{i-1}(1)}) \quad (2.6)$$

$$a_{5i}(2) = (a_i(2) \cdot \overline{a_{i-1}(3)} \cdot \overline{a_{i-1}(1)}) + (a_i(0) \cdot \overline{a_{i-1}(2)} \cdot a_{i-1}(1)) + (\overline{a_i(0)} \cdot a_{i-1}(3)) \quad (2.7)$$

$$a_{5i}(3) = (a_i(0) \cdot a_{i-1}(2) \cdot a_{i-1}(1)) + (a_i(0) \cdot a_{i-1}(3)) \quad (2.8)$$

where $a_{5i}[j]$ indicates the j^{th} bit of the i^{th} digit of $5A$

$a_{i-1}[j]$ indicates the j^{th} bit of the previous digit of a_i

All the secondary multiples are available after 6 gate delays as seen in Figure 2.3 ($2A$ can be generated from A in three gate delays, $4A$ can be generated from $2A$ in three more gate delays, and in parallel $5A$ can be generated from A). All the multiples are generated using combinational logic, and also the value of A does not change throughout the iterations [Erle and Schulte, 2003].

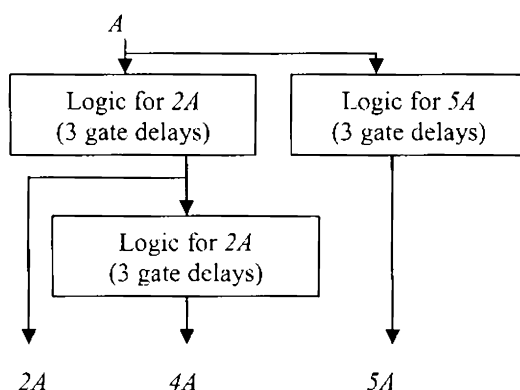


Figure 2.3: Secondary Multiple Generation

2.2.1.2 Multiplier Shift Register

The multiplier input, B is loaded into the ‘Multiplier Shift Register’ using an asynchronous load input. Let b_i ($0 \leq i < n$) be the n digits of the multiplier B . For each clock cycle, 2 digits of the multiplier B (b_i and b_{i+1}) are shifted out by the ‘Multiplier Shift Register’ block.

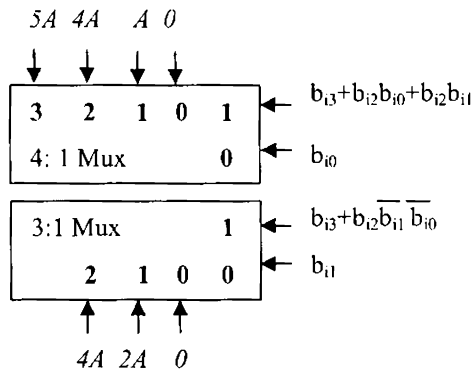
2.2.1.3 Multiplexer Block

Suitable secondary multiples are selected using a pair of multiplexers for each digit of the multiplier (b_i) according to Table 2.1 [Erle and Schulte, 2003].

Table 2.1: Recoding of Digits of b_i

b_i	M_i	M'_i	b_i	M_i	M'_i
0	0	0	5	5A	0
1	A	0	6	4A	2A
2	0	2A	7	5A	2A
3	A	2A	8	4A	4A
4	0	4A	9	5A	4A

A 4:1 Multiplexer selects one out of the 4 possible values from (0, A, 4A, 5A) at M_i , and a 3:1 Multiplexer does the selection of one of the 3 possible values from (0, 2A, 4A) at M'_i . For example if the multiplier digit (b_i) is 9 then 5A and 4A are selected by the multiplexer pair. The control logic at the control inputs of the multiplexers are shown in Figure 2.4. The same process is repeated for the second digit of the multiplier (b_{i+1}) using another multiplexer pair.

**Figure 2.4:** Multiplexer Block

2.2.1.4 Carry Save Adder Block

The secondary multiples selected by the multiplexer block are to be added to get the missing multiples of the multiplicand A using a BCD adder. For example, the $5A$ and $4A$ secondary multiples selected by the multiplexer pair (when the multiplier digit (b_i) is 9) are to be added to get the multiple $9A$. This is done by a decimal carry save adder that use direct decimal addition [Schmookler and Weinberger, 1971]. A direct decimal adder accepts two 4-bit BCD digits, x_i and y_i , along with a 1-bit carry-in, $c_i(0)$. It directly produces a 4-bit BCD sum digit, s_i , and a 1-bit carry-out, $c_{i+1}(0)$, such that

$$(c_{i+1}(0), s_i) = x_i + y_i + c_i(0) \quad (2.9)$$

The equations for performing the direct decimal addition of two BCD digits are given below, where bit 0 is the LSB of the 4-bit BCD digit.

$$g_i(j) = x_i(j) \cdot y_i(j) \quad 0 \leq j \leq 3 \quad (2.10)$$

$$p_i(j) = x_i(j) + y_i(j) \quad 0 \leq j \leq 3 \quad (2.11)$$

$$h_i(j) = x_i(j) \oplus y_i(j) \quad 0 \leq j \leq 3 \quad (2.12)$$

$$k_i = g_i(3) + (p_i(3) \cdot p_i(2)) + (p_i(3) \cdot p_i(1)) + (g_i(2) \cdot p_i(1)) \quad (2.13)$$

$$l_i = p_i(3) + g_i(2) + (p_i(2) \cdot g_i(1)) \quad (2.14)$$

$$c_i(1) = g_i(0) + (p_i(0) \cdot c_i(0)) \quad (2.15)$$

$$s_i(0) = h_i(0) \oplus c_i(0) \quad (2.16)$$

$$s_i(1) = ((h_i(1) \oplus k_i) \cdot c_i(1)) + ((h_i(1) \oplus l_i) \cdot c_i(1)) \quad (2.17)$$

$$s_i(2) = (p_i(2) \cdot g_i(1)) + (p_i(3) \cdot h_i(2) \cdot p_i(1)) + ((g_i(3) + (h_i(2) \cdot h_i(1)))) \cdot c_i(1) + ((p_i(3) \cdot p_i(2) \cdot p_i(1)) + (g_i(2) \cdot g_i(1)) + (p_i(3) \cdot p_i(2))) \cdot c_i(1) \quad (2.18)$$

$$s_i(3) = ((k_i \cdot l_i) \cdot c_i(1)) + (((g_i(3) \cdot h_i(3)) + (h_i(3) \cdot h_i(2) \cdot h_i(1))) \cdot c_i(1)) \quad (2.19)$$

$$c_{i+1}(0) = k_i + (l_i \cdot c_i(1)) \quad (2.20)$$

A direct decimal adder forms the basic functional block of decimal carry save addition, and is referred as a decimal (3:2) counter [Erle and Schulte, 2003]. Two decimal carry save adders (each of (n+1) digits) are used to add the secondary multiples selected by the multiplier pairs. Each decimal carry save adder gives an (n+1)-digit sum and (n+1)-bit carry. These two sums and the carries are added by a decimal (4:2) compressor. A decimal (4:2) compressor's basic functional block accepts two 4-bit BCD digits, x_i and y_i , and two 1-bit carry-ins, $c_i(0)$ and $c'_i(0)$ as inputs [Erle and Schulte, 2003]. It produces a 4-bit BCD sum digit, s_i , and a 1-bit carry-out $c_{i+1}(0)$. The decimal (4:2) compressor uses a standard decimal (3:2) counter to compute

$$(c'_{i+1}(0), s'_i) = x_i + y_i + c_i(0) \quad (2.21)$$

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:

$$(c_{i+1}(0), s_i) = s'_i + c'_i(0) + c'_{i+1}(0) \quad (2.22)$$

In the Figure 2.2 the 'Carry save Adder Block' comprises of two 'Decimal Carry Save Adder' blocks and a 'Decimal 4:2 compressor'. Decimal carry save adder adds the two secondary multiples of (n+1)-bits selected by the multiplexer block using decimal 3:2 compressor. It generates an (n+1)-digit sum output (S_i) and an (n+1)-bit carry output (C_i). Similar addition is done by the second decimal carry save adder to produce $S_{(i+1)}$ and $C_{(i+1)}$. These four outputs are now added by a decimal 4:2 compressor to give temporary sum (TS) and temporary carry (TC) values.

2.2.1.5 Partial Product Register

The TS and TC values are stored in the ‘Temporary Product Registers’. The stored values are added with the shifted output of the previous partial product (PS_i and PC_i) in the ‘Partial Product Register’ using a decimal 4:2 compressor to get a new partial product. The last two digits of the partial product formed is a part of the final product. The new partial product is stored in the ‘Partial Product Shift Register’ at the negative edge of the clock in shifted form. For this purpose, the data in the ‘Final Product Shift Register’ is shifted for two digits during the previous positive edge, giving room to store the new two digits of the final product during the negative clock edge.

For each iteration cycle, the multiplicand is multiplied by two digits of the multiplier. The partial product formed is shifted by two digits, and the process is repeated for $\lceil (n/2) \rceil$ iterations. After $\lceil (n/2) \rceil$ iterations the final product in the form of ‘carry save’ sum and carry is available at the output of the ‘Partial Product Shift Register’. This is then passed to a ‘Decimal Carry Propagate Adder’.

2.2.1.6 Decimal Carry Propagate Adder

‘Decimal Carry Propagate Adder’ is a Decimal Incrementer, and is shown in Figure 2.5. It is a circuit that adds a single bit (C_i) to a decimal digit (X_i) along with the carry in $C_{o(i-1)}$, and gives the result in BCD with a carry out (C_{oi}).

$$C_{oi} = X_{i(3)}C_iC_{o(i-1)} + X_{i(3)}X_{i(0)}C_i + X_{i(0)}C_{o(i-1)} \quad (2.23)$$

The C_{0i} is generated after two gate delays for each digit the maximum complexity of the gate being a 5-input AND gate. For an n-digit multiplication, the ripple delay for C_{0i} at the final Decimal Propagate Adder is $2n$ gate delays. The Boolean expressions for a single digit Decimal Incrementer are given in equations 2.24-2.27.

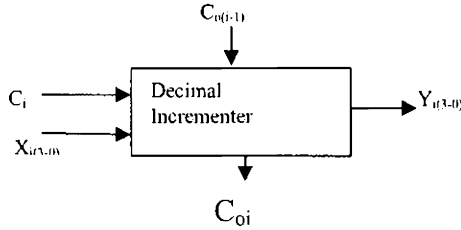


Figure 2.5: Decimal Incrementer

$$Y_{i(3)} = \overline{X_{i(3)} C_i C_{0(i-1)}} + (X_{i(2)} X_{i(1)} X_{i(0)} + X_{i(3)} \overline{X_{i(0)}})(C_i \oplus C_{0(i-1)}) + \overline{X_{i(3)} X_{i(2)} X_{i(1)}} C_i C_{0(i-1)} \quad (2.24)$$

$$Y_{i(2)} = \overline{X_{i(2)} C_i C_{0(i-1)}} + (X_{i(1)} X_{i(0)} + \overline{X_{i(3)} X_{i(2)}})(C_i \oplus C_{0(i-1)}) + (\overline{X_{i(3)} X_{i(2)} X_{i(1)}} + \overline{X_{i(2)} X_{i(1)}}) C_i C_{0(i-1)} \quad (2.25)$$

$$Y_{i(1)} = \overline{X_{i(3)} C_i C_{0(i-1)}} + (X_{i(1)} \overline{X_{i(0)}} + \overline{X_{i(3)} X_{i(1)} X_{i(0)}})(C_i \oplus C_{0(i-1)}) + \overline{X_{i(3)} X_{i(1)}} C_i C_{0(i-1)} \quad (2.26)$$

$$Y_{i(0)} = X_{i(0)} \oplus C_i \oplus C_{0(i-1)} \quad (2.27)$$

Total delay of the ‘Decimal Carry Propagate Adder’ is the delay of one digit Decimal Incrementer and $2n$ gate delays. This is much less than the delay of an n-digit BCD ripple adder. The adder output is then stored in the ‘Final Product Register’. The final product is available after $\lceil (n/2) + 1 \rceil$ clock cycles.

DFxP multiplication of significand digits is an integral component of floating point multiplication. Decimal floating point has 3 representations: 32-bit format with 7 significand digits, 64-bit format with 16 significand

digits and 128-bit format with 34 significant digits. DDDM for 7-digit, 16-digit and 34-digit are simulated using Leonardo Spectrum from Mentor Graphics Corporation using ASIC Library 0.18micron, 1.8V CMOS technology. The designs are then compared with single digit decimal multipliers (SDDM) in terms of area and delay. Since the floating point multiplier may need to handle operands up to 34 decimal digits improvement in latency is an important factor to be considered. This is achieved by a double digit decimal multiplication technique.

Further improvement in speed is achieved for a design using partitioned blocks. 34-digit multipliers are implemented as a combination of 17-digit multipliers also. The block diagram for 34-digit implementation partitioned into 17-digit multipliers is given in Figure 2.6. The design for 34-digit implementation partitioned into 17-digit multipliers is simulated using Leonardo Spectrum from Mentor Graphics Corporation using ASIC Library and compared with DDDM in terms of area and delay.

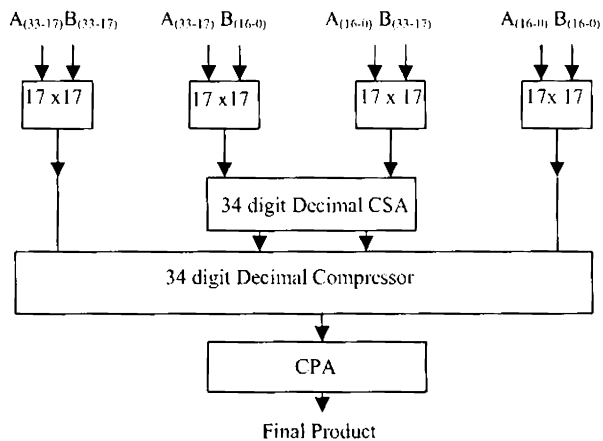


Figure 2.6: Block Diagram for the Partitioned DDDM for 34-digits

2.2.2 Decimal Fixed Point Multiplication using RPS Algorithm

The second approach for iterative DFxP multiplication uses a novel RPS algorithm. This design differs from the DDDM technique in the mode of partial product generation and accumulation. In this approach partial products for column accumulation are generated from the least significant end in an iterative manner using BCD digit multipliers. The BCD digit multipliers perform the digit by digit multiplication on the digits of the multiplicand and multiplier. The partial products are then accumulated in a column manner using multi-operand decimal adders.

2.2.2.1 BCD Digit Multiplier

A general conventional paper and pencil view of digit by digit BCD multiplication is given in Figure 2.7. Each BCD digit product, $A_i \times B_j$ is represented by two BCD digits, P_{ijH} and P_{ijL} such that the weight of P_{ijH} is ten times as much as P_{ijL} .

$$\begin{array}{r}
 \begin{array}{r}
 A_2 A_1 A_0 \times \\
 B_2 B_1 B_0
 \end{array} \\
 \hline
 \begin{array}{r}
 P_{20L} P_{10L} P_{00L} \\
 P_{20H} P_{10H} P_{00H} \\
 P_{21L} P_{11L} P_{01L} \\
 P_{21H} P_{11H} P_{01H} \\
 P_{22L} P_{12L} P_{02L} \\
 P_{22H} P_{12H} P_{02H}
 \end{array} \\
 \hline
 \begin{array}{r}
 P_5 \quad P_4 \quad P_3 \quad P_2 \quad P_1 \quad P_0 \\
 C_5 \quad C_4 \quad C_3 \quad C_2 \quad C_1
 \end{array} \\
 \hline
 \begin{array}{r}
 FP_6 \quad FP_5 \quad FP_4 \quad FP_3 \quad FP_2 \quad FP_1 \quad FP_0
 \end{array}
 \end{array}$$

Figure 2.7: Conventional Paper and Pencil View of Digit by Digit Multiplication

BCD digit multiplier is the basic building block of a digit by digit decimal multiplication process. A BCD digit multiplier accepts two BCD inputs (A, B) that can take values from (0-9). It realizes a function $P=F_{(A, B)}$, giving a product in the range (0, 81) represented by two BCD digits. The function may be realized, using an 8-input, 8-output combinational logic, or using a 256×8 look-up table in a straightforward manner. But, the practical constraints on area and latency call for more optimum designs. An alternate method is to use a standard 4×4 unsigned binary multiplier generating an 8-bit binary output that needs to be corrected to two BCD digits.

A novel design for BCD digit decimal multiplication that reduces the critical path delay and area is presented. It has been observed that there are one hundred possible combinations of BCD inputs for multiplication, out of which only 4 combinations require 4×4 multiplication, 64 combinations need 3×3 multiplication, and the remaining 32 combinations use either 3×4 or 4×3 multiplication. This design makes use of this property. The BCD digit multiplier consists of two parts: a binary multiplier that gives a binary product $p_{(7-0)}$, and a binary to BCD converter. Since the multiplier accepts only BCD inputs, the maximum value of the 4-bit input is 9 (1001_2). This restricts the binary product bits to $p_{(6-0)}$.

2.2.2.2 Binary Multiplier

The binary multiplier consists of a 3×3 multiplier, a 4×3 multiplier, and a 4×4 multiplier. Figures 2.8, 2.9 and 2.10 show the 3×3 , 4×3 and 4×4 multiplications for BCD inputs respectively. In 4×3 multiplication for BCD inputs, one of the inputs is either 8(1000_2) or 9(1001_2). So, the 4×3 multiplier gets simplified to three 2-input AND gates. In 4×4 multiplication for BCD

inputs, both inputs are either $8(1000_2)$ or $9(1001_2)$. So the 4×4 multiplier gets simplified to a half adder (a 2-input AND gate and a 2-input XOR gate) as seen in Figure 2.10.

$$\begin{array}{r}
 \begin{array}{r}
 x_2 \quad x_1 \quad x_0 \quad \times \\
 y_2 \quad y_1 \quad y_0 \\
 \hline
 \end{array} \\
 \begin{array}{r}
 x_2 y_0 \quad x_1 y_0 \quad x_0 y_0 \\
 x_2 y_1 \quad x_1 y_1 \quad x_0 y_1 \\
 x_2 y_2 \quad x_1 y_2 \quad x_0 y_2 \\
 \hline
 \end{array} \\
 \begin{array}{r}
 p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0
 \end{array}
 \end{array}$$

Figure 2.8: 3×3 Multiplication

$$\begin{array}{r}
 \begin{array}{r}
 1 \quad 0 \quad 0 \quad x_0 \quad \times \\
 y_2 \quad y_1 \quad y_0 \\
 \hline
 \end{array} \\
 \begin{array}{r}
 y_0 \quad 0 \quad 0 \quad x_0 y_0 \\
 y_1 \quad 0 \quad 0 \quad x_0 y_1 \\
 y_2 \quad 0 \quad 0 \quad x_0 y_2 \\
 \hline
 \end{array} \\
 \begin{array}{r}
 y_2 \quad y_1 \quad y_0 \quad x_0 y_2 \quad x_0 y_1 \quad x_0 y_0
 \end{array}
 \end{array}$$

Figure 2.9: 4×3 Multiplication of BCD inputs

$$\begin{array}{r}
 \begin{array}{r}
 1 \quad 0 \quad 0 \quad x_0 \quad \times \\
 1 \quad 0 \quad 0 \quad y_0 \\
 \hline
 \end{array} \\
 \begin{array}{r}
 y_0 \quad 0 \quad 0 \quad x_0 y_0 \\
 1 \quad 0 \quad 0 \quad x_0 \\
 \hline
 \end{array} \\
 \begin{array}{r}
 1 \quad 0 \quad x_0 y_0 \quad x_0 \oplus y_0 \quad 0 \quad 0 \quad x_0 y_0
 \end{array}
 \end{array}$$

Figure 2.10: 4×4 Multiplication of BCD inputs

The 3×3 and 4×3 multipliers give a 6-bit binary product, while a 4×4 multiplier produces a 7-bit binary result. The binary to BCD converter, following the binary multiplier, should be a 7-input converter. The design can be further simplified if conversion needs to be done only for 6-bit products. The 6-bit converter converts the binary output of the 3×3 multiplier or 3×4 multiplier outputs to its corresponding BCD. Instead of using a 7-bit binary to BCD converter, the 4×4 multiplier is designed to produce an 8-bit BCD output as shown in Figure 2.11. The 4×4 multiplier and the binary to BCD conversion circuit of its product now gets reduced to a 2-input AND, NAND, XOR and NOR gates.

$$\begin{array}{r}
 1 0 0 x_0 \times \\
 1 0 0 y_0 \\
 \hline
 x_0 y_0 \quad (x_0 y_0)' \quad (x_0 y_0)' \quad x_0 \oplus y_0 \quad 0 \quad (x_0 + y_0)' \quad x_0 \oplus y_0 \quad x_0 y_0
 \end{array}$$

Figure 2.11: 4×4 Multiplication of BCD inputs generating 8-bit BCD output

2.2.2.3 6-bit Binary to BCD Converter

Binary product can be converted to an equivalent BCD by a 6-input, 8-output combinational logic. Although the general binary-to-BCD conversion is extensively addressed in the literature [Schmookler 1968], [Rhyne, 1970], [Arazi and Naccache, 1992] a special, simpler and faster, binary-to-BCD converter depicted in [Jaberipur and Kaivani, 2007] for a 6-bit input is used for this research. The first row in Figure 2.12 shows the BCD weights. The weights of p_3 , p_2 , p_1 and p_0 are the same as the corresponding weights in the original binary number $p_{(5-0)}$. But, weights 16 and 32 of p_4 and p_5 have been decomposed to (10, 4, 2) and (20, 10, 2) respectively. The four BCD digits in

the right four columns (lower BCD digits) are added using a single decimal adder to get the sum ($D_3D_2D_1D_0$) and the carry. The first ($0 p_2 p_1 p_0$) and last ($0 0 p_5 0$) two lower BCD digits are added using a binary adder. Binary adder is enough since the result will never exceed 9. This binary adder can be implemented using two XOR gates and two AND gates as shown in Figure 2.13.

<i>80</i>	<i>40</i>	<i>20</i>	<i>10</i>	<i>8</i>	<i>4</i>	<i>2</i>	<i>1</i>
0	0	p_5	p_4	0	p_2	p_1	p_0
0	0	0	0	p_3	0	0	0
0	0	0	p_5	0	p_4	p_4	0
0	0	0	0	0	0	p_5	0
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

Figure 2.12: The principle of 6-bit binary to BCD conversion

0	p_2	p_1	$p_0 +$
0	0	p_5	0
$p_1p_2p_5$	$p_2 \oplus p_1p_5$	$p_1 \oplus p_5$	p_0

Figure 2.13: Addition of first and last lower BCD digits

The other two lower BCD digits, ($p_3 0 0 0$) and ($0 p_4 p_4 0$), are added to get a BCD sum and a carry-out using three AND gates and two NOT gates as shown in Figure 2.14. Hence the 6-bit converter gets reduced to a single digit BCD adder and a 2-bit Adder as shown in Figure 2.15. The BCD adder finds the sum of results of Figure 2.13 and Figure 2.14. The carry-out p_3p_4 in Figure 2.14 is added to the two higher BCD digits (“ $0 0 p_5 p_4$ ” and “ $0 0 0 p_5$ ”) along

with the carry from the BCD adder using a 2-bit adder to get $(D_7D_6D_5D_4)$. If the product were a 7-bit number then product term will have a p_6 bit with weight 64 that has to be decomposed into (40, 20, 4). This increases the depth of BCD addition required by one more level in both lower and higher digit levels. So, the depth of addition is reduced by making use of a 6-bit converter.

$$\begin{array}{rcccc}
 p_3 & 0 & 0 & 0 & + \\
 0 & p_4 & p_4 & 0 & \\
 \hline
 p_3 p_4 & p_3 \overline{p_4} & p_4 & \overline{p_3} p_4 & 0
 \end{array}$$

Figure 2.14: Addition of other two lower BCD digits to get a BCD Sum

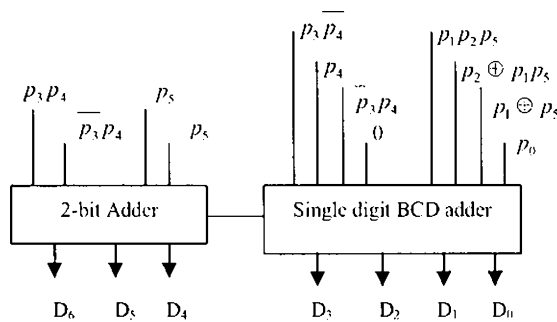


Figure 2.15: 6-bit Binary to BCD converter

The block diagram of the proposed BCD digit multiplier is shown in Figure 2.16. The 4-bit 2:1 multiplexer selects the inputs to 4×3 multiplier depending on x_3 bit. The 6-bit 2:1 multiplexer does the selection between the 3×3 multiplier output and the 4×3 multiplier output depending on the status of x_3 and y_3 bits. If x_3 and y_3 are different then the output of 4×3 multiplier is passed to the BCD converter, else the output of 3×3 multiplier is passed. After the 6-bit binary to BCD conversion the third multiplexer (8-bit 2:1

vector Mux) selects the BCD converter output or the 4×4 multiplier output (which gives an 8-bit BCD result) depending on x_3 and y_3 bits. If both are '1' then the 4×4 multiplier output is selected, else the BCD converter output is passed as the final product.

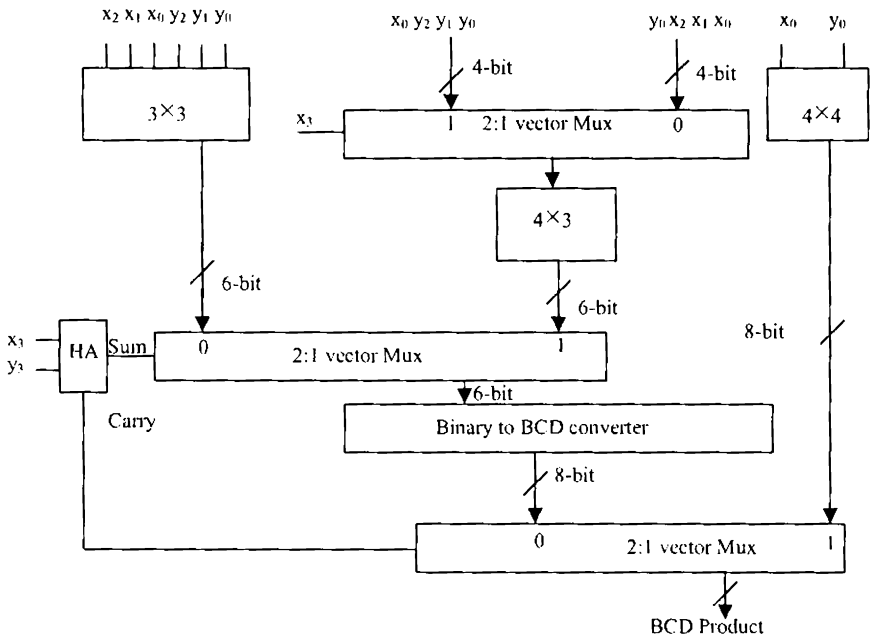


Figure 2.16: BCD Digit Multiplier

2.2.2.4 Hex/Decimal Multiplier

The single digit decimal multiplier design is extended to a Hex/Decimal multiplier that gives either a decimal output or a binary output depending on the requirement. Digital Signal Processing (DSP) applications require binary multipliers while financial and commercial applications require decimal multipliers. A Hex multiplier accepts two 4-bit binary inputs and gives an 8-bit product. The single digit decimal multiplier has a 3×3 multiplier block. This can be extended to realize a 4×4 multiplier with some

extra hardware. The terms marked as bold in Figure 2.17 indicates the additional AND products that are required for a 4×4 multiplication compared to a 3×3 multiplication.

				x_3	x_2	x_1	$x_0 \times$
				y_3	y_2	y_1	y_0

				x_3y_0	x_2y_0	x_1y_0	x_0y_0
			x_3y_1	x_2y_1	x_1y_1	x_0y_1	
		x_3y_2	x_2y_2	x_1y_2	x_0y_2		
	x_3y_3	x_2y_3	x_1y_3	x_0y_3			

h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

Figure 2.17: 4×4 Hex Multiplication

Hence to realize a 4×4 multiplication only those terms which are marked in bold need be added to the result of a 3×3 multiplication. The design for the single digit BCD multiplier is modified as shown in Figure 2.18 to make it a Hex/Decimal multiplier. The additional hardware required is seven AND gates, and an adder that adds three 4-bit numbers. The adder can be realized by five full adders and one half adder. The AND array works in parallel with the 3×3 multiplier, and the adder works in parallel with the rest of the BCD multiplier circuit. Hence, no additional delay is added up due to the additional hardware. Finally, a 2:1 vector multiplexer selects one of the two products (Hex/Decimal) based on a control input.

A comparison of the Hex/Decimal multiplier design with one designed using the multiplier in [Jaberipur and Kaivani, 2007] in terms of area and critical path delay is done with the logic synthesis tool Leonardo Spectrum from Mentor Graphics Corporation using ASIC Library 0.18 micron, 1.8 V CMOS technology.

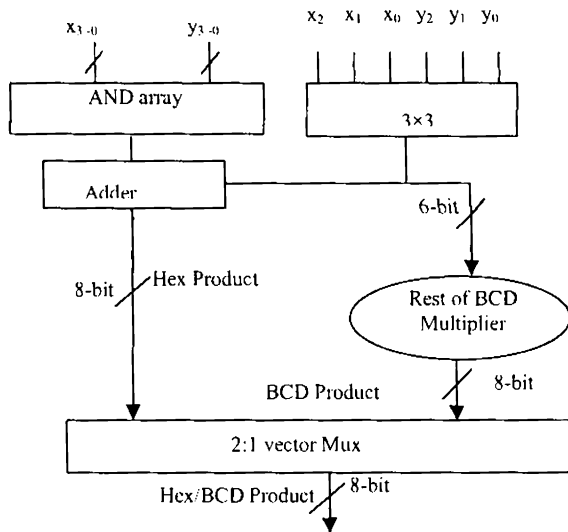


Figure 2.18: Hex/Decimal multiplier

2.2.2.5 Multi-operand Decimal Adders

The partial products of a DFxP multiplication generated by the BCD digit multipliers are then accumulated in a column manner using multi-operand decimal adders. Efficient multi-operand decimal addition is essential for the implementation of fast decimal multipliers. Many techniques have been developed to speed up the process of decimal addition for multi-operand decimal adders. Direct decimal addition is one of the efficient techniques for two-operand decimal addition [Schmookler and Weinberger, 1971]. A variant of direct decimal addition to produce intermediate results in a decimal carry-save format for an iterative decimal multiplier is proposed in [Erle and Schulte, 2003]. In another approach, a correction value of six is added to each digit of the first partial product using a binary carry save adder [Ohtsuki et al., 1987]. A technique for constant time decimal addition, called Redundant

Binary Coded Decimal (RBCD) was proposed in [Shirazi, Yun and Zhang, 1988], [Shirazi, Yun and Zhang, 1989].

Multi-operand decimal addition is also performed by adding each operand repeatedly using two-operand decimal addition. This approach, however, is very slow since each addition has carry propagation. A faster approach introduced by [Kenney and Schulte, 2005], use binary carry-save addition to compute intermediate results. When several operands are added together, the intermediate results are kept in binary carry-save format, to delay the carry-propagate addition until the end. Three algorithms were introduced by [Kenney and Schulte, 2005] for performing fast decimal addition on multiple BCD operands: non-speculative tree, double correction speculation array, and single correction speculation array. Two of the techniques speculate BCD correction values, and correct intermediate results for the addition of input operands. The first technique, Single Correction Speculation, speculates over one addition. The second technique, Double Correction Speculation, speculates over two additions. The third technique, Non-speculative Addition, uses a binary carry-save adder tree and produces a binary sum. Combinational logic is then used to correct the sum, and determine the carry into the next significant digit. The non-speculative tree algorithm that gives the minimum delay with same area among the three algorithms is best suited for multi-operand decimal addition. This technique is used in this research.

Non-speculative Addition:

Non-speculative adders add BCD input operands in a binary carry-save tree, passing carries generated along the way to the next significant digit. Preliminary binary sums and carry-outs from the carry-save adder tree are fed

into the combinational logic, which produce a decimal sum and carry corrections, if needed. These values are determined based on the carry out of the current digit position and the preliminary sum digit.

In the non-speculative addition, efficient combinational logic can be designed to produce sum and carry correction for a given number of input operands. A 1-digit, m-operand Non-speculative Adder requires (m-2) 4-bit carry-save adders, one 4-bit carry-propagate adder, one 5-level combinational logic block to generate the carry-out and correction digits (for up to 16 input operands), and one 3-bit carry-propagate adder to add the correction digit to the binary sum. In this research, non-speculative multi-operand adders are used for column accumulation of partial products generated by the BCD digit multipliers. The partial products that are to be generated and accumulated are selected using RPS algorithm.

2.2.2.6 RPS algorithm

A DFxP multiplier unit using RPS algorithm accepts two n-digit operands, calculates n^2 partial products, and returns their sum as a 2n-digit integer. This design uses n BCD digit multipliers for partial product generation for an $n \times n$ digit DFxP multiplication. The RPS algorithm selects appropriate inputs for n BCD digit multipliers for generation of partial products in each cycle. The inputs are selected in such a way that the partial products can be accumulated column wise from the least significant end in an iterative manner. Column accumulation is done using multi-operand decimal adders. The process repeated n times, generates a 2n-digit product after the $(n+1)^{\text{th}}$ cycle. The steps for RPS algorithm to multiply two n-digit numbers are as follows.

For partial product generation:

Step 1: Initialize i, j, k, m, c to 0. (i - to select A_i, j - to select B_j)

Step 2: i=k, j=m

Step 3: c = c + 1, Select A_i and B_j as inputs to the cth BCD digit multiplier

Find P_{ij} = A_i × B_j

Step 4: If c = n then set c = 0 and wait until next clock

Step 5: If i>0 then i = i-1, j = j+1 and go to step 3

Step 6: k=k+1, if k<n then go to step 2

Step 7: k = n-1, m = 1

Step 8: i=k, j=m

Step 9: c = c + 1, Select A_i and B_j as inputs to the cth BCD digit multiplier

Find P_{ij} = A_i × B_j

Step 10: If c = n then set c = 0 and wait until next clock

Step 11: If j<n-1 then i = i-1, j = j+1 and go to step 9

Step 12: m = m+1, if m<n-1 then go to step 8

For Partial product accumulation:

Step 1 : Multi-operand BCD addition:

$$\text{Adder}_x = (\sum P_{yzL} + \sum P_{abH}) + C_{\text{Ladder}(x-1)} + C_{\text{Hadder}(x-2)}$$

If $x < n$, then $y = x$ to 0, $z = 0$ to x , $a = (x-1)$ to 0, $b = 0$ to $(x-1)$

If $x = n$, then

$y = (n-1)$ to $x-(n-1)$, $z = x-(n-1)$ to $(n-1)$, $a = (x-1)$ to 0, $b = 0$ to $(x-1)$

If $x > n$, then

$y = (n-1)$ to $x-(n-1)$, $z = x-(n-1)$ to $(n-1)$, $a = (n-1)$ to $(x-n)$,

$b = (x-n)$ to $(n-1)$

$C_{\text{Ladder}(x-1)}$ is the lower carry digit from the previous addition

$C_{\text{Hadder}(x-2)}$ is the higher carry digit from the addition prior to previous addition

The algorithm is explained for a (7-digit \times 7-digit) DFxP multiplication as shown in Figure 2.19. This example is considered since it is an integral component of a 32-bit decimal floating point multiplication that has 7 significand digits. For a 7-digit \times 7-digit multiplication the partial products are generated in 7 cycles and the final product (14-digit) after the 8th cycle. A (7-digit \times 7-digit) multiplication results in 7×7 (49) partial products, each having 2 digits given by P_{ijL} and P_{ijH} . This design makes use of only seven single digit BCD multipliers. Usually, the partial product of the complete multiplicand is generated with a single digit of the multiplier in each cycle. But this design generates those partial products that are necessary for early accumulation using the RPS algorithm. The multiplication using RPS algorithm is explained with the help of Figure 2.19. The first iteration generates seven partial products as seen from the right most end of the partial product array of Figure 2.19. Hence after the first cycle, the partial products generated are P_{00} , P_{10} , P_{01} , P_{20} , P_{11} , P_{02} , P_{30} (both P_L and P_H), as shown in red in the array. Similarly during the second cycle, seven more partial products are generated. The partial products that are generated in first cycle are added simultaneously using multi-operand decimal adders to generate the final products FP_0 , FP_1 and FP_2 .

For a 7-digit \times 7-digit multiplication, the maximum number of BCD operands to be added in a column is 15. The block schematic of a one-digit 15-operand BCD adder that adds 15 one digit operands is shown in Figure 2.20. Non-speculative adder for 15 operands makes use of three one digit 5-operand BCD adders, two 3-operand BCD adders and a 2-operand BCD adder. The design of a one-digit 5-operand BCD adder is shown in Figure 2.21.

$A_6 A_5 A_4 A_3 A_2 A_1 A_0 \times$
 $B_6 B_5 B_4 B_3 B_2 B_1 B_0$

$P_{60L} P_{50L} P_{40L} P_{30L} P_{20L} P_{10L} P_{00L}$
 $P_{60H} P_{50H} P_{40H} P_{30H} P_{20H} P_{10H} P_{00H}$
 $P_{61L} P_{51L} P_{41L} P_{31L} P_{21L} P_{11L} P_{01L}$
 $P_{61H} P_{51H} P_{41H} P_{31H} P_{21H} P_{11H} P_{01H}$
 $P_{62L} P_{52L} P_{42L} P_{32L} P_{22L} P_{12L} P_{02L}$
 $P_{62H} P_{52H} P_{42H} P_{32H} P_{22H} P_{12H} P_{02H}$
 $P_{63L} P_{53L} P_{43L} P_{33L} P_{23L} P_{13L} P_{03L}$
 $P_{63H} P_{53H} P_{43H} P_{33H} P_{23H} P_{13H} P_{03H}$
 $P_{64L} P_{54L} P_{44L} P_{34L} P_{24L} P_{14L} P_{04L}$
 $P_{64H} P_{54H} P_{44H} P_{34H} P_{24H} P_{14H} P_{04H}$
 $P_{65L} P_{55L} P_{45L} P_{35L} P_{25L} P_{15L} P_{05L}$
 $P_{65H} P_{55H} P_{45H} P_{35H} P_{25H} P_{15H} P_{05H}$
 $P_{66L} P_{56L} P_{46L} P_{36L} P_{26L} P_{16L} P_{06L}$
 $P_{66H} P_{56H} P_{46H} P_{36H} P_{26H} P_{16H} P_{06H}$

$P_{13} P_{12} P_{11} P_{10} P_9 P_8 P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0$
 $C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1$
 $C_{7H} C_{6H} C_{5H}$

$FP_{13} FP_{12} FP_{11} FP_{10} FP_9 FP_8 FP_7 FP_6 FP_5 FP_4 FP_3 FP_2 FP_1 FP_0$

Figure 2.19: Partial product generation and accumulation in different cycles

The intermediate result, z' , and the carry from each carry-save adder (CSA) determine the sum correction, g , and the carry correction, c_{out} values. Table 2.2 gives g and c_{out} values for a 5-operand adder, where z' is the intermediate sum. The design of a one-digit 3-operand BCD adder has only one CSA and a Carry Propagate Adder (CPA). The combinational logic for 'sum and carry correction logic' of a 3-operand BCD adder is a simplified version of that of 5-operand BCD adder. The 2-operand adder is a normal BCD adder with 6-correction logic when sum exceeds 9. The remaining BCD adders for column accumulation of partial products require multi-operand adders of size 14, 13, 11, 10, 8, 7, 5 and 3. The designs for multi-operand BCD adders for 14, 13 and 11 inputs are the same as that of 15-operand adder. For 10, 8 and 7-operand adders, only two 5-operand adders, and three 2-operand adders are required.

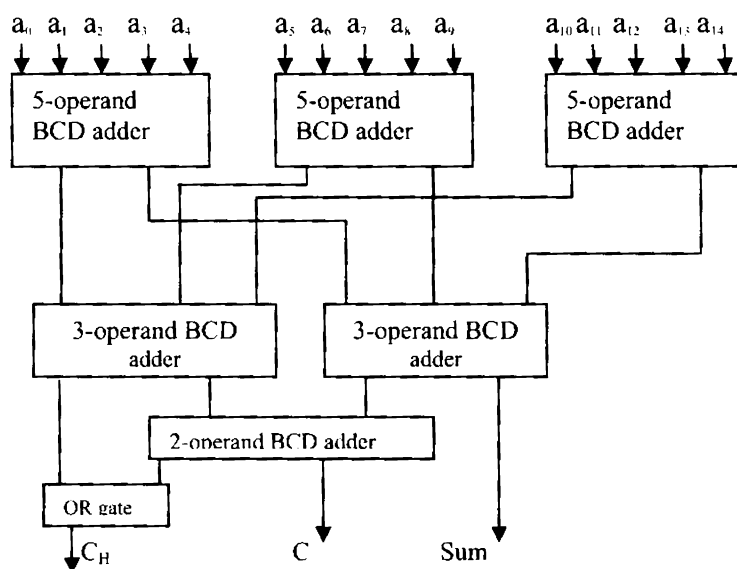
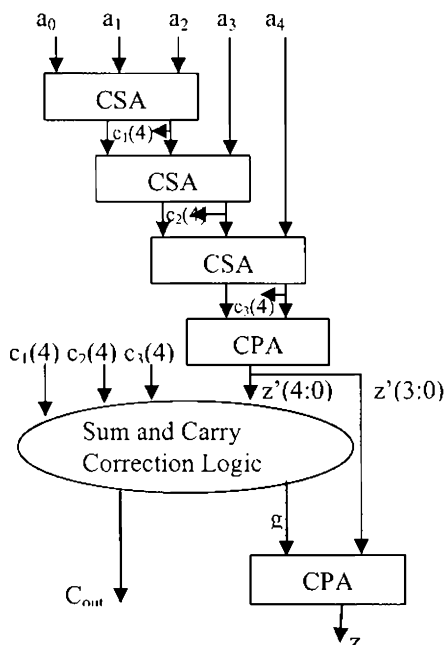


Figure 2.20: One-digit, 15-operand Non-speculative Adder

Table 2.2: Sum and carry correction values for 5-operand BCD adder

$c_1(4)+c_2(4)+c_3(4)$	$z'(4:0)$	c_{out}	g
0	$0 \leq z'(4:0) < 10$	0	0
0	$10 \leq z'(4:0) < 20$	1	6
0	$20 \leq z'(4:0) < 30$	2	12
0	$30 \leq z'(4:0) < 32$	3	2
1	$0 \leq z'(4:0) < 4$	0	6
1	$4 \leq z'(4:0) < 14$	1	12
1	$14 \leq z'(4:0) < 24$	2	2
1	$24 \leq z'(4:0) < 32$	3	8
2	$0 \leq z'(4:0) < 8$	1	2
2	$8 \leq z'(4:0) < 16$	2	8

For a 7-digit \times 7-digit multiplication the partial product accumulation using such multi-operand BCD adders starts accumulation from the second cycle onwards, and is completed in the eighth cycle. The final product is available after the eighth cycle.

**Figure 2.21:** One-digit, 5-operand Non-speculative Adder

The block diagram of an n-digit DFxP multiplier using RPS algorithm is shown in Figure 2.22.

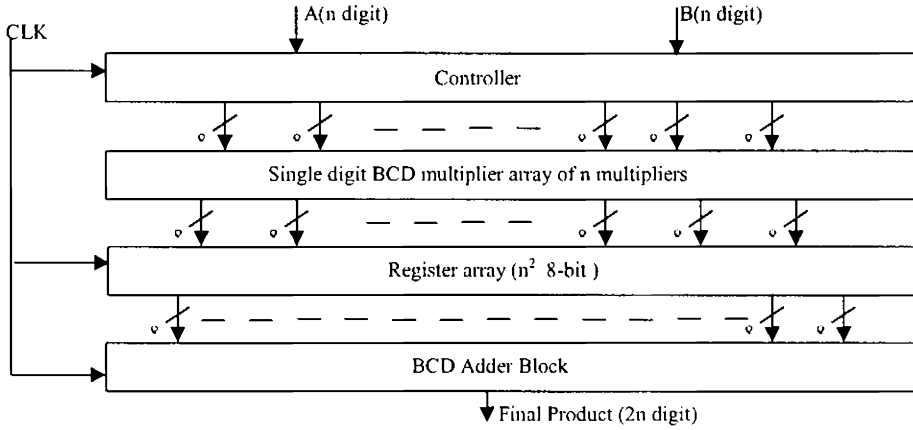


Figure 2.22: DFxP multiplier using RPS algorithm

The controller block uses RPS algorithm to determine the flow of inputs to n BCD digit multipliers for each cycle. The detailed design of the controller block is shown in Figure 2.23.

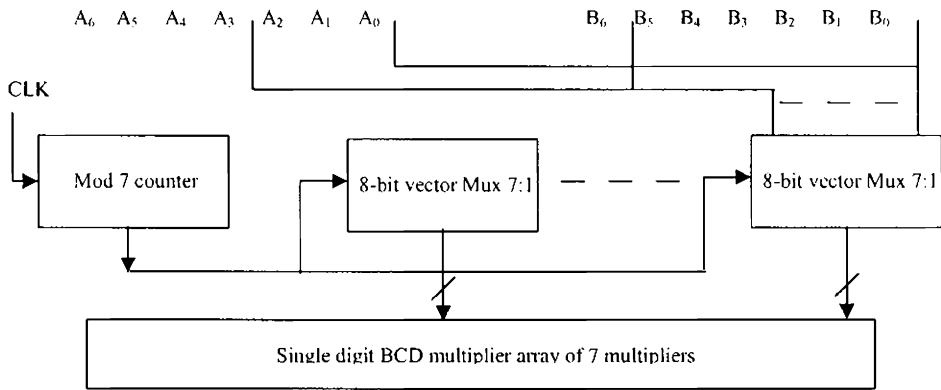


Figure 2.23: Decimal Fixed Point multiplier controller block

The controller block is followed by an array of n BCD digit multipliers that give the set of partial products. These are stored in n registers of an $n \times n$ 8-bit register array as shown in Figure 2.24. During the first cycle, data stored in registers are marked as red. The inputs to the multi-digit BCD adders are selected using the RPS algorithm from this stored data as shown in Figure 2.25.



Figure 2.24: Register array for storing output of BCD-digit multiplication

The complete multi-operand BCD adder array is shown in Figure 2.26. During the eighth cycle FP_9 , FP_{10} , FP_{11} , FP_{12} , and FP_{13} are generated. This is done using 11-operand (9-partial products and two carries (C_8 and C_{7H}) that are generated in previous cycles), 7-operand, 5-operand, 3-operand decimal adders, and a 4-digit high speed decimal adder. The 4-digit high speed decimal adder is used to add all the corresponding carries generated. The maximum depth of addition occurs at the eighth cycle, and this determines the maximum operating frequency. Simulations are done using Leonardo Spectrum from Mentor Graphics Corporation using ASIC Library 0.18micron, 1.8V CMOS technology. When multiplying two n -digit operands to produce a $2n$ -digit

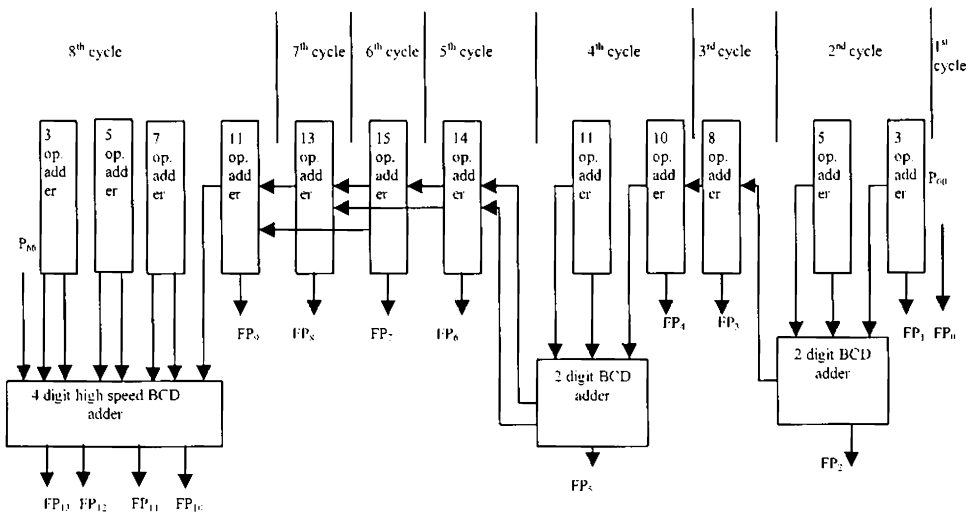


Figure 2.26: BCD adder array

In design using RPS algorithm $2n$ -digit result of an n -digit \times n -digit DFxP multiplication is available after $(n+1)$ clock cycles. For example for the 7-digit multiplication the final 14-digit result is available after 8 clock cycles. If the result has to be rounded to 7-digits then rounding process should be done after the multiplication process. Usually, for iterative multipliers the rounding process can begin only after completing the DFxP multiplication process. But, since the design using RPS algorithm generates the digits from the least significant end of the final product in each cycle, the rounding process can be initiated during the DFxP multiplication process itself. Hence the rounding process works in parallel with the DFxP multiplication process. This speeds up the floating point multiplication that in turn reduces delay. This iterative approach is suitable for high speed applications.

2.3 Parallel DFXP Multipliers

This chapter also presents three new approaches for parallel DFXP multiplier. Parallel designs are adopted when latency and throughput are considered more important than area. These multipliers attain high speeds at the expense of area. The first implementation of a parallel decimal multiplier was presented by [Lang and Nannarelli, 2006]. Two architectures for parallel decimal multipliers based on decimal carry save multi-operand addition that used a BCD-4221 recoding for decimal digits were introduced by [Vazquez, Antelo and Montuschi, 2007]. Instead of using a tree of carry save adders the partial product accumulation is done using the multi-operand decimal addition by [Dadda, Nannarelli and Milano, 2008]. Dadda obtained the sum of each column of the partial product array in binary form and then converted it to decimal. This scheme gives slight reduction in delay keeping the area almost constant compared to the design by [Lang and Nannarelli, 2006]. In all the parallel decimal multiplier designs published so far, generation of partial products was done by some recoding scheme of decimal digits and then generating multiples of multiplicand. An alternative approach is to generate the partial product using BCD digit multipliers. The accumulation of partial products can be done in two ways: Row accumulation or Column accumulation. As the accumulation of partial products occurs in parallel, these designs are pipelined to allow a throughput of one.

2.3.1 Row Accumulation

Decimal Carry Save Adder (DCA) described in Section 2.2.1.4 is an integral building block of partial product accumulation. The partial products

generated by using BCD digit multipliers are added using a tree structure of DCAs as shown in Figure 2.27. Level 1 is a set of simplified DCA that adds two BCD digits to generate the ‘Intermediate Sum’ and ‘Carry’ outputs. The ‘Intermediate Sum’ and ‘Carry’ outputs are added using two different design schemes. ‘Design 1’ uses Decimal (4:2) Compressors for accumulation of ‘Intermediate Sum’ and ‘Carry’ outputs. A Decimal (4:2) Compressor is shown in Figure 2.27 as D4:2C. It accepts two 4-bit BCD digits, x_i and y_i , and two 1-bit carry-inputs, c_i and c_{i+1} as inputs, and produces a 4-bit BCD sum digit, s_i , and a 1-bit carry-out c_{i+1} as described in Section 2.2.1.4. The last level is also a DCA of $(2n-1)$ digits for an n -digit \times n -digit multiplication.

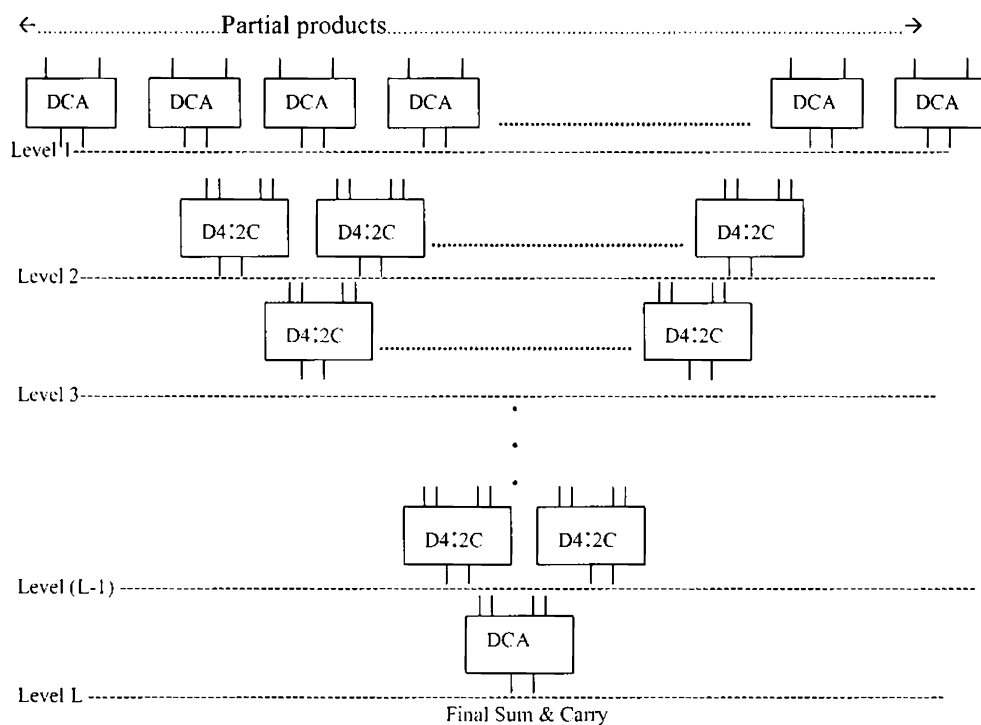


Figure 2.27: Partial Product accumulation using carry save adders

The Fixed Point Parallel Decimal Multiplier for row accumulation using ‘Design 1’ is verified for a (7-digit \times 7-digit) multiplication. All partial product digits are generated using the BCD digit multipliers. The partial product row accumulation for a (7-digit \times 7-digit) fixed point multiplier is shown in Figure 2.28. It consists of a decimal carry save adder block with six 7-digit decimal carry save adders and a Decimal 4:2 Compressor block. Decimal 4:2 Compressor block has three 7-digit Decimal (4:2) Compressors, two 9-digit Decimal (4:2) Compressors and one 13-digit DCA as shown in Figure 2.29. The Final Sum and Carry are then added using a ‘Decimal Carry Propagate Adder’, which is a Decimal Incrementer. The adder output is then stored in the ‘Final Product Register’.

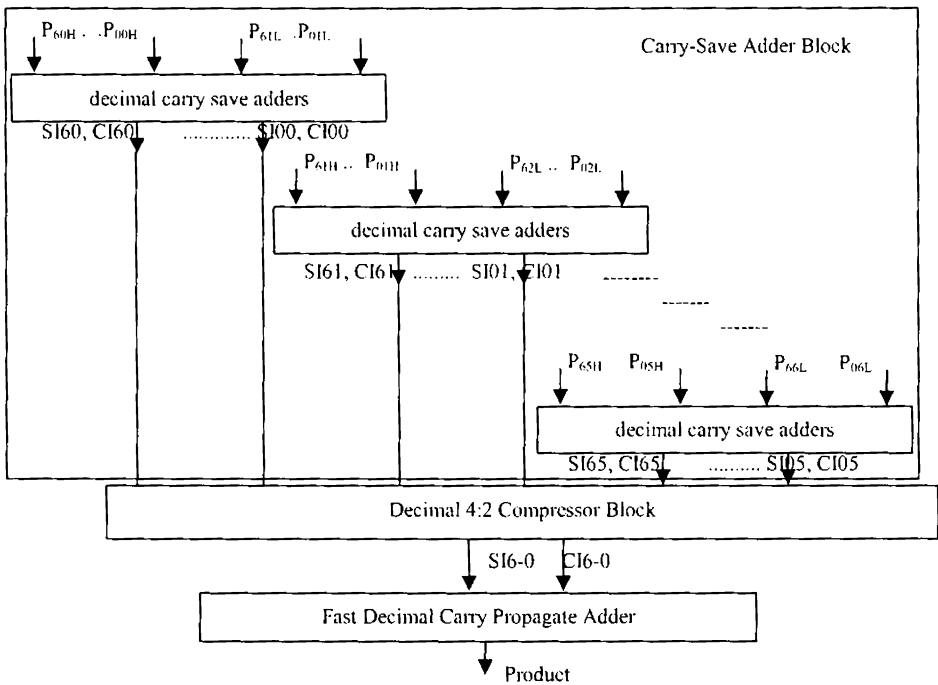


Figure 2.28: Partial Product accumulation of 7-digit \times 7-digit multiplier

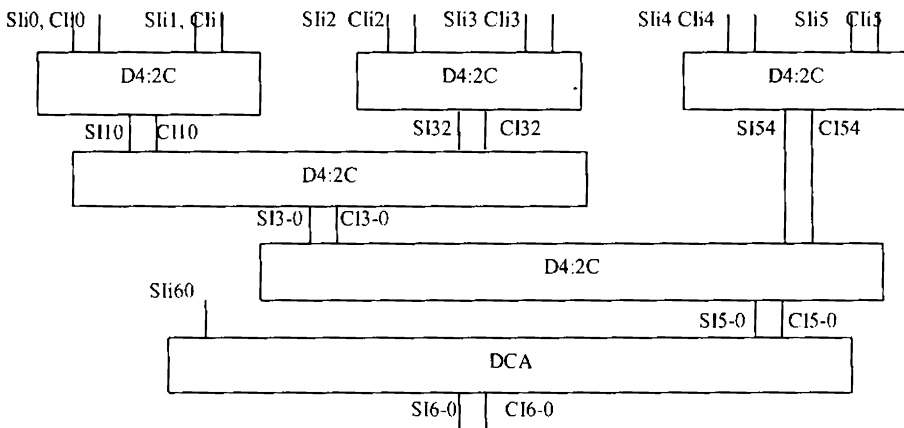


Figure 2.29: Decimal 4:2 Compressor Block for a 7-digit \times 7-digit multiplier

‘Design 2’ uses BCD digit multipliers for partial product generation while retaining the design by Lang and Nannarelli (2006) for partial product accumulation. In ‘Design 2’, partial product accumulation is done using Decimal carry save adders and Carry Counters in place of Decimal 4:2 Compressors. Carry Counter is a combinational circuit that generates BCD equivalent of number of ‘1’s at its input. The verification of a 7-digit multiplier using this modified design is done. The use of BCD digit multipliers for partial product generation makes it a more regular implementation with lesser delay. The simulation results show that the design using Carry Counters (Design 2) has reduced area and delay compared to the design using Decimal 4:2 Compressors (Design 1). The design is extended to 16-digit and 34-digit multipliers since they form integral components of 64-bit and 128-bit decimal floating point multipliers.

2.3.2 Column Accumulation

In column accumulation, the partial products in each column are added simultaneously to generate an 'Intermediate Sum Digit' and 'Intermediate Carry digit/s'. This is implemented using multi-operand decimal adders. The method uses a scheme in which the decimal digits in each column are added using Decimal Carry-save Adders (DCA), and the carry out for each column are accumulated using Carry Counters (CC). The realization of an N-operand decimal adder using tree structure of decimal carry save adders will have N decimal carry save adders at L levels where $2^{(L-1)} < N \leq 2^L$ and a carry counter for generation of 'carry out'. The complete tree structure for L=4 for the addition of 16 operands is shown in Figure 2.30.

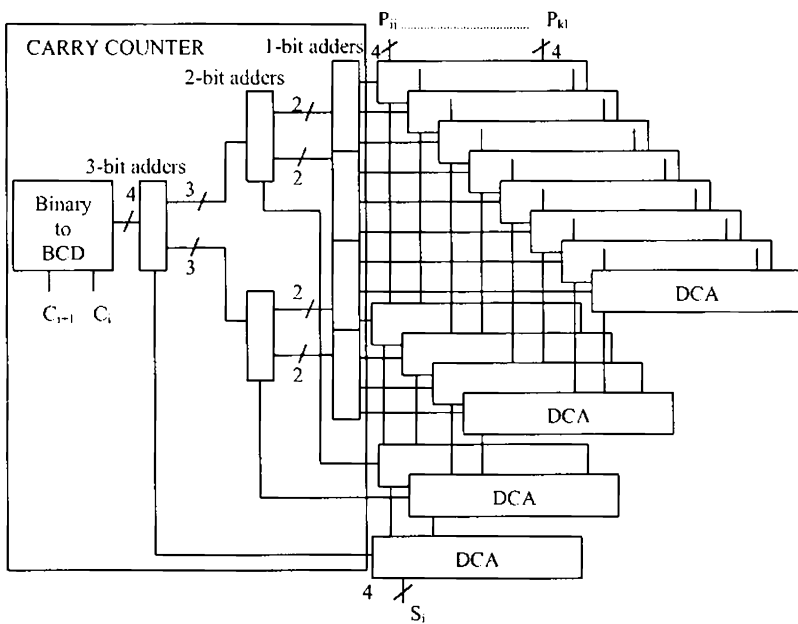


Figure 2.30: Column Adder array for 16-operand BCD addition

Partial product accumulation is implemented column wise for a (7-digit \times 7-digit) fixed point multiplier, and is shown in Figure 2.31. The multi-operand adder block for a (7-digit \times 7-digit) fixed point multiplier consists of two 3-operand, 5-operand, 7-operand, 9-operand, 11-operand, 13-operand decimal adders. For a (7-digit \times 7-digit) fixed point multiplier, the maximum delay to obtain the sum is that of 4 stages of carry save adders (for 13-operand addition) along with the additional delay for accumulating the carry digits. The final stage of partial product accumulation uses a fast decimal adder that generates the final products (FP_i). This fast decimal adder is a fixed block adder with a block size of four.

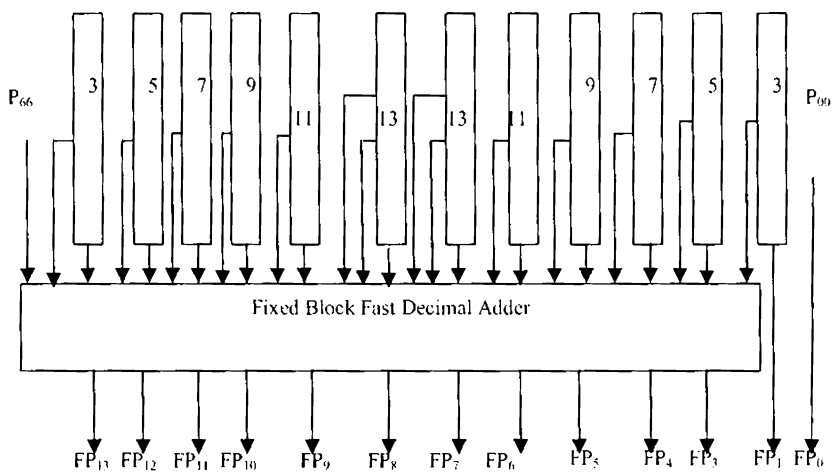


Figure 2.31: Adder array for accumulating partial products column wise

2.4 Summary

The DFxP multipliers are designed using iterative and parallel approaches. For iterative designs two design approaches are presented. The first approach makes use of a double digit decimal multiplication (DDDM) technique that performs two digit multiplications simultaneously. In one cycle the entire multiplicand is multiplied by two multiplier digits. The partial products are generated by selectively adding the secondary multiples depending on the multiplier digit. DDDM for 7-digit, 16-digit and 34-digit are simulated using Leonardo Spectrum from Mentor Graphics Corporation using ASIC Library 0.18micron, 1.8V CMOS technology. The designs are then compared with existing design in terms of area and delay. The area and delay comparison for implementations of 7-digits, 16-digits and 34-digits DDDM on different families of Xilinx, Altera, Actel and Quick logic FPGAs are also tabulated. FPGAs are increasingly being used for improving performance by scientific computing community to implement floating-point based hardware accelerators.

The second approach performs DFxP multiplication using a novel RPS algorithm. The partial products for column accumulation are generated using BCD digit multipliers from the least significant end in an iterative manner. The design for DFxP multiplier using RPS algorithm has n BCD digit multipliers for an $n \times n$ multiplication. This design leads to a more regular VLSI implementation, and does not require special registers for storing easy multiples. The latency for the multiplication of two n -digit BCD operands is $(n+1)$ cycles and a new multiplication can begin every n cycle. The design was validated using a 7-digit \times 7-digit DFxP multiplier that is required for a 32-bit DFP multiplication. Area and delay analysis is done using logic

synthesis tool Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library 0.18micron, 1.8V CMOS technology.

This chapter also presents parallel decimal multipliers that offer reduced area and delay compared to the existing implementations. BCD digit multipliers are used for partial product generation. The design is synthesised for (7-digit \times 7-digit) fixed point decimal multiplication. Partial product accumulation was done row wise using carry counters along with decimal carry save adders, and using decimal 4:2 compressors. The designs were then extended to 16-digit and 34-digit multipliers since they form integral components of 64-bit and 128-bit decimal floating point multipliers. The comparison between column and row accumulations shows that the column accumulation gives a reduction in delay with decrease in area. This parallel fixed point multiplier can be used for implementing parallel decimal floating point multipliers. The simulation results of different decimal fixed point multiplier designs are given in Section 6.2.

Chapter 3

Decimal Floating Point Multipliers and MAC Unit

Floating-point representation can support a much wider range of values over fixed point representation. In this chapter, iterative and parallel decimal floating-point multiplier designs in IEEE 754-2008 format are designed. The design also incorporates the necessary decimal floating-point exponent processing, rounding and exception detection capabilities. Floating point MAC unit implements the fused multiply-add operation with a single final rounding after add operation. The fused multiply-add unit uses parallel or iterative multipliers and a floating point adder unit. The DFP adder is implemented using ripple carry BCD adders, kogge-stone adders and reduced delay BCD adders.

3.1 Decimal Floating Point Multipliers

Many hardware designs for Decimal Floating Point (DFP) multiplication [L. Eisen et al., 2007], [E. M. Schwarz, J. S. Kapernick, and M. F. Cowlshaw, 2009], [Cohen et al., 1983] and [Bolenger *et al.*, 1987] are available in literature. Hardware design that follow IEEE 754–2008 standard for DFP multiplication are given in [M. A. Erle, M. J. Schulte, and B. J. Hickmann, 2007], [Hickmann *et al.*, 2007], [M. A. Erle, B. J. Hickmann and M. J. Schulte, 2009], [Charles Tsen *et al.*, 2009] and [H. A. H. Fahmy *et al.*, 2009]. The design in [M. A. Erle, M. J. Schulte, and B. J. Hickmann, 2007], uses decimal carry save adders of [M. A. Erle and M. J. Schulte, 2003] for DFxP multiplication of its significand digits. The design in [M. A. Erle, B. J. Hickmann and M. J. Schulte, 2009] uses partial product accumulation based on non-pipelined iterative technique using decimal carry save adders. A combined decimal and binary floating-point multiplier is presented in [Charles Tsen *et al.*, 2009]. Parallel DFP multipliers using parallel DFxP multipliers are presented in [Hickmann *et al.*, 2007] and [H. A. H. Fahmy *et al.*, 2009].

In this chapter two approaches for iterative decimal floating point multiplication complying with the IEEE 754–2008 standard are presented. The first approach has a DFxP multiplier using RPS algorithm. In this method, partial products for column accumulation are generated from the least significant end in an iterative manner. The second approach has a DFxP multiplier using Double Digit Decimal Multiplication (DDDM) technique that performs two digit multiplications simultaneously in one cycle. A parallel decimal floating point multiplier having a parallel DFxP multiplier for significand digit multiplication is also presented in this chapter.

3.2 DFP Multiplication using RPS Algorithm

DFP multiplier design of first approach extends the iterative DFxP multiplier design described in Chapter 2 and is shown in Figure 3.1.

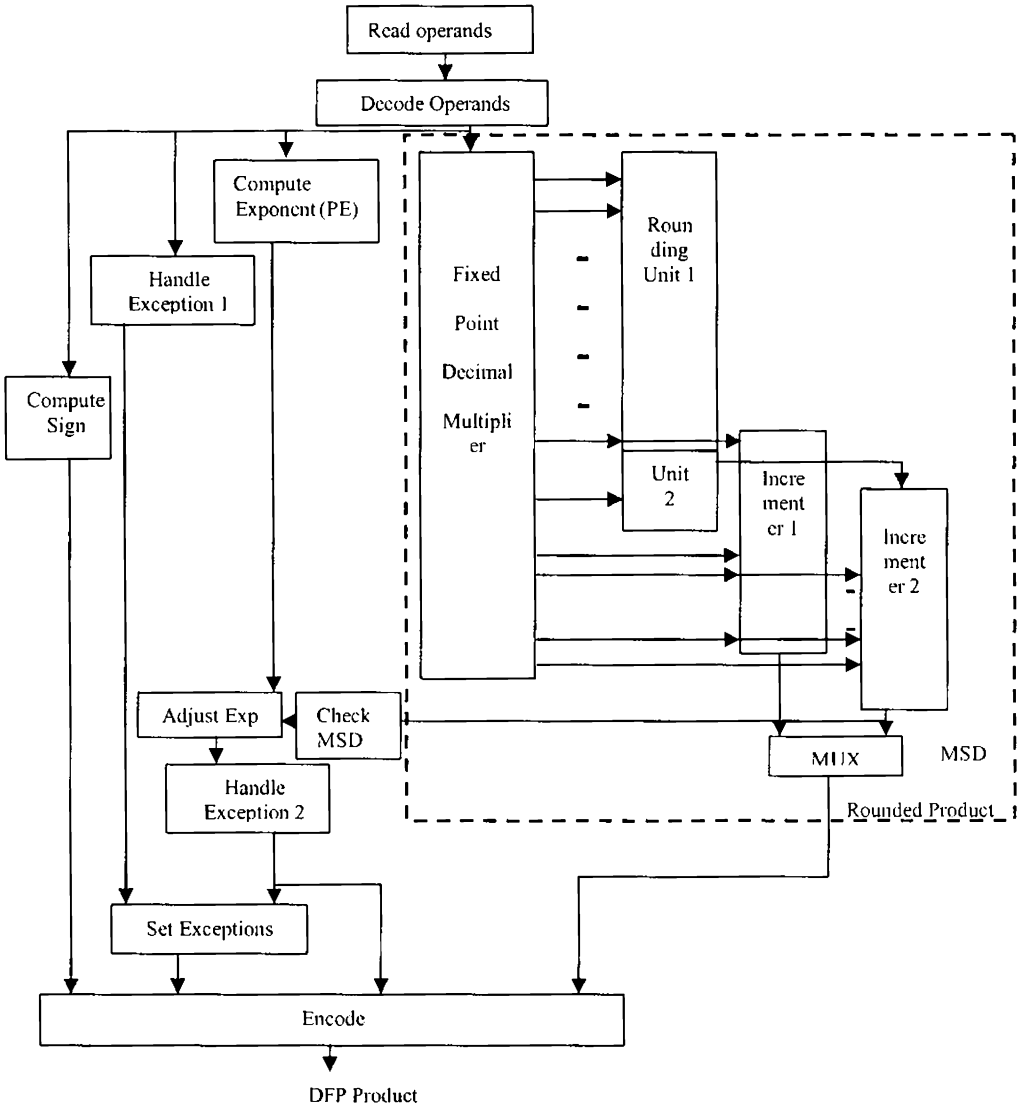


Figure 3.1: Decimal Floating Point (DFP) Multiplier

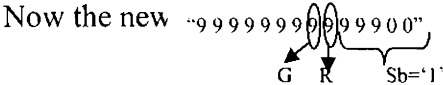
The operands encoded in DPD are decoded to get BCD significand digits and binary exponents. Then the product of significand digits of the two operands is generated using RPS algorithm for iterative DFXP multiplication. The maximum precision or the maximum length of the significand is denoted as 'P_{limit}', which is equal to 7, 16, and 34 digits, for decimal32, decimal64, and decimal128 formats respectively. Rounding is required when all the essential digits of the 2n-digit product of an n-digit × n-digit DFXP multiplication cannot be accommodated in 'P_{limit}' digits. This is accomplished by selecting either the product truncated to 'P_{limit}' or its incremented value based on Sticky bit (Sb), Round digit (R) and Guard Digit (G). The least significant (n-2) digits determine the sticky bit (Sb). It is desirable to generate sticky bit on-the-fly with DFXP multiplication to improve the speed of DFP. When the final product of the DFXP multiplication is one digit less than the total length, it may be necessary to shift left the product by one digit to make the Most Significant Digit (MSD) a non-zero number. The corrective left shift of one digit necessitates maintaining an additional digit to the right of the decimal point. This digit is referred to as the guard digit. The corrective left shift may lead to an overflow to the (n+1)th digit while rounding the result. This situation is demonstrated below using an example of 7-digit × 7-digit DFXP multiplication for a 32-bit DFP multiplier.

Let C1 be the significand of operand 1, and C2 the significand of operand 2.

If C1="3333330"
and C2="3000003"

Then the 14 digit result of $C1 \times C2 = "099999999999990"$

Since the MSD is zero a left shift is performed to avoid leading zero, and the exponent is adjusted accordingly.



Since $G > 5$, the first 7 digits are to be rounded up, and results in an 8 digit number "1000000". This overflow to the eighth digit has to be corrected again. This additional correction or shift can be overcome by doing the shift after rounding. So, in this case the rounding is done initially for the product and the result is shown below.

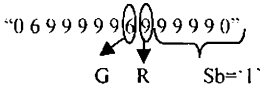


In this case, since $G > 5$, the most significant 7 digits are rounded up to get a value "1000000". The MSD of the rounded result is not zero anymore, and hence left shift is not required.

Now consider another example.

If $C1 = "2333330"$ and $C2 = "3000003"$

The 14 digit result of $C1 \times C2 =$



Since $G > 5$, the first 7 digits are rounded up, which results in a number “070000000”. The MSD of the rounded result is a zero, hence a left shift is done, and the exponent value is adjusted. The shifted result is now “7000000” if the ‘shifted in digit’ is zero, or a more erroneous result, “7000006” if the ‘shifted in digit’ is the Guard digit (G). Shifting before rounding would have given a result “6999997”. This result is equivalent to rounding the 7 digits excluding the MSD. It also performs the actual purpose of retaining the Guard digit. The error generated is more if rounding of the result is done before shifting. So, in such cases shifting has to be done first. In other words, select the result after rounding the n digits excluding the MSD. A suitable selection method is required to determine if rounding is to be done before shifting or vice versa.

In the DFP multiplier using RPS algorithm, the Sticky bit (Sb), Round digit (R) and Guard digit (G) generation are done in parallel with DFxP multiplication process. Rounding is accomplished by selecting either the product truncated to ‘ P_{limit} ’ digits or its incremented value. Initiation of the rounding operation along with the DFxP multiplication speeds up the entire process. After rounding, a ‘zero’ at MSD leads to a single digit shift towards left. The shifter module is a major component that contributes to the critical path delay of a floating point multiplier. So, a second rounding module is used to round the product, excluding the MSD. Then a selection is made between the two rounded results based on MSD. The exponent is adjusted, exceptions are set accordingly, and the result is encoded back in DPD.

The algorithm is explained using a 32-bit DFP multiplication. Initially, the two 32-bit operands are read from registers, and decoded to generate 7 significand digits, 8-bit biased exponent (E) and a Sign bit (S). The exceptions

such as ‘NaN’ and ‘Infinity’ are also decoded out from the 32-bit input. The output exceptions are set at the logic block named ‘Handle exception 1’ depending on the input exceptions. There are four exceptions that may be signalled during multiplication: Invalid operation, overflow, underflow, and inexact. The exponent is computed, and the sign bit of the result is determined as the XOR operation of the sign bits of the input operands. The significand digits are multiplied using a DFXP multiplier. A 7-digit \times 7-digit DFXP multiplier is used for 32-bit DFP input. This DFXP multiplier makes use of the RPS algorithm of Chapter 2 that generates the final product (FP) in $(n+1)$ cycles. Figure 3.2 gives the detailed schematic of the blocks included in the dashed box in Figure 3.1, for a 7-digit \times 7-digit DFXP multiplication.

The MSD of the significand of each input operand is suggested to be a non-zero number. This leads to a unique representation of the DFP inputs avoiding the need to count the number of leading zeroes. The smallest number that can be represented in this format for a 32-bit representation is 1000000×10^{-101} and the largest number is $9999999 \times 10^{+90}$. Any number that is greater than the largest number is encoded as ‘Infinity’ with an ‘Overflow’ exception. Similarly, any number that is less than the smallest number are truncated to ‘Zero’ with an ‘Underflow’ exception. If the result from an operation needs more digits than ‘ P_{limit} ’ which is the maximum precision of the significand, then the result will be rounded. If this rounding causes removal of non-zero digits then, the ‘Inexact’ exception is set. The sticky bit (Sb) generation is done in parallel with the DFXP multiplication. It is seen from Figure 2.26 that for a 7-digit \times 7-digit DFXP multiplication, the least significant product digits FP_4 - FP_0 are available after the fourth clock cycle. Hence, the sticky bit (Sb) can be generated after the fourth clock cycle. In

general, for an n -digit DFP multiplication using RPS algorithm, sticky bit generation can be done after $\lceil (n/2) \rceil$ cycles.

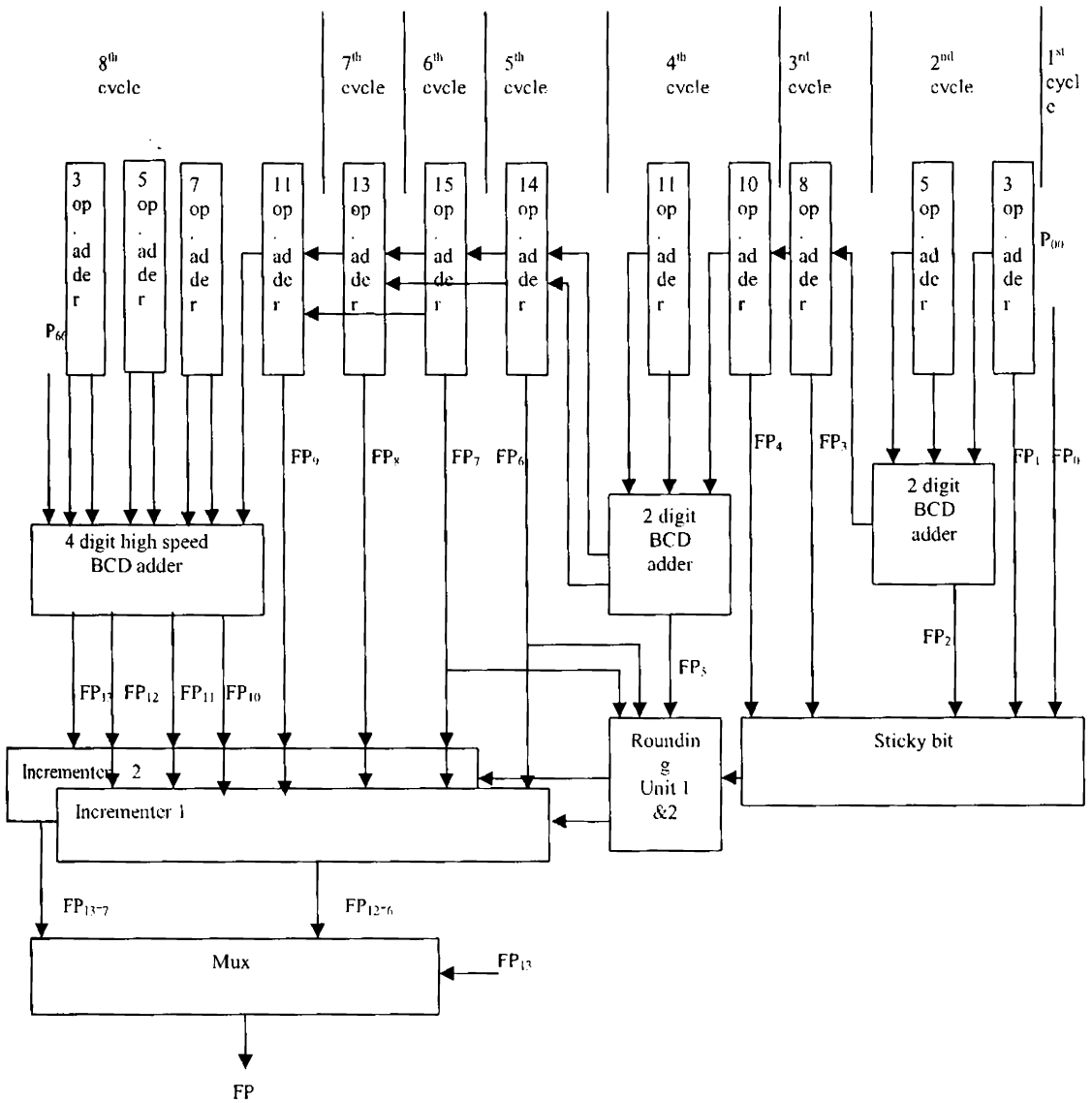


Figure 3.2: Block Schematic of DFP Multiplier using RPS algorithm and Rounding Unit

In this DFP multiplier, rounding is done for both the most significant n digits (using Incrementer 2 of Figure 3.2) and for the n digits excluding MSD

(using Incrementer 1 of Figure 3.2). The appropriate result is selected based on the MSD of the Incrementer 2. This improves the accuracy of the result if a left shift is required. Also, it avoids the need for a shifter module, and reduces the critical path delay. After rounding, the required ‘exponent adjust’ is done, and the exceptions are modified if necessary. The final result is then encoded back to DFP format.

3.3 DFP Multiplication using DDDM

The second approach for DFP multiplier design extends the iterative DFxP multiplier design using DDDM technique described in Chapter 2. The design for DFP multiplication is similar to the first approach except for the difference that DFxP multiplication is done using DDDM.

The operands encoded in DPD are decoded to get BCD significand digits and binary exponents. Then the product of significand digits of the two operands is generated using the iterative DDDM for DFxP multiplication. The least significant n digits of the $2n$ -digit DFxP product are available after $\lceil n/2 \rceil$ cycles. The most significant n digits are generated in $\lceil (n/2) + 1 \rceil^{\text{th}}$ cycle by the ‘Decimal Carry Propagate adder’. The Sticky bit (Sb), Round digit (R) and Guard digit (G) generation depend on the least significant n digits, and so are generated in $\lceil (n/2) + 1 \rceil^{\text{th}}$ cycle. Rounding is done in the $\lceil (n/2) + 2 \rceil^{\text{th}}$ cycle, and it is accomplished by the two rounding modules as explained in Section 3.2. The rounded result is selected between these two results based on the MSD. The exponent is adjusted, exceptions are set accordingly, and the result is encoded back in DPD.

3.4 DFP Multiplication using Parallel DFxP Multiplier

The parallel approach for DFP multiplier design uses the parallel DFxP multiplier described in Chapter 2. The operands encoded in DPD are decoded to get BCD significand digits and binary exponents. Then, the product of significand digits of the two operands is generated using parallel DFxP multiplication. The Sticky bit (Sb), Round digit (R) and Guard digit (G) are generated from the DFxP result. Rounding is accomplished by selecting either the DFxP product truncated to 'P_{limit}' digits or its incremented value depending on the status of Sb-bit, R and G digits. The exponent is adjusted, exceptions are set accordingly, and the result is encoded back in DFP format.

3.5 DFP MAC Unit

Decimal Floating Point MAC units perform the floating point fused multiply add operations $((A \times B) + C)$ of three DFP inputs that are in compliance with IEEE 754-2008 standard. The general block diagram for a Decimal Floating Point MAC unit is shown in Figure 3.3. The encoded inputs A, B and C are decoded to get the sign, exponent and significand parts of each input. The significands are then multiplied by the fixed point multiplier unit. The multiplied output is added with the C input. The result is corrected to the required number of digits and encoded back to standard decimal floating point format.

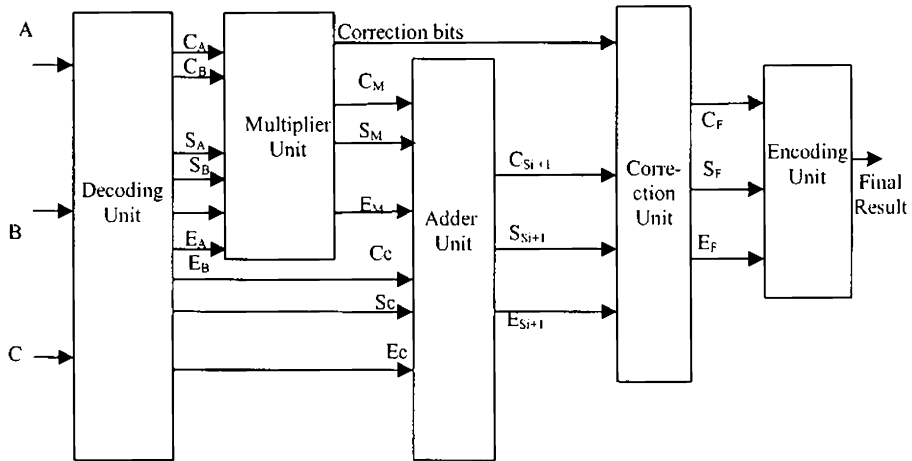


Figure 3.3: Block diagram of a Decimal Floating Point MAC unit

Multiply operation is performed by using the different methods described earlier. DFP adder using ripple carry BCD adders, kogge-stone adders and reduced delay BCD adders are implemented aiming at reducing the delay. The performances of these adders are compared in terms of area and delay in order to find the best configuration. The correction unit consists of the hardware for rounding, shifting, exponent correction and setting infinity, NaN, inexact flags. The correction unit also checks for error conditions such as overflow and underflow. Implementing error detection and correction in a design adds significant delays to the circuit since it requires the exponent and/or significand values to be modified.

3.5.1 DFP Adders

Addition is the most basic arithmetic operation of an ALU through which subtraction, multiplication and division can be realized. The decimal floating-point adder includes an alignment unit that aligns the significands (the part of a decimal floating-point number that contains its significand digits) of two floating-point numbers so that the exponents associated with the floating-point numbers have equal values. A binary adder adds the aligned significands. A correction unit and a rounding unit are also included in the floating-point adder to produce the final decimal floating point result. The different types of binary adders available in literature are Brent-Kung adder, Kogge-Stone adder, Carry-skip adder, carry look ahead adder, carry-select adder, Ling adder etc; the fastest being the Kogge-Stone adder.

The block schematic of a DFP Adder is shown in the Figure 3.4. A DPD to Decimal converter performs the conversion from Densely Packed Decimal form to the corresponding BCD significands and biased binary exponents. The module can be bypassed if the adder inputs are in BCD as in the case where the adder is a subsystem of a DFP system. The exponent difference module finds the difference between the exponents and gives the magnitude and the sign of the result. The multiplexer module then selects the final exponent of the result depending on the sign of the difference of the two exponents. The second multiplexer selects the suitable significands depending on the sign bit of the difference of the exponents. The output from this multiplexer is given to the 'shift right' module. The number of shifts of the shifter is determined by the magnitude of the difference of the exponents. The shifted output is then given to the adder unit. The adder performs the addition of the two inputs and gives the sum and carry. The scheme used for rounding

is 'Round to even' (RTE). The Decimal to DPD conversion module performs the conversion from BCD back to the IEEE 754-2008 floating point format. The module is not required if the output in BCD is to be fed in as BCD inputs to another subsystem such as in the case where the adder is a subsystem of another DFP system.

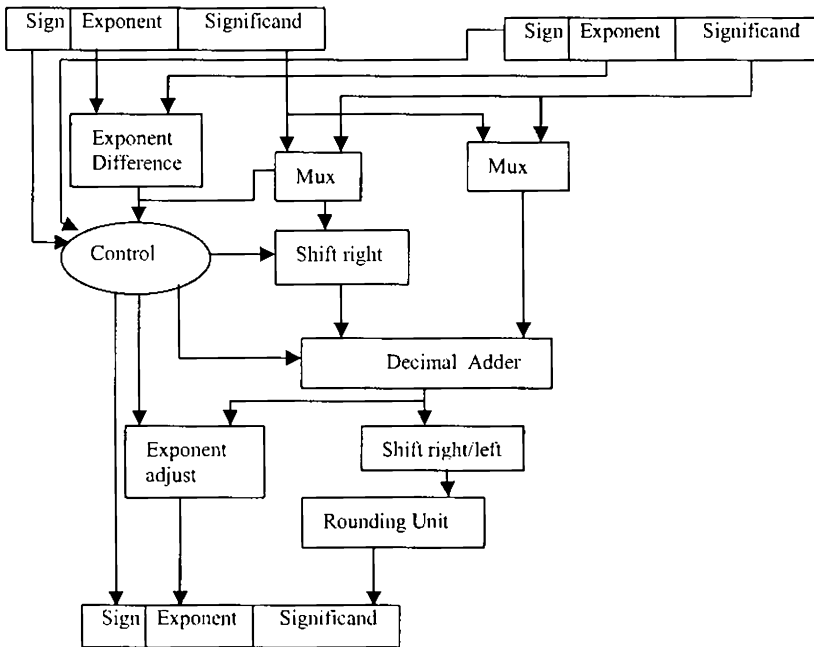


Figure 3.4: Block diagram of a Decimal Floating Point Adder unit

In this research three different types of 16-digit decimal floating point adders are implemented.

- DFP adder using concatenated ripple carry BCD adders
- DFP adder using kogge-stone adders.
- DFP adder using a Reduced Delay BCD Adder

The DFP implementations differ in the method in which the addition is done for the aligned significands. The rest of the modules of the DFP adder are the same for all the implementations. The adders implemented are of length 16 digits which can be used for the DFP MAC unit for 32-bit DFP input.

3.5.1.1 Ripple Carry BCD Adders

In this implementation, the addition of the aligned significands is done by 16 BCD adders. Each BCD adder is made of 2 ripple carry adders. The carry output of the BCD adder ripples to the next more significant adder. The ripple carry BCD adder is slow due to the rippling of carry through each adder stage.

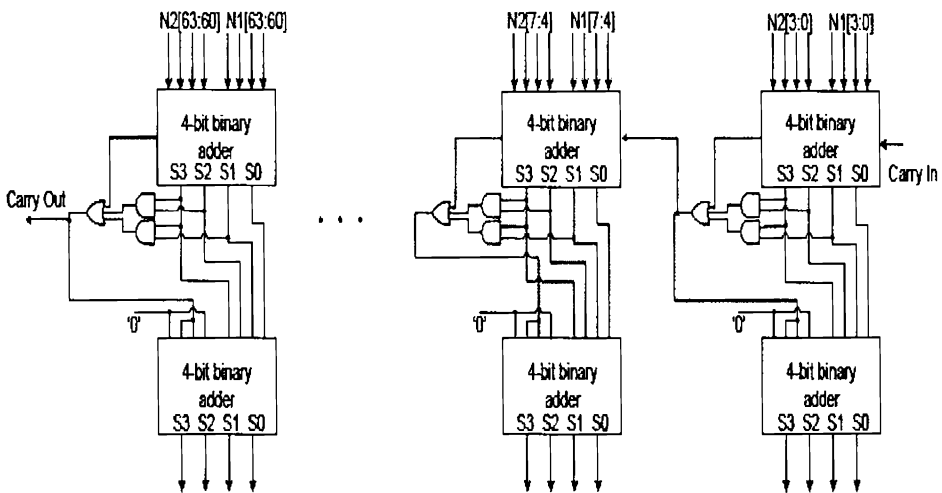


Figure 3.5: Concatenated BCD Adders

3.5.1.2 Kogge-Stone Adders

The Kogge-Stone adder is a parallel prefix form carry look-ahead adder and is considered as a faster adder design. The Kogge-Stone adder concept was developed by Peter M. Kogge and Harold S [Kogge, P. and Stone, H., 1973]. Stone. Figure 3.6 shows the an example of a a 4-bit Kogge-Stone adder [kogge-stone_adder/ freebase].

The implementation of the DFP adder using Kogge-Stone makes use of BCD adders implemented using 4-bit Kogge-stone adders as proposed by [Thompson *et al.*, 2004]. The adder consists of a pre-correction unit, a Kogge-Stone binary adder, which adds two pre-corrected operands, and a post correction unit. The speed is further increased by using ‘reduced delay BCD adders’ for decimal addition and is explained in next section.

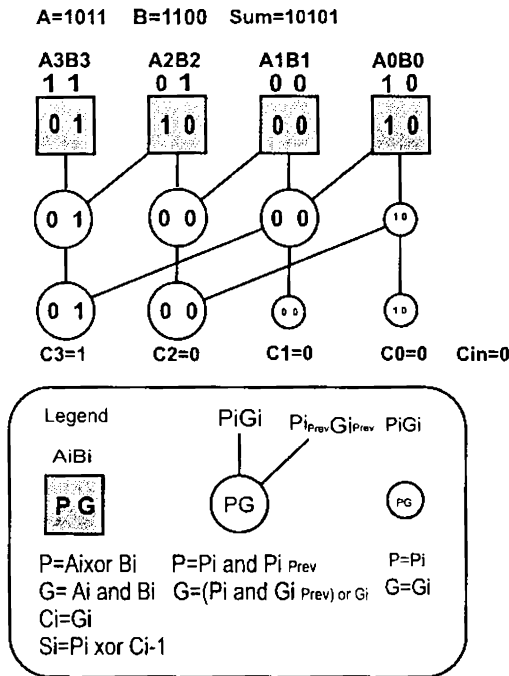


Figure 3.6: Kogge –Stone Adder

3.5.1.3 Reduced Delay BCD Adder

This DFP adder uses reduced delay BCD adder by [A. A. Bayrakci and A. Akkas, 2007]. The sum of a BCD addition can fall in any of the 3 cases:

Case 1: The sum of two BCD digits is smaller than 9. In this case, it is certain that there is no carry output, even if there is a carry input. Furthermore, the result for this digit does not require a correction.

Case 2: The sum of two BCD digits is greater than 9. In this case, a correction is required. Moreover, a carry output is produced regardless of the carry input.

Case 3: The sum of two BCD digits is exactly 9. In this case, the input carry determines whether a correction is required and whether a carry output is produced.

For the first two cases, the incoming carry has no effect on determining the carry output; therefore, the carry output can be determined without knowing the existence of the carry input. On the other hand, if the addition result is 9 (Case 3), then the input carry determines the existence of the carry output, which may ripple even up to the most significant digit. Therefore, Case 2 and Case 3 can be represented by a digit generate (DG) and a digit propagate (DP) signals, respectively. Figure 3.7 shows how the DG and DP signals of a digit are computed in the design [A. A. Bayrakci and A. Akkas, 2007]. After having all the DG and DP signals, the output carry for each digit is found by Equation (3.1).

$$\text{OutputCarry} = \text{DG} + \text{DP} \cdot \text{InputCarry} \quad (3.1)$$

The combination of the first level 4-bit adders and the Carry Network is shown in Figure 3.8 [A. A. Bayrakci and A. Akkas, 2007]. The carry value for each digit is computed inside the Carry Network using Equation (3.1).

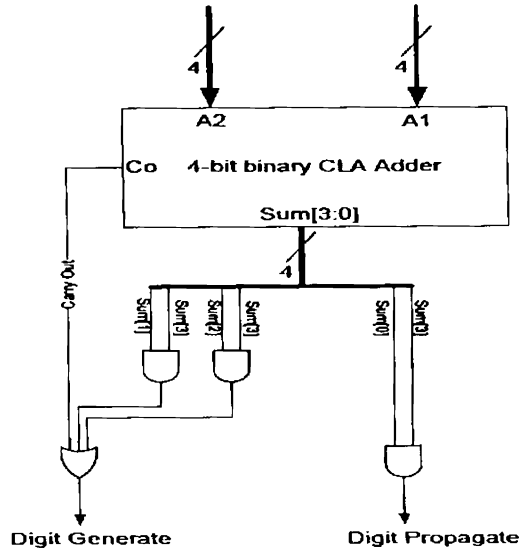


Figure 3.7: Adder and Analyzer Unit

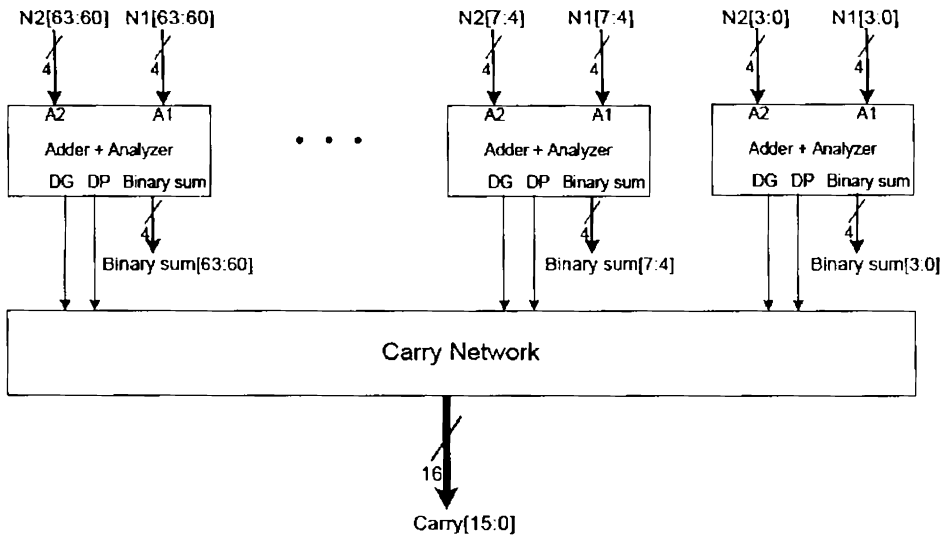


Figure 3.8: Adder, Analyzer and Carry Network

The carries computed by Carry Network are used in the correction step. Correction is done by adding 0, 1, 6, or 7 to the binary sum from the first

level adder. For each digit, the output carry and the input carry determine the value to be added for correction. Figure 3.9 shows the complete BCD adder including the 4-bit adders used for correction. Table 3.1 shows the correction value to be added for all cases [A. A. Bayrakci and A. Akkas, 2007].

Comparison of different DFP adder implementations using different BCD adders is given in Section 6.3.

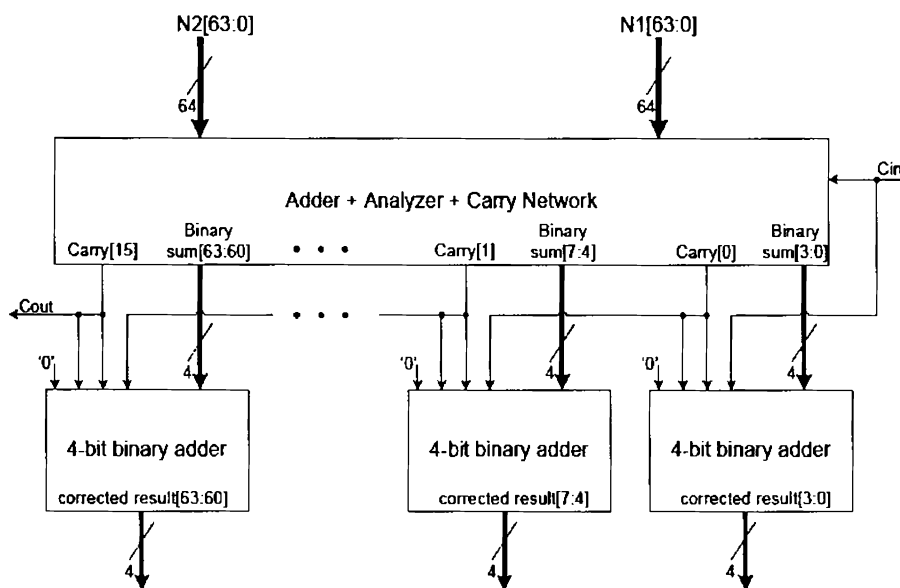


Figure 3.9: Reduced Delay BCD Adder

Table 3.1: Selection of the Value to be added for Correction

The value added for correction	Possible cases	
	Input carry from prev. digit	Output carry to next digit
0	0	0
6	0	1
1	1	0
7	1	1

3.6 Summary

The DFP multipliers presented includes the floating point extensions to the iterative DFxP multiplier design using RPS algorithm, DDDM and the parallel DFxP multiplier designs. These DFP multiplier designs are in compliance with IEEE 754-2008 standard. 32-bit DFP multipliers are synthesized to find the area and delay using Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library. These designs and the design in [M. A. Erle, B. J. Hickmann and M. J. Schulte, 2009] for the DFP Multiplication are synthesized in the same environment.

A delay reduction is achieved for the approach using RPS algorithm because of the initiation of rounding process during the DFxP multiplication. This parallelism decreases the delay of the last cycle, which in turn reduces the worst case period and increases the throughput. When multiplying two DFP numbers with n -digit significands using this approach, the worst case latency is $(n+2)$ cycles, and initiation interval is $(n+1)$ cycles. The DFP design using DDDM for DFxP multiplication requires lesser number of clock cycles compared to first approach and that in [M. A. Erle, B. J. Hickmann and M. J. Schulte, 2009]. The latency to complete n -digit \times n -digit multiplication is almost halved compared to single digit design with an increase in area of 50%. The latency for the multiplication of DFP numbers with n -digit significands using DDDM is $\lceil (n/2) + 2 \rceil$ cycles, and a new multiplication can begin every $\lceil (n/2) + 1 \rceil$ cycle. This in turn reduces the total delay even though the worst case cycle time is more, giving a delay reduction compared to the delay in [M. A. Erle, B. J. Hickmann and M. J. Schulte, 2009]. Compared to the first

approach using RPS algorithm, the second approach is more regular and occupies lesser area, but has more delay.

Parallel designs are adopted when latency and throughput are of more importance than area. Comparison of parallel design with the iterative designs and that of [M. A. Erle, B. J. Hickmann and M. J. Schulte, 2009] is done.

Decimal Floating Point MAC unit implements the fused multiply-add operation with a single final rounding after add operation. The fused multiply add unit uses parallel and iterative multipliers and a floating point adder unit. The DFP adder is implemented using ripple carry BCD adders, kogge stone adders and reduced delay BCD adders. Comparison of DFP adders shows that the 'reduced delay adder' achieves the highest speed. The simulation results of different decimal floating point multipliers, adders and MAC unit are given in Section 6.3.

Chapter 4

Reversible Circuits for Decimal Adders

Low power designs with high performance are given prime importance, since power has become an important design consideration. In recent years, reversible logic has emerged as one of the most important approaches for power optimization. So, reversible logic is in demand in high-speed power aware circuits. In this chapter different designs for reversible logic implementation of BCD adder are presented. This chapter also suggests two new universal 4×4 'reversible RPS gates' that can function as a reversible 4-bit Binary to BCD converter with a garbage count of zero. The reversible circuits presented here forms the initial step in the building of complex reversible systems. Complex reversible systems can execute more complicated operations for a reversible Decimal MAC unit. Low power circuit designed in reversible logic for Hamming code generation and error detection using a new 4×4 reversible Hamming Code Gate (HCG) is also presented.

4.1 Reversible Logic

Low power designs with high performance are given prime importance by researchers, as power has become a first-order design consideration. While efforts are being made to reduce power dissipation due to leakage currents, alternate circuit design considerations are also gaining importance. Energy loss during computation is an important consideration in low power digital design. Landauer's principle states that a heat equivalent to $kT \ln 2$ is generated for every bit of information lost, where k is the Boltzmann's constant and T is the temperature [R. Landauer, 1961]. At room temperature, though the amount of heat generated may be small it cannot be neglected for low power designs. The amount of energy dissipated in a system bears a direct relationship to the number of bits erased during computation. Bennett showed that energy dissipation would not occur if the computations were carried out using reversible circuits [Bennett C., 1973] since these circuits do not lose information. Information is lost when the input data cannot be uniquely recovered from the output data. A gate that does not lose information is called a reversible gate (for example, an inverter). A completely specified n -input, n -output Boolean function is called a reversible function, if it maps each input vector to a unique output vector and vice versa. There is a significant difference in the synthesis of logic circuits using conventional gates and reversible gates [T. Toffoli, 1980]. While constructing reversible circuits with the help of reversible gates, fan-out of each output must be 1 without feedback loops. As the number of inputs and outputs are made equal there may be a number of unutilized outputs in certain reversible implementations. The unutilized outputs from a reversible gate/circuit are called "garbage". This is

the number of outputs added to make an n-input k-output function reversible. For example, a single output function of n variables will require at least (n-1) garbage outputs. An important aspect for evaluating reversible circuits is the garbage count. Hence, one of the major issues in designing a reversible circuit is in garbage minimization.

Classical logic gates such as AND, OR, and XOR are not reversible. Hence, these gates dissipate heat and may reduce the life of the circuit. In recent years, reversible logic has emerged as one of the most important approaches for power optimization with its importance in nanotechnology and quantum computing.

4.2 Reversible gates

This section describes reversible logic gates that are used in various implementations of a BCD adder.

4.2.1 New Gate (NG)

Figure 4.1 shows a 3×3 New Gate [Md. M. H. Azad Khan, 2002]. New Gate can be used as a gate that generates an AND gate, an OR gate or an XOR gate. If $B=0$, then $Q=C$ and $R=(A'C')\oplus 1=A+C$. Similarly, when $C=0$, then $Q=AB$ and $R=A\oplus B$ which are the carry and sum outputs of a half adder.

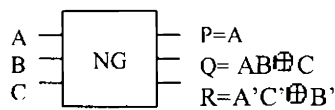


Figure 4.1: 3×3 New Gate (NG)

Any Boolean function can be written in Positive Polarity Reed Muller (PPRM) form. A New Gate can be considered as a universal gate since any Boolean function in PPRM form can be realized using only New Gates.

4.2.2 Toffoli Gate (TG)

Figure 4.2 shows a 3×3 Toffoli Gate [E. Fredkin, T. Toffoli, 1982]. Like New Gate, Toffoli gate can also be used to generate an AND gate or an XOR gate. If $C='0'$, then $R=AB$ and if $B='1'$ then $R=A \oplus C$. TG can also be considered as a universal gate since any Boolean function in PPRM form can be realized using only TGs.

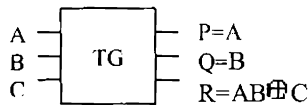


Figure 4.2: Toffoli Gate (TG)

4.2.3 New Toffoli Gate (NTG)

Figure 4.3 shows a 3×3 New Toffoli Gate [H. Md. H.Babu, 2003] or Peres Gate. New Toffoli gate can also be used to generate an AND gate or an XOR gate. If $C='0'$, then $Q=A \oplus B$ and $R=AB$.

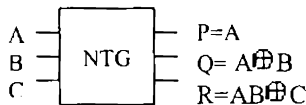


Figure 4.3: 3×3 New Toffoli Gate (NTG)

NTG is also a universal gate since any Boolean function in PPRM form can be realized using only NTGs.

4.2.4 TSG

Figure 4.4 shows a TS Gate [H. Thapliyal and M.B Srinivas, 2005]. A full adder circuit can be implemented completely by a single TSG. TSG is also a universal gate since any gate can be implemented using only TSGs.

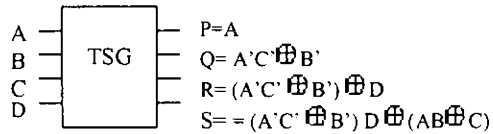


Figure 4.4: 4×4 TS Gate (TSG)

4.2.4 Fredkin Gate (FRG)

Figure 4.5 shows a Fredkin Gate (FRG) [E. Fredkin, T. Toffoli, 1982]. This is a parity preserving reversible logic gate which satisfies the condition

$$A \oplus B \oplus C = P \oplus Q \oplus R.$$

In general, the following condition is valid for a parity preserving reversible gate:

$$\oplus \sum X_i = \oplus \sum Y_i$$

where 'X' indicates an input, 'Y', an output, and 'i' the number of inputs or outputs of the reversible gate.

A circuit implemented using parity preserving gates makes it suitable for fault detection. The 'Q' or 'R' output expression of FRG is same as that of a 2:1 multiplexer. Any Boolean function can be realized using 2:1 multiplexer. So FRG is also a universal gate.

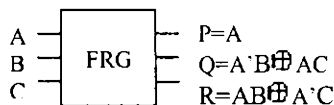


Figure 4.5: Fredkin Gate (FRG)

4.2.5 Feynman Gate

Figure 4.6 shows a Feynman Gate [R. Feynman, 1985]. Feynman Gate can be used as a copying gate. Since a fanout greater than one is not allowed, this gate is useful for duplication of the required outputs. If $B=0$, then $P=A$ and $Q=A$.

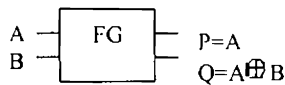


Figure 4.6: 2×2 Feynman Gate (FG)

A 3×3 Feynman Double Gate (F2G) [B. Parhami, 2006] has 3 inputs A, B, C and 3 outputs $P=A$, $Q=A \oplus B$, $R=A \oplus C$ as shown in Figure 4.7. F2G is a copying gate as well as a parity preserving reversible gate.

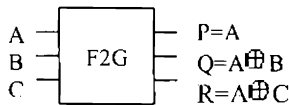


Figure 4.7: 3×3 Feynman double Gate (F2G)

4.3 Reversible Full Adders

A full adder is an integral component of a BCD adder and has 3 inputs and 2 outputs. To make a full adder reversible some garbage outputs are to be added. A reversible full adder circuit is proposed which can be realized with two garbage outputs [H. Md. H. Babu, A. R. Chowdhury, 2006]. In the full-adder circuit, there are three input combinations (0, 0, 1), (0, 1, 0) and (1, 0, 0) for which the output is same (1, 0). So, at least two garbage bits are required to make a unique output combination for each input combination.

4.3.1 Full adder using NG

Figure 4.8 shows the implementation of full adder using NG and NTG gates with 2 garbage outputs. It is seen that the implementation requires 2 gates at 2 levels with 2 garbage outputs to generate the sum and carry outputs. So, this is an optimum solution in terms of number of garbage outputs. The full adder gives an additional delay of one NTG to generate 'Carry' after receiving C_{in} . The total delay for generating the C_{out} for an n-bit ripple adder using such full adders in terms of number of gate delay is

$$T_{out} = 1+n \tag{4.1}$$

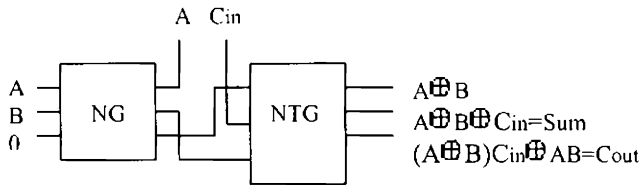


Figure 4.8: A reversible full adder with NG and NTG

4.3.2 Full adder using NG

The reversible implementation of full adder using only NG is shown in Figure 4.9. It is seen that the implementation requires 3 gates at 3 levels with 4 garbage outputs to generate the sum and carry outputs. So, this is not an optimum solution. C_{in} is the last input to be received for the full adder. The C_{in} input passes through a maximum of 2 NGs to generate the 'Carry' output. Thus the full adder gives an additional delay of 2 NGs after receiving C_{in} . The total delay for generating C_{out} for an n-bit ripple adder implemented using such full adders in terms of NG delay is

$$T_{\text{cout}} = 1 + 2n \quad (4.2)$$

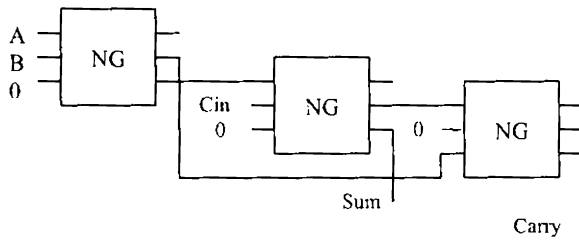


Figure 4.9: Reversible full adder using NG

4.3.3 Full adder using NTG

The reversible implementation of full adder using only NTG is shown in Figure 4.10. It is seen that the implementation requires 2 gates at 2 levels with 2 garbage outputs to generate the sum and carry outputs. This is similar to the implementation in Figure 4.8 and the total delay for generating the C_{out} for an n-bit ripple adder using such full adders in terms of NTG delay is

$$T_{\text{cout}} = 1 + n \quad (4.3)$$

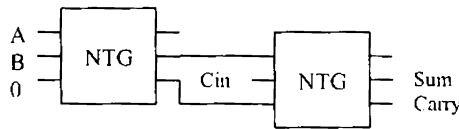


Figure 4.10: Reversible full adder using NTG

4.3.4 Full adder using TG

The reversible implementation of full adder using only Toffoli Gates is shown in Figure 4.11. The implementation requires 4 gates at 4 levels with 4 garbage outputs to generate the sum and carry outputs. So, this is not an optimum solution. The full adder gives an additional delay of one TG to generate 'Carry' after receiving C_{in} . The total delay for generating the C_{out} for

an n-bit ripple adder implemented using such full adders in terms of TG delay is

$$T_{\text{cout}} = 2+n \quad (4.4)$$

Here 'Sum' is generated after one more TG delay. So the total delay in generating the 'Sum' output for an n-bit ripple adder is $3+n$.

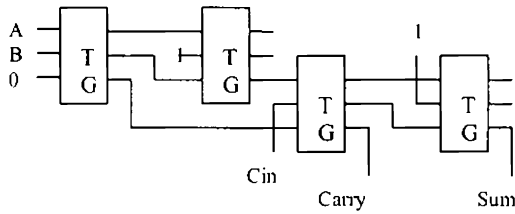


Figure 4.11: Reversible full adder using TG

4.3.5 Full adder using TSG

The reversible implementation of full adder using only TSG is shown in Figure 4.12.

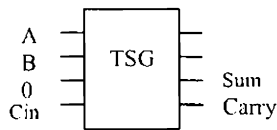


Figure 4.12: Reversible full adder using TSG

The implementation requires only 1 gate with 2 garbage outputs to generate the sum and carry outputs. So, this is an optimum solution in terms of number of gates and garbage outputs. But, TSG is a 4×4 gate and has a more complex structure compared to the other reversible gates so far discussed. The

total delay for generating the C_{out} for an n-bit ripple adder implemented using such full adders in terms of TSG delay is

$$T_{cout} = n \quad (4.5)$$

4.3.6 Full adder using FRG

Fredkin gate is a parity preserving reversible logic gate. A number of parity preserving reversible full adders are available in literature [B. Parhami, 2006], [Dmitri Maslov, 2003]. Figure 4.13 shows an implementation of full adder using parity preserving Fredkin gates.

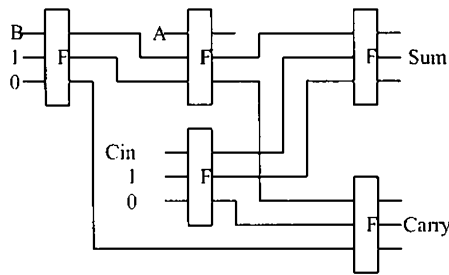


Figure 4.13: Reversible Full adder using Fredkin Gates

This full adder implementation requires only 5 Fredkin gates at 3 levels compared to 3-level 6-gate (5 Fredkin gates and 1 Feynman gate) implementation in [B. Parhami, 2006], and 5-level 5-gate implementation in [J.W.Bruce et al., 2002] while observing the fanout restrictions. The Fredkin gate implementation in Figure 4.13 generates 5 garbage outputs. So, this is not an optimum solution in terms of gates or garbage. But the use of parity preserving Fredkin gates makes it a reversible fault tolerant implementation

suitable of single error detection. Total delay for generating the C_{out} for an n -bit ripple adder using such full adders in terms of FRG delay is

$$T_{cout} = 1+2n \quad (4.6)$$

Table 4.1 shows a comparative analysis of different implementations of full adders using universal reversible gates.

Table 4.1: Comparison of Reversible Full Adders

Type of gate	Number of			Delay for n-bit adder		Parity preservation
	gate	level	garbage	C_{out}	Sum	
NG & NTG	2	2	2	$1+n$	$1+n$	No
NG	3	3	4	$1+2n$	$2n$	No
NTG	2	2	2	$1+n$	$1+n$	No
TG	4	4	4	$2+n$	$3+n$	No
TSG	1	1	2	n	n	No
FRG	5	3	5	$1+2n$	$1+2n$	Yes

Several other reversible implementations of full adders are available in literature in [Md. M. H. Azad Khan, 2002], [H. Md. Hasan Babu *et al.*, 2003], [H. Md. Hasan Babu *et al.*, 2004], [Dmitri Maslov, 2003], [J. W. Bruce *et al.*, 2002]. But the implementation of a full adder using TSG takes least number of gates, and produces least number of garbage outputs.

4.4 Reversible Decimal Adder

A conventional BCD adder shown in Figure 4.14 has three blocks: 4-bit binary adder, 6-correction circuit and a modified special adder. 4-bit full adder adds the BCD inputs and generates a binary sum, S (S_{3-0}). This output is checked for a value greater than '9' or for a carry out, by the 6-correction circuit which generates a '6-correction' bit, 'L' using Equation 4.7.

$$L = C_{out} + S_3 (S_1 + S_2) \quad (4.7)$$

The inputs to the second adder stage are S (S_{3-0}) and 4-bit number N (N_{3-0}) whose value is 6 (0110_2) or 0 (0000_2) depending on 'L' bit. So, N_0 and N_3 are always zero, and N_1 and N_2 is 'L' bit. To reduce the hardware and to increase the speed of the circuit, the final adder stage (special adder) is a modified version of the 4-bit binary adder with two half adders and one full adder.

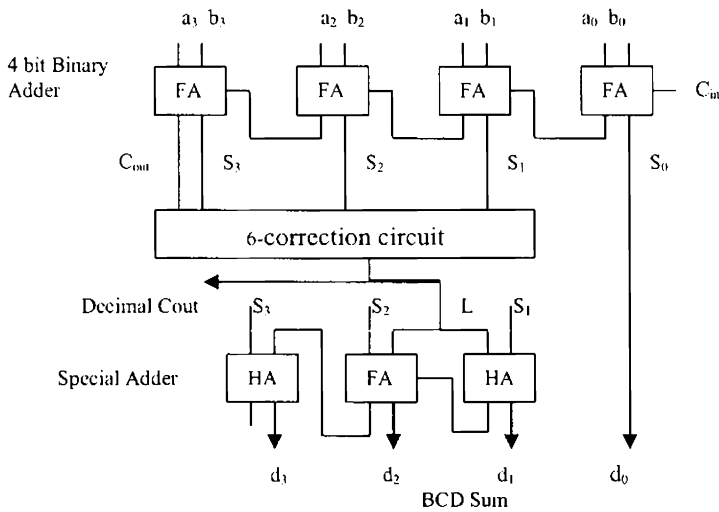


Figure 4.14: BCD Adder

A reversible conventional BCD adder was proposed in [Hafiz Md. Hasan Babu and A. R. Chowdhury, 2005] using conventional reversible gates. In [Hafiz Md. Hasan Babu and A. R. Chowdhury, 2005], a full adder design using two types of reversible gates, NG (New Gate) and NTG (New Toffoli Gate) with 2 garbage outputs was implemented. The BCD adder was then designed using these full adders. The implementation was improved in [Himanshu. Thapliyal, S. Kotiyal and M.B Srinivas, 2006] using TSG reversible gates. But, this approach was not taking care of the fanout restriction of reversible circuits, and hence it was only a near-reversible implementation. In this research, a modified version of decimal addition using reversible gate is designed which is a fully reversible circuit with a fanout of 1. The reversible design of the BCD adder is done using the reversible gates such as TSG, FG and NG. Since a full adder can be implemented using one TSG, a 4-bit binary reversible adder implementation using TSG gates requires 4 TSG gates and produces 8 garbage outputs. Reversible implementation of the 6-correction circuit is given in Figure 4.15. For reducing the number of gates, the 6-correction circuit output 'L' can be modified as in Equation (4.8).

$$L = \text{Cout} + S_3(S_1 + S_2) = \text{Cout} \oplus S_3(S_1 + S_2) \quad (4.8)$$

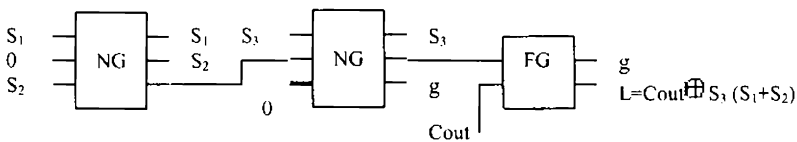


Figure 4.15: Reversible Implementation of 6-correction Circuit

It can be seen that the implementation requires 3 gates (2 NGs and 1 FG) to produce the 6-correction output, 'L', and the sum outputs (S_{3-1}) along with 2 garbage outputs. The S_1 , S_2 and S_3 outputs produced without using any copying gate (FG) can be used as inputs for the next stage. This gives a reduction of 3 gates and 4 garbage outputs compared to the implementation in [Hafiz Md. Hasan Babu and A. R. Chowdhury, 2005].

Special adder is implemented using 4 gates of 3 types (NG, TSG, 2 FGs). It is already seen that a 3×3 NG can implement a half adder, and a 4×4 TSG can implement a full adder. An FG replaces the final half adder in the special adder. This is because only the sum bit (d_3) is required as decimal sum output, and the carry is discarded from the final addition. So, using an NG will give rise to 2 garbage outputs while an FG will produce only one garbage output. The BCD sum is indicated as $d_{3,0}$ and carryout from the stage as 'Decimal C_{out} ' in Figure 4.16. The complete circuit of the new BCD adder is given in Figure 4.17.

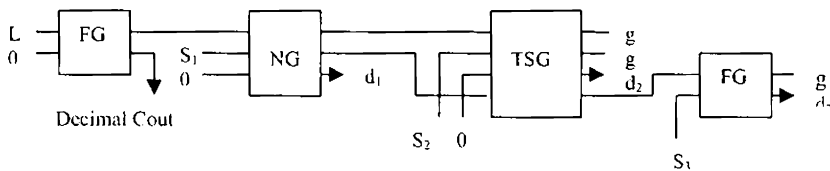


Figure 4.16: Reversible Implementation of Special Adder

A modified version of decimal addition using reversible gates that further reduces the number of gates and garbage outputs with a fan-out of 1 is designed. The design is implemented using 3 types of reversible gates. The modified design of this special adder speeds up the addition process with a reduction in the number of gates as shown in Figure 4.18.

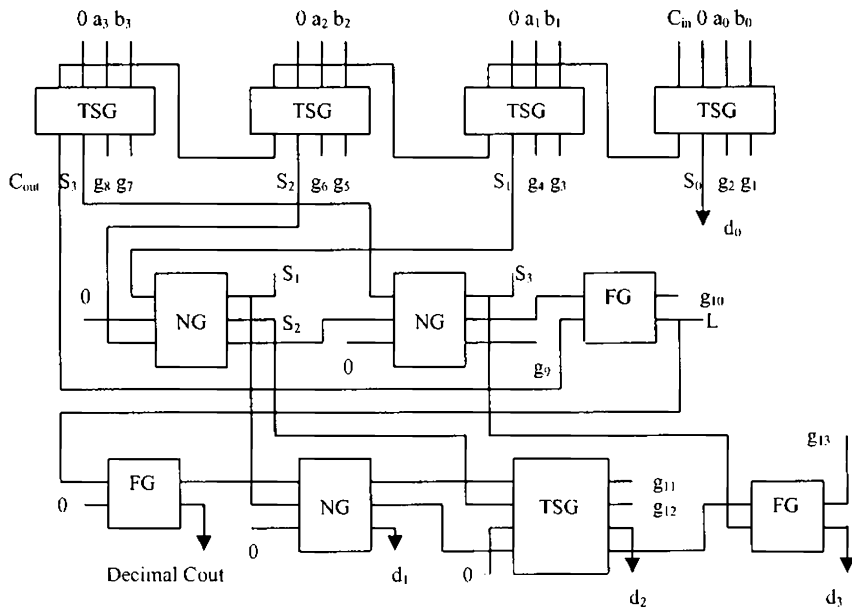


Figure 4.17: Reversible Implementation of BCD Adder

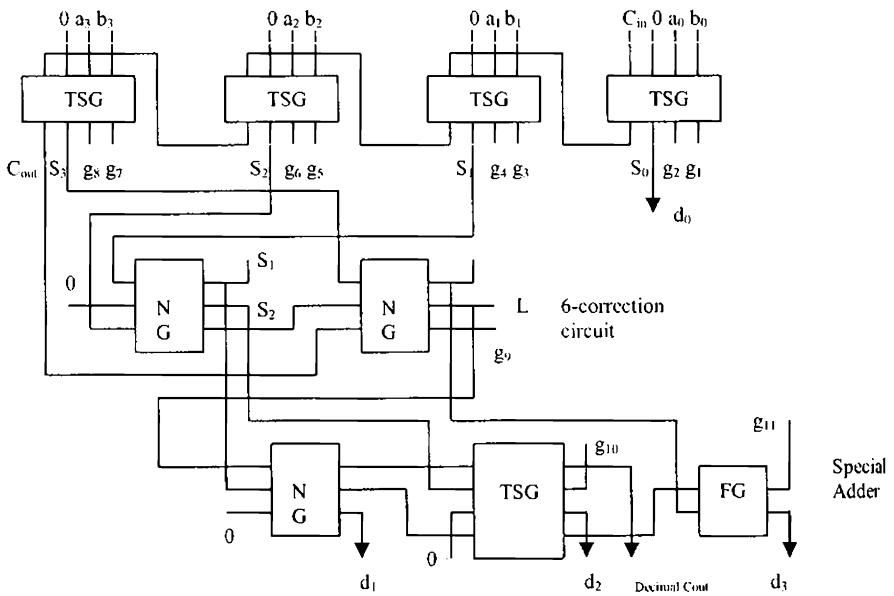


Figure 4.18: Modified Implementation of BCD Adder

The reversible BCD adder implementation is done using this modified special adder with all the features of reversible logic synthesis. The restrictions of reversible circuits are also taken care of in this design while achieving a reduction in the number of gates. The design is done using 3 types of reversible gates and results in lesser number of garbage outputs.

Further reduction in number of logical computations was achieved by using HNG gates in the implementation by [M. Haghparast and K. Navi, 2008]. Figure 4.19 shows an HNG gate. A full adder circuit can be implemented completely by a single HNG similar to a TSG. The logical complexity counted in terms of number of logical computations such as number of XOR, NOT, AND operations are less for HNG compared to TSG. HNG is also a universal gate since any gate can be implemented using only HNGs.

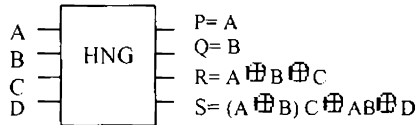


Figure 4.19: 4×4 HN Gate (HNG)

All these implementations are for conventional BCD adders. These are relatively slow, and are implemented using different types of reversible gates. Next section deals with the implementation of fast decimal adders with only one type of reversible gate.

4.5 Reversible Fast Decimal Adders

This research presents reversible implementations of three different fast decimal adders: Quick Addition of Decimals (QAD), carry select, and hybrid decimal adders. These implementations result in reduced number of gates and garbage outputs compared to the reversible implementation of carry skip BCD adder proposed in [Himanshu Thapliyal, S. Kotiyal, and M. B. Srinivas, 2006]. These multi-digit BCD adders are implemented using parity preserving reversible Fredkin gates. Fredkin gates are conservative reversible gates. A gate is conservative if the Hamming weight (number of logical 1's) of its input equals the Hamming weight of its output. If a gate is conservative and reversible then it is parity preserving. Fault detection can be done by using parity-preserving reversible logic gates. The feasibility of the parity-preserving approach in the design of reversible logic circuits was demonstrated by [B. Parhami, 2006] with examples of adder circuits. Parity checking is one of the oldest as well as one of the most widely used methods for error detection in digital systems. The parity preservation proves useful for ensuring the robustness of reversible logic circuits.

4.5.1 Quick Decimal Adder

The BCD adder shown in Figure 4.20 consists of a 4-bit binary adder, a 6-correction circuit, and a modified special adder along with a circuit (3-input AND, 2-input OR) to generate decimal carry out (d_{cout}).

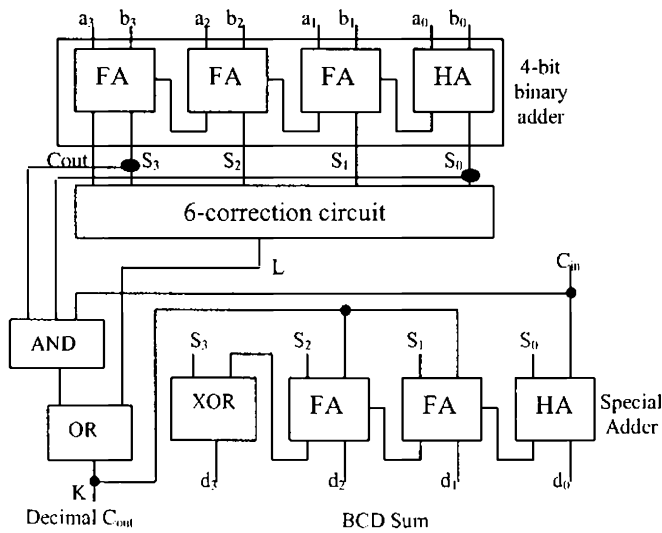


Figure 4.20: BCD adder for Quick Addition of Decimals (QAD)

On receiving C_{in} the circuit checks whether the sum, S is '9' and C_{in} is '1'. It then generates 'K' bit using Equation (4.9).

$$K = S_3 S_0 C_{in} + L \quad (4.9)$$

The inputs to the second adder are S (S_{3-0}) and 4-bit number, N (N_{3-0}) whose value is depending on 'K' and ' C_{in} ' as given below.

If $K=1$ then $N=6$ (0110_2) if $C_{in}='0'$

$N=7$ (0111_2) if $C_{in}='1'$

else $N=0$ (0000_2) if $C_{in}='0'$

$N=1$ (0001_2) if $C_{in}='1'$

So, N_3 is always zero. N_2 and N_1 is K -bit and N_0 is ' C_{in} '. The final adder stage (4-bit special adder) is a modified version of the 4-bit binary adder consisting of a half adder, 2 full adders and an XOR gate. This will reduce the

hardware, and will increase the speed of the circuit. The implementation of the special adder is shown in Figure 4.20.

A 4-Digit Quick Decimal Adder accepts two 4 digit decimal numbers as inputs in BCD (16 bits), and generates the BCD_{sum} . Figure 4.21 shows the 4-digit QAD implementation. The shaded parts indicate the critical path.

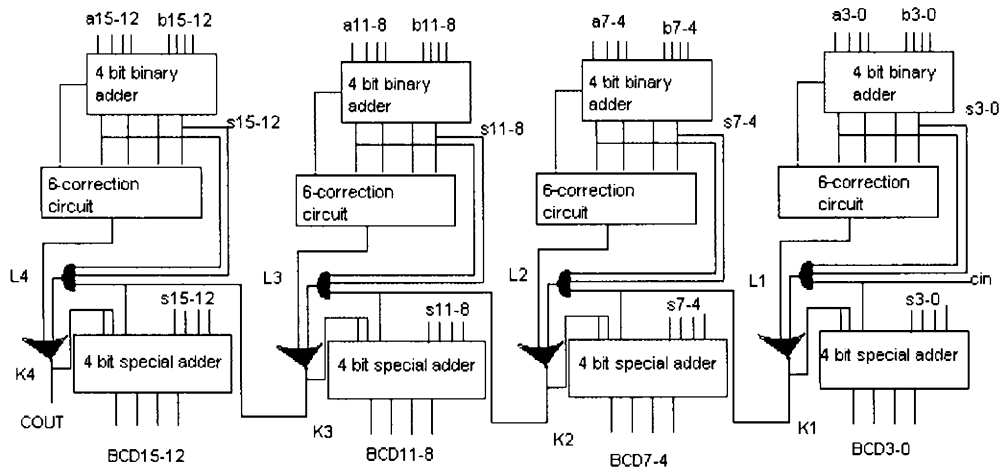


Figure 4.21: 4-Digit Quick Decimal Adder

The first stage addition is carried out in parallel for all digits, and a ‘6-correction’ bit (L_i) is generated simultaneously by all the stages. So, the delay up to this stage is given as

$$T_{\text{delay}} = T_{\text{adder}} + T_{6\text{-correction}} \quad (4.10)$$

where T_{adder} is the delay of the 4-bit binary adder, and $T_{6\text{-correction}}$ is the delay of 6-correction circuit

On receiving the C_{in} , the Decimal C_{out} bit (K_i) is generated after the delay of a 3-input AND gate and a 2-input OR gate at each stage as given in (4.11).

$$T_{\text{dcout}} = T_{\text{and}} + T_{\text{or}} \quad (4.11)$$

So, the delay in generating Decimal C_{out} (d_{cout}) after receiving C_{in} is $4T_{dcout}$ for a 4-digit adder. In general, for an N-digit BCD adder the delay for generating the Decimal C_{out} (d_{cout}) after receiving ' C_{in} ' is mT_{dcout} . The total delay of the m-digit adder is given in (4.12).

$$T_{mdigit} = T_{adder} + T_{6\text{-correction}} + NT_{dcout} + T_{sp\text{-adder}} \quad (4.12)$$

where $T_{sp\text{-adder}}$ is the delay of the special adder.

The total delay for an m-digit 'Quick Decimal Adder' is 'N' times the delay of the additional logic required (a 3-input AND and a 2-input OR) along with the delay of single stage BCD adder which is

$$T_{1stage} = (T_{adder} + T_{6\text{-correction}} + T_{sp\text{-adder}}) \quad (4.13)$$

4.5.1.1 Parity Preserving Reversible Quick Decimal Adder

The reversible implementation of the new quick decimal adder is done using Fredkin gates. The basic component of any adder is a full adder. Figure 4.22 and Figure 4.23 show the implementation of a half adder and a full adder using parity preserving Fredkin gates. The full adder implementation requires only 5 Fredkin gates at 3 levels, compared to 3 level 6 gate (5 Fredkin gates and 1 Feynman gate) implementation in [B. Parhami, 2006], and 5 level 5 Fredkin gate implementation in [J.W. Bruce et al., 2002] while observing the fanout restrictions.

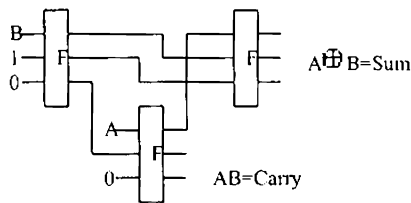


Figure 4.22: Half adder using Fredkin Gates

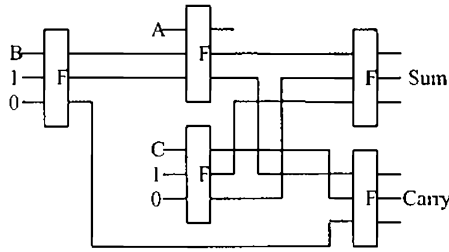


Figure 4.23: Full adder using Fredkin Gates

The 4-bit binary adder realized using one half adder and 3 full adders will achieve a reduced delay by using these implementations. The least significant half adder requires a path delay of two FRGs to generate C_0 from the addends. Then the carry ripples through the subsequent full adders with a path delay of two FRGs per bit. This is because the first Fredkin gate of all full adders works in parallel with the first Fredkin gate of the half adder in an n -bit binary adder. But in the implementation in [B. Parhami, 2006] the delay is of 3 levels. So, an advantage of single delay level/bit is achieved in the implementation presented. The delay to generate C_{out} in the n -bit binary adder is

$$T_{c\text{-ripple}} = 2 + 2(n-1) \quad (4.14)$$

For a conventional n -bit adder, the delay is

$$T_{c\text{-ripple (conventional)}} = 3n \quad (4.15)$$

For a BCD adder this delay is the delay with $n=4$ for each digit. In QAD adder the delay remains the same as the delay of a single digit for m -digit addition, since all digits are added in parallel.

The parity preserving reversible implementation of the 6-correction circuit is shown in Figure 4.24. The implementation requires 3 FRGs to

generate the 'L' output where $L=C_{out}+S_3(S_1+S_2)$. This circuit takes only 2 more delays after generating the Sum to generate the L bit. The delay to generate 'L' bit from the inputs for QAD adder is as given in Equation (4.16).

$$T_L = 4+2(n-1) \tag{4.16}$$

The delay for 'L' bit generation for conventional BCD adders is given in Equation (4.17).

$$T_{L(\text{conventional})} = 2+3n \tag{4.17}$$

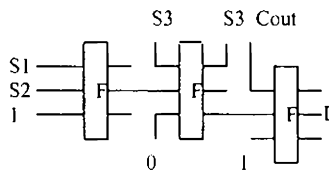


Figure 4.24: Generation of 'L' bit using Fredkin Gates

Figure 4.25 shows the reversible implementation for generating Decimal C_{out} (d_{cout}) or 'K' bit. The design makes use of three FRGs. It is seen that the first gate generate S_3S_0 as soon as the sum output 'S' is produced. When C_{in} is received the next two FRGs generate the Decimal C_{out} or 'K' bit. So, the additional delay in each stage is only due to the two FRGs.

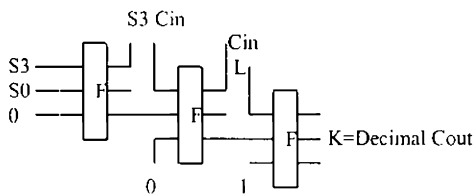


Figure 4.25: Generation of 'K' bit using Fredkin Gates

For an N-digit BCD addition the delay for the generation of 'K' bit or Decimal C_{out} from the BCD inputs is

$$T_{d\text{-cout}} = 4+2(n-1) +2N \tag{4.18}$$

Special adder implemented using one half adder, two full adders and one XOR gate requires 15 Fredkin gates (3 for half adder, 5 for each full adder, 2 for XOR gate) to generate the BCD sum $d_{3,0}$. The Decimal C_{out} or the 'K' bit is the last input for the special adder. The 'K' input passes through a maximum of 5 Fredkin gates to generate the BCD sum $d_{3,0}$. So, the special adder gives an additional delay of 5 Fredkin gates. The total delay in generating the BCD sum ($d_{3,0}$) from the inputs in terms of Fredkin gate delay is

$$T_{d\text{-sum}} = 9 + 2(n-1) + 2N \quad (4.19)$$

For a conventional BCD adder the final adder is a 4-bit binary adder with delay as given in (4.19) with $n=4$. So the total delay given by an N-digit conventional BCD adder is

$$T_{d\text{-sum (conventional)}} = (2+3n + 3n)N = (2+6n)N \quad (4.20)$$

4.5.2 Carry Select BCD Adder

The carry select BCD adder is shown in Figure 4.26. On receiving C_{in} , a 'K' bit is generated using Equation (4.21).

$$K = S_3S_0C_{in} + L = P_0C_{in} + L \quad (4.21)$$

where P_0 is the carry propagate signal (S_3S_0)

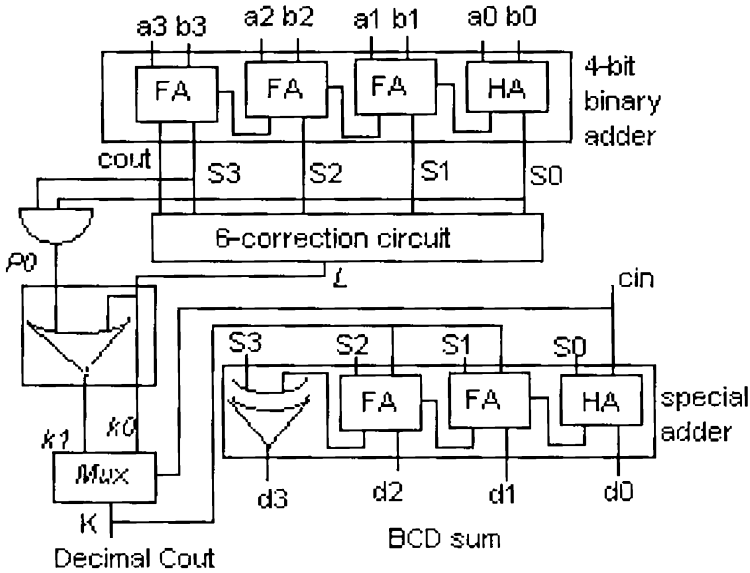


Figure 4.26: Carry Select BCD adder

If the carry select technique is adopted for 'K' bit generation then k_1 denotes the 'K' bit with $C_{in} = 1$ and k_0 with $C_{in} = 0$. This is given by $k_1 = P_0 + L$ and $k_0 = L$. After computing both bits (k_1 and k_0) a selection is done using a 2:1 multiplexer.

An N-digit carry select adder will have a total (worst case) delay (T_{dsum} (carry select)) equal to the sum of the 'carry delay' through the first digit (T_{dcout}), the carry select delays through the next (N-1) digits, and the 'sum delay' through the last digit ($T_{sum-digit}$). This is given in Equation (4.22).

$$T_{dsum \text{ (carry select)}} = T_{d-cout} + (N-1)T_{mux} + T_{sum-digit} \quad (4.22)$$

where T_{dcout} is the delay to generate 'K' bit from the BCD inputs for the first digit

T_{mux} is the delay of a 2:1 multiplexer

$T_{sum-digit}$ is the delay of special adder for the last digit

4.5.2.1 Parity Preserving Reversible Carry Select BCD Adder

The design of a carry select BCD adder is same as that for QAD till the ‘L’ bit generation. Figure 4.27 shows the generation of ‘K’ bit or the Decimal C_{out} . The generation of k_1 and k_0 takes the delay of only one Fredkin gate after receiving ‘L’ bit as shown in Figure 4.27. A selection is done by a single FRG After computing both values (k_1 and k_0). An FRG works as a 2:1 multiplexer with ‘A’ input as control input and ‘B’ and ‘C’ as data inputs. So, the additional delay in each digit to generate ‘K’ bit after receiving C_{in} is due to only one FRG.

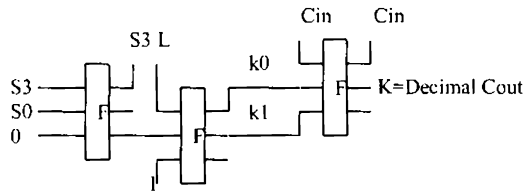


Figure 4.27: Generation of ‘K’ bit using k_0 and k_1

The delay in generation of ‘K’ bit (Decimal C_{out}) for one digit is given in Equation (4.23), where $n=4$.

$$T_{d-cout} = 6 + 2(n-1) \quad (4.23)$$

Special adder implemented using one half adder, two full adders and one XOR gate requires 15 Fredkin gates (3 for half adder, 5 for each full adder, 2 for XOR gate) to generate the BCD sum (d_{3-0}). The Decimal C_{out} or the ‘K’ bit is the last input to be received for the special adder. The ‘K’ input passes through a maximum of 5 Fredkin gates to generate the BCD sum (d_{3-0}). But, C_{in} is received by the special adder along with the ‘K’ bit. The half adder of the special adder generates the carry bit after one Fredkin gate delay on

receiving C_{in} . So, the delay of special adder ($T_{sum-digit}$) is the delay of 6 Fredkin gates.

For an N-digit BCD adder, Decimal C_{out} at N^{th} digit ($K_{(N-1)}$) is generated after a delay equal to the sum of delay of 'K' bit generation for the first digit (T_{dcout}) and the multiplexer delays through the next (N-1) digits. It is given in Equation (4.24).

$$T_{Nd-cout} = T_{dcout} + (N-1)T_{mux} = 6+2(n-1) + (N-1) \quad (4.24)$$

Substituting the delays in Equation (4.25), the total worst case delay ($T_{dsum (carry select)}$) in terms of Fredkin gate delay is

$$T_{d-sum (carry select)} = 6+2(n-1) + (N-1) + 6 \quad (4.25)$$

4.5.3 Hybrid BCD Adder

Hybrid logic for N-digit BCD addition can be used for delay reduction and is shown in Figure 4.28. The N-digit BCD input is divided into m-digit fixed blocks. Each m-digit adder consists of 'm' single digit carry select adders. Carry lookahead logic is included in m-digit blocks to speed up addition. For an m-digit adder, Decimal C_{out} at m^{th} digit (K_{m-1}) can be computed as given in Equation (4.26) using Equation (4.21).

$$K_{m-1} = C_{in} \prod_{k=0}^{m-1} P_k + \sum_{i=0}^{m-1} L_i \left[\prod_{j=i+1}^{m-1} P_j \right] \quad (4.26)$$

where L_i is the 'L' bit of i^{th} digit; P_i is the propagate bit for i^{th} digit

This can be written as

$$K_{m-1} = k_{0(m-1)} C_{in}' + k_{1(m-1)} C_{in} \quad (4.27)$$

where $k_{0(m-1)} = K_{m-1}$ with $C_{in} = 0$; $k_{1(m-1)} = K_{m-1}$ with $C_{in} = 1$

The computations up to the generation of ‘ L_i ’ and ‘ P_i ’ bits at each digit are carried out in parallel for all digits. The delay for ‘ L ’ bit generation is given as

$$T_L = T_{\text{adder}} + T_{6\text{-correction}} \tag{4.28}$$

where T_{adder} is the delay of the 4-bit binary adder, and

$T_{6\text{-correction}}$ is the delay of the 6-correction circuit

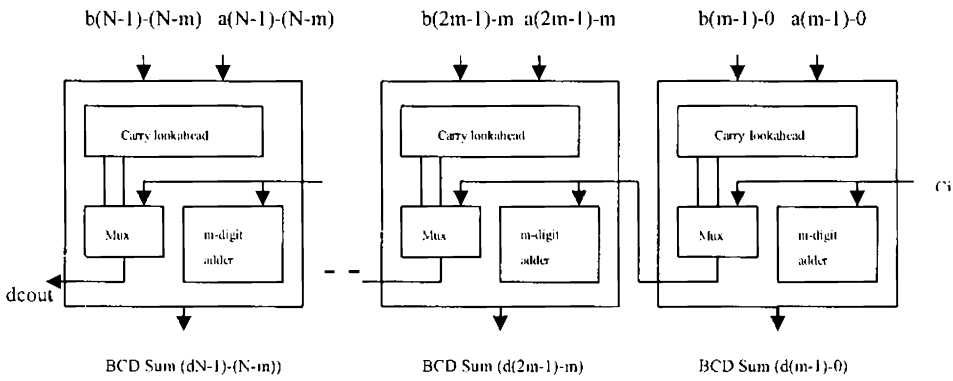


Figure 4.28: Hybrid N-digit Decimal Adder

$k_{0(m-1)}$ and $k_{1(m-1)}$ for an m -digit block are computed using ‘ L_i ’ and ‘ P_i ’ as given in Equation (4.26) after a delay of $T_{kl(m-1)}$. $T_{kl(m-1)}$ is the delay of an m -input AND gate and $(m+1)$ input OR gate. On receiving C_{in} , the Decimal C_{out} at m^{th} digit (K_{m-1}) is generated after an additional delay of a 2:1 multiplexer (T_{mux}), and is given as

$$T_{m\text{-dcout}} = T_L + T_{kl(m-1)} + T_{mux} \tag{4.29}$$

The total (worst case) delay of an N -digit hybrid BCD adder ($T_{\text{dsum (hybrid)}}$) with fixed size carry look ahead block is the sum of the ‘carry delay’ through the first m -digit lookahead adder block ($T_{m\text{-dcout}}$), the carry select

delays through the intermediate blocks, and the ‘sum delay’ through the last m-digit block ($T_{\text{sum-m-digit}}$). This is given in Equation (4.30).

$$T_{\text{dsum (hybrid)}} = T_{\text{m-dcout}} + [(N/m)-2] T_{\text{mux}} + T_{\text{sum-m-digit}} \quad (4.30)$$

$$\text{where } T_{\text{sum-m-digit}} = m + T_{\text{sum-digit}} \quad (4.31)$$

While the total (worst case) delay of an N-digit conventional BCD adder ($T_{\text{dsum (conventional)}}$) given in equation (4.32) is the sum of ‘N’ times the ‘carry delay’ through one digit and the ‘sum delay’ through the last digit ($T_{\text{sum-digit}}$).

$$T_{\text{dsum (conventional)}} = NT_{\text{dcout}} + T_{\text{sum-digit}} \quad (4.32)$$

4.5.3.1 Hybrid Reversible BCD Adder

The total (worst case) delay of an N-digit hybrid BCD adder with fixed size carry look ahead block is the given in Equation (4.30). The first term in equation (4.30) requires a delay as given in Equation (4.29). In reversible implementation using Fredkin gates the delay to generate all ‘ L_i ’ bits is T_L (given in Equation (4.16)) with $n=4$. All ‘ P_i ’ will be available when the generation of ‘ L_i ’ gets over. $T_{k_{l(m-1)}}$ is the delay of an m-input AND gate and (m+1) input OR gate. A 2-input AND or a 2-input OR can be implemented by a single Fredkin gate. Higher order AND and OR gates can be constructed using Fredkin gates arranged in a binary tree. An m-input AND gate or an m-input OR gate requires (m-1) Fredkin gates. An input passes through a maximum of $\lceil \log_2 m \rceil$ Fredkin gates [B. Parhami, 2006]. On receiving C_{in} , the selection of $k_{l(m-1)}$ or $k_{0(m-1)}$ requires one more Fredkin delay for each m-digit block. Hence the ‘carry delay’ through the first m-digit lookahead adder block is

$$T_{\text{m-dcout}} = 10 + \lceil \log_2 m \rceil + \lceil \log_2 (m+1) \rceil + 1 \quad (4.33)$$

The delay for carry select for intermediate blocks is $\frac{N}{m} - 2$.

The sum delay through the last m -digit block is $(m+6)$.

Total delay in generating N -digit BCD sum is given as

$$T_{d\text{-sum (hybrid)}} = 11 + \lceil \log_2 m \rceil + \lceil \log_2 (m+1) \rceil + \frac{N}{m} - 2 + m + 6 \quad (4.34)$$

However, the assumption $\lceil \log_2 m \rceil = m/2$ is valid for the small block sizes applicable to carry look ahead adder designs. Thus, (4.34) can be written as

$$T_{d\text{-sum (hybrid)}} = 15 + 2m + \frac{N}{m} \quad (4.35)$$

Minimizing $T_{d\text{-sum (hybrid)}}$ with respect to block size m

$$m_{\text{opt}} = \sqrt{0.5N} \quad (4.36)$$

Substituting (4.36) into (4.35) gives the shortest delay for a fixed block size hybrid BCD reversible adder.

$$T_{d\text{-sum (hybrid)}} = 15 + \sqrt{8N} \quad (4.37)$$

4.5.4 Toffoli Gate Implementation

In this section, Toffoli Gate (TG) reversible implementations of conventional and fast decimal adders are presented. Implementations using TG and that using Fredkin Gates (FRG) are compared based on quantum cost (QC), number of gates, garbage count and delay. Quantum cost analysis is done to compare the equivalent number of two-qubit quantum gates required for the implementation.

Figure 4.29 and Figure 4.30 show the implementation of a half adder and a full adder using Toffoli gates simulated using RC viewer. The

implementation of half adder makes use of 2 Toffoli gates: 3-input Toffoli (T3) and 2-input Toffoli (T2) with one garbage output. Full adder implementation makes use of 4 Toffoli gates (two T3, two T2) and results in 2 garbage outputs (This is same as Figure 4.11). These are optimum solutions in terms of number of garbage outputs. The RC Viewer gave the quantum cost of the full and half adders as 8 and 4 respectively.



Figure 4.29: Half Adder

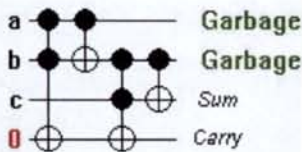


Figure 4.30: Full Adder

A 4-bit binary adder for the conventional BCD adder realized using 4 full adders is shown in Figure 4.31. The least significant bit requires a path delay of three TGs to generate C_0 (carry) from the addends. Carry ripples through the subsequent full adders with a path delay of one TG per bit. This is because the first two TGs of all full adders work in parallel in an n-bit binary adder. The 'Sum' is generated after one more TG delay after generating Carry. The delay to generate 'Sum' in the n-bit binary adder is given in Equation (4.38).

$$T_{\text{sum-ripple(conventional)}} = 4+(n-1) \quad (4.38)$$

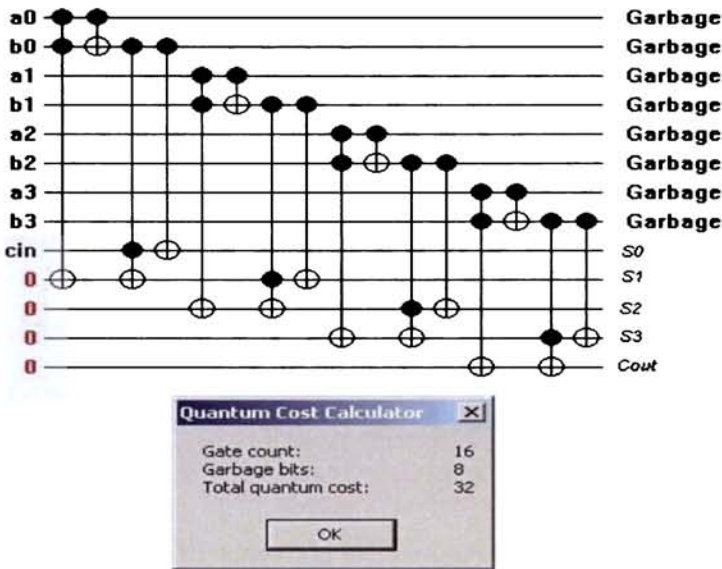


Figure 4.31. 4-bit Binary Adder using Toffoli Gates

For a BCD adder this delay is the delay with $n=4$ for each digit. The implementation gives a gate count of 16, with 8 garbage outputs at a quantum cost of 32. The Toffoli reversible implementation of the 6-correction circuit is shown in Figure 4.32. The implementation requires 4 TGs to generate the ‘L’ output, with 1 garbage output at a quantum cost of 12. This circuit takes only 1 more delay after generating the ‘S₃’ to generate the ‘L’ bit and is given in Equation (4.39).

$$T_{L(\text{conventional})} = 5 + (n-1) = 8 \text{ (with } n=4) \tag{4.39}$$

Special adder shown in Figure 4-20 requires 8 TGs to generate the BCD_{sum} , d_{sum} . The first T2 is used to duplicate ‘L’ bit or Decimal C_{out} , d_{cout} . So, the total delay in terms of one TG delay for generation of d_{cout} for an N-digit conventional BCD adder is given in Equation (4.40).

$$T_{dcout(\text{conventional})} = 9N \tag{4.40}$$

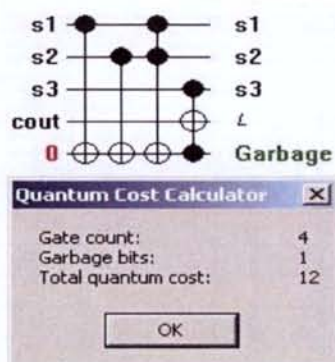


Figure 4.32: 6-Correction Circuit using Toffoli Gates

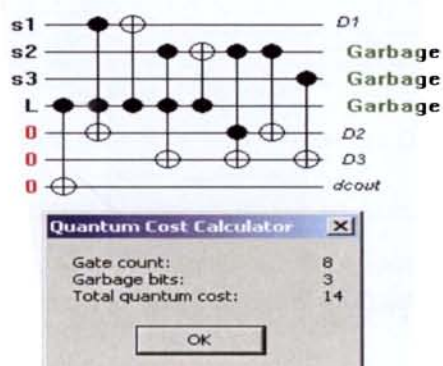


Figure 4.33: Modified Special Adder of Conventional BCD Adder

The modified special adder gives an additional delay of 6 TGs to generate BCD_{sum} . Figure 4.34 shows the schematic of the reversible circuit for the conventional BCD adder implemented using Toffoli gates given by the RC Viewer. The circuit uses 28 gates and results in 12 garbage outputs. The quantum cost of the implementation is 58. The total delay for generating the BCD_{sum} , d_{sum} from the inputs in terms of TG delay for N-digit BCD addition is given in Equation (4.41).

$$T_{d-sum(Conventional)} = 6 + 9N \quad (4.41)$$

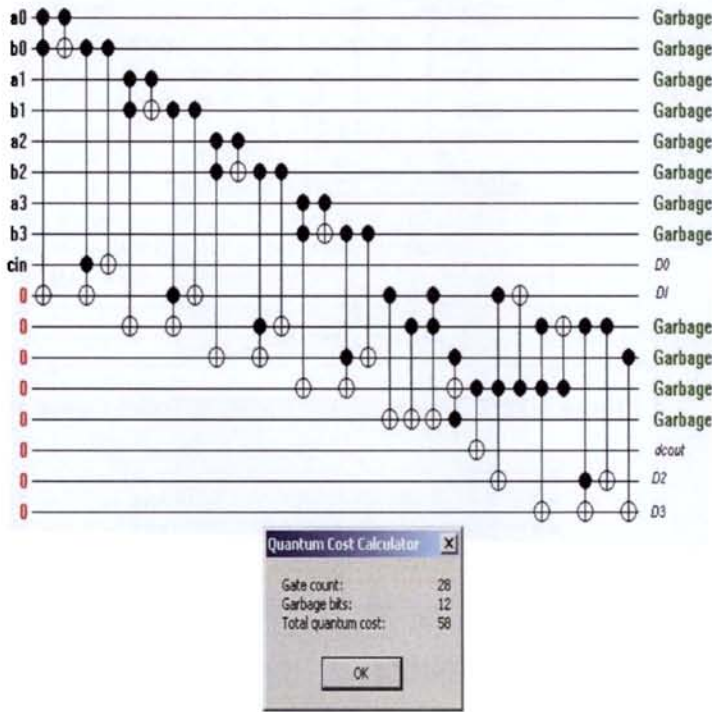


Figure 4.34: Toffoli Gate implementation of Conventional BCD Adder

Reversible implementation of QAD is also done using TGs. A 4-bit binary adder realized using one half adder and 3 full adders is shown in Figure 4.35. The least significant bit requires a path delay of two TGs only. The carry ripples through the subsequent full adders with a path delay of one TG per bit. This is because the first two TGs of all full adders work in parallel with the least significant bit half adder. ‘Sum’ is generated after one more TG delay subsequent to generating the final carry. The delay to generate ‘Sum’ in the 4-bit binary adder in QAD is given in Equation (4.42).

$$T_{\text{sum-rippleQAD}} = 3+(n-1) = 6 \text{ (with } n=4\text{)} \quad (4.42)$$

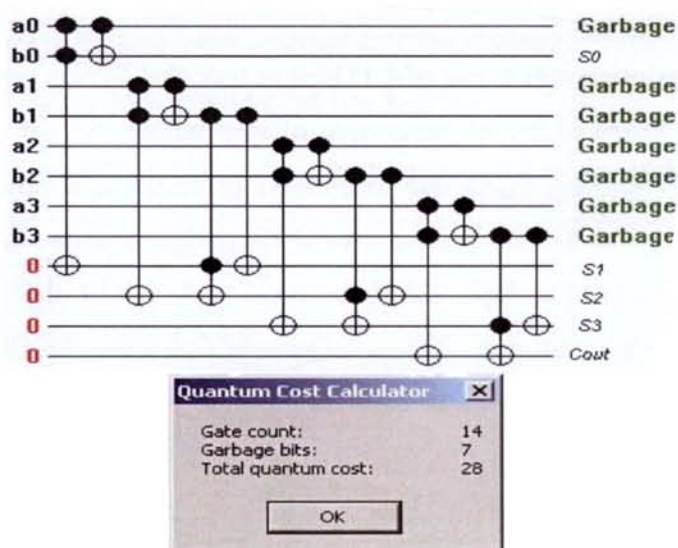


Figure 4.35: 4-bit Binary Adder for QAD

The delay to generate 'L' bit from the BCD inputs in a QAD adder is given in Equation (4.43). The circuit takes only 1 more delay after generating the 'S₃' to generate the 'L' bit as in conventional BCD adder.

$$T_{L(QAD)} = 4 + (n-1) = 7 \text{ (with } n=4\text{)} \quad (4.43)$$

Figure 4.36 shows the reversible implementation for generating Decimal C_{out} (d_{cout}) or 'K' bit in a QAD.

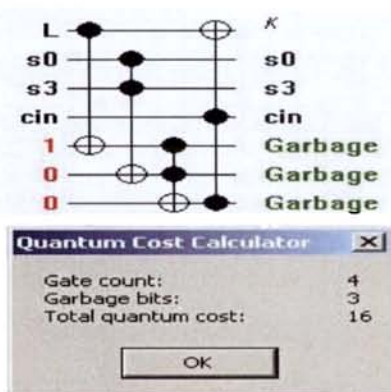


Figure 4.36: K-bit Generation using Toffoli Gates

The design makes use of 4 TGs. It is seen that S_3S_0 is generated after 3 TG delays followed by the generation of sum output 'S'. So, the additional delay after receiving C_{in} is due to one TG in each stage. Special adder for QAD requires 12 TGs to generate the $BCD_{sum} d_{3-0}$ as shown in Figure 4.37. The first T2 is used to duplicate 'K' bit for d_{cout} . For an N-digit BCD addition the delay for generation of K-bit or Decimal C_{out} from the BCD inputs for QAD adder is as given in Equation (4.44).

$$T_{d-cout(QAD)} = 7+3+1+N = 11+N \quad (4.44)$$

The ' C_{in} ' input or 'K' bit passes through a maximum of 7 TGs to generate the BCD sum d_{3-0} . So, the special adder gives an additional delay of 7 gates. The total delay for generating the BCD_{sum}, d_{sum} from the inputs in terms of TG delay is

$$T_{d-sum(QAD)} = 11+7+ N = 18+N \quad (4.45)$$

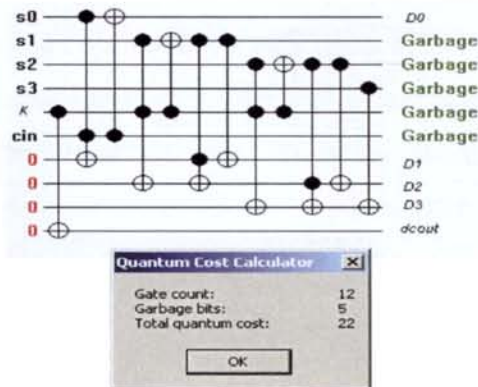


Figure 4.37: Toffoli implementation of Special Adder for QAD

Figure 4.38 shows the schematic of the reversible circuit for the QAD implemented using TGs given by the RC Viewer.

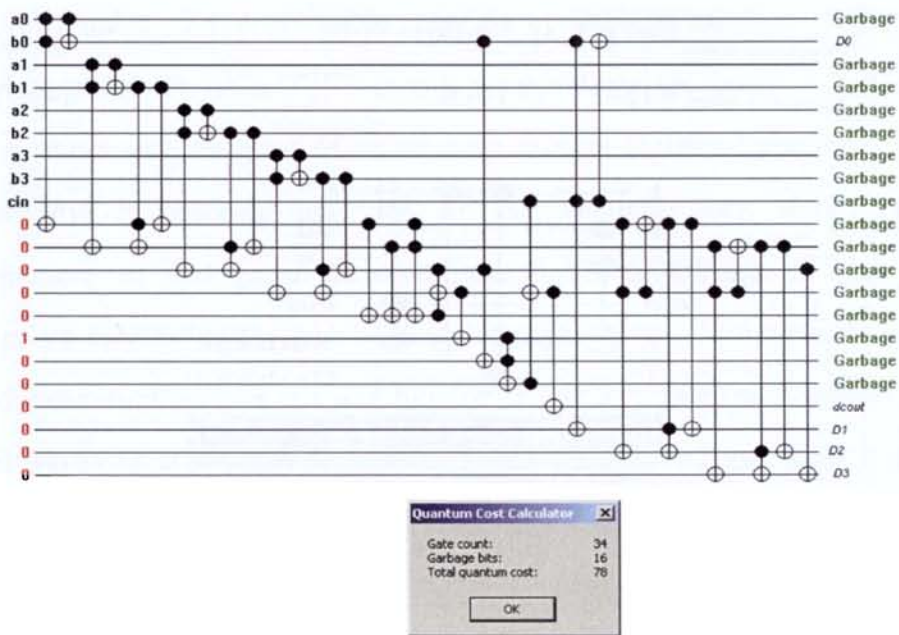


Figure 4.38: Toffoli Gate Implementation of QAD

The circuit uses 33 gates and results in 16 garbage outputs. The quantum cost of the implementation is 81.

Reversible implementation of carry select BCD adder differs from QAD implementation only in the generation of 'K' bit from 'L' bit and C_{in} . Figure 4.39 shows the generation of K-bit using k_1 and k_0 for a carry select BCD adder. The generation of k_1 and k_0 takes the delay of four TGs after receiving 'L' bit as shown in Figure 4.39. 'K' bit is computed after a delay of one TG after receiving C_{in} . So, the additional delay in each stage to generate 'K' bit on receiving C_{in} is due to only one TG. One more T2 is used to

duplicate 'K' bit for d_{cout} . Now, the delay gets modified as in Equations (4.46 and 4.47).

$$T_{d\text{-cout}(\text{carry select})} = 7+4+1+N = 12+N \quad (4.46)$$

$$T_{d\text{-sum}(\text{carry select})} = 12+7 + N= 19+N \quad (4.47)$$

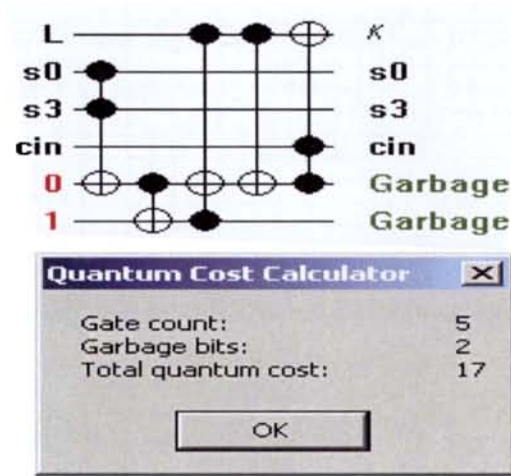


Figure 4.39: K-bit generation of Carry select BCD Adder

4.6 New reversible RPS gate

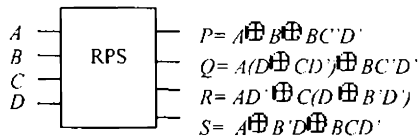
This part of the chapter presents a new fully reversible RPS gate and a new partially reversible RPS gate used for the design of reversible conventional BCD adder.

4.6.1 Fully Reversible RPS Gate

A number of 4×4 reversible gates are available in literature. For a 4-input/4-output truth table, $2^{(16 \times 4)}$ output combinations are possible. Out of these a one-to-one mapping of inputs with outputs (essential condition for reversibility) is observed only for 16! combinations only. Hence, 16! combinations of 4×4 gates are reversible. But it is difficult to find an appropriate application for each of these reversible gates. This research uses one of these combinations as a 4×4 fully reversible RPS gate that functions as a 4-bit Binary to BCD converter. The RPS gate is shown in Figure 4.40. It can be verified from the truth table given in Table 4.2 that the input pattern corresponding to a particular output pattern has a unique mapping. RPS gates can be considered as universal gates since any Boolean function can be realized by using the combination of RPS gates. The fully reversible RPS gate is an optimized gate for the implementation of a reversible 4-bit binary to BCD converter. It can also be used as the offset correcting circuit after the addition of 2 single digit BCD numbers. It works as the hardware equivalent of the logic function, 'decimal adjust after BCD addition'.

Table 4.2: Truth Table of Fully Reversible RPS Gate

Inputs				Outputs			
A	B	C	D	P	Q	R	S
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	1	1	0	1
1	0	1	1	1	1	1	0
1	1	0	0	1	1	1	1
1	1	0	1	0	1	0	1
1	1	1	0	0	1	1	0
1	1	1	1	0	1	1	1

**Figure 4.40:** 4×4 Fully reversible RPS gate

4.6.2 4-bit Binary to BCD Converter using RPS Gate

For a 4-bit Binary to BCD converter, let the binary input be B_3 - B_0 and the corresponding BCD outputs are D_3 - D_0 with D_4 as the carry to the next higher digit. Hence 4-bit Binary to BCD conversion is a logic function with 4 inputs and 5 outputs as shown in Table 4.3. To make it reversible the circuit requires at least 5 inputs since there are 5 outputs. But it can be seen from Table 4.3 that D_0 output is same as B_0 input. Hence, only the remaining 4 outputs need to be generated. Now, Table 4.3 becomes a 3-input/4-output

truth table, and hence there is a possibility that it can be realized using a 4×4 reversible gate with one of the inputs as constant.

The fully reversible RPS gate with A input as '0', B as B_3 , C as B_2 and D as B_1 , gives a 4-bit output pattern at P , Q , R and S , same as the $D_4 \dots D_1$ bits of Table 4.3. Thus, a single fully reversible RPS gate functions as a 4-bit Binary to BCD converter as shown in Figure 4.41. It can also be noted that the garbage count is zero.

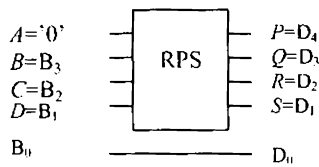


Figure 4.41: 4-bit Binary to BCD converter using Fully Reversible RPS gate

Table 4.3: Truth Table of 4-Bit Binary to BCD Converter

Inputs				Outputs				
B_3	B_2	B_1	B_0	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1
0	1	1	0	0	0	1	1	0
0	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	0	0	1	0	1	0	0	1
1	0	1	0	1	0	0	0	0
1	0	1	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0
1	1	0	1	1	0	0	1	1
1	1	1	0	1	0	1	0	0
1	1	1	1	1	0	1	0	1

4.6.3 4-bit Binary to BCD Converter using other existing Reversible Gates

4-bit Binary to BCD converter can be designed using other universal reversible gates such as Toffoli gates, Fredkin gates or HNG gates. But such implementations require more number of reversible gates, and produce more garbage outputs. The converter designed using RPS gate requires only one reversible gate without any garbage outputs. Figure 4.42 shows the reversible implementation of a 4-bit Binary to BCD converter using a suitable combination of these reversible gates. The implementation requires 5 gates with a garbage count of 4 at 5 levels.

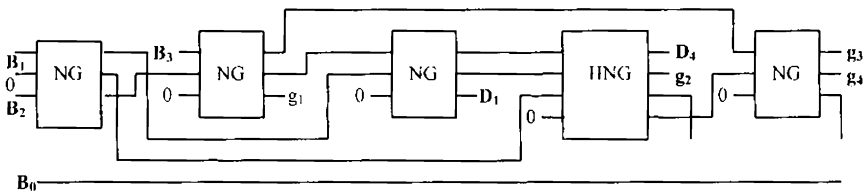


Figure 4.42: 4-bit Binary to BCD converter using HNG and NG

4.6.4 4-bit Binary to BCD converter using HNG gates

Figure 4.43 shows an implementation of 4-bit Binary to BCD converter using HNG gates. The implementation requires 5 gates with a garbage count of 8. The critical path delay is that of 5 reversible gates. This implementation makes use of only one type of gate, making it suitable for regular implementation compared to the implementation in Figure 4.42. HNG gate is most suitable for reversible full adder implementation, but it is not apt for the implementation of 4-bit Binary to BCD converter, since it requires more number of gates, garbage and levels compared to RPS implementation.

The logical complexity is also more in HNG gate implementation for 4-bit binary to BCD converter. The combination of HNG and RPS can be used in the design of a BCD adder that requires full adders and Binary to BCD converters.

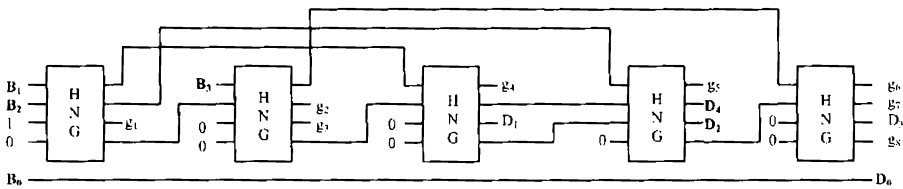


Figure 4.43: 4-bit Binary to BCD converter using only HNG gates

4.6.5 Partially Reversible RPS Gate

The concept of partial reversible gate was first introduced in [H. Thapliyal *et al.*, 2007] for the implementation of BCD to Excess 3 converter. The new partially reversible RPS gate can be used for partial reversible implementation of a BCD adder. The new partially reversible RPS gate satisfies the reversibility criteria for some specific inputs such as BCD inputs. This can be effectively utilized in BCD arithmetic circuits. Such gates can minimize the logical complexity, and sometimes gate and garbage count, while designing the reversible BCD arithmetic circuits.

The partially reversible RPS gate is shown in Figure 4.44, and its truth table in Table 4.4. This gate is reversible only in Part 1 that consists of all valid BCD inputs as shown in Table 4.4. The outputs are ‘don’t cares’ for Part 2 since the remaining combinations are not valid BCD inputs. So, this gate functions as a circuit that accepts only BCD inputs.

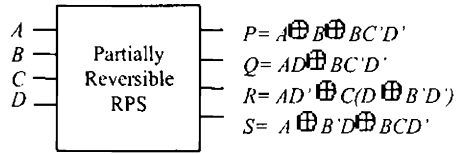


Figure 4.44: 4 × 4 Partially Reversible RPS Gate

Like fully reversible RPS gate, partially reversible RPS gate can also function as a reversible 4-bit Binary to BCD converter. Let the binary inputs be B₃-B₀ and the corresponding BCD outputs are D₃-D₀ with D₄ as the carry to the next higher digit as shown in Table 4.3.

Table 4.4: Truth Table of the Partial Reversible RPS Gate

Inputs				Outputs			
A	B	C	D	P	Q	R	S
PART 1							
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
PART 2							
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

For a partially reversible RPS gate if A input is '0', B is B_3 , C is B_2 and D is B_1 , then, a 4-bit output pattern at P , Q , R and S , is the same as the $D_4 \dots D_1$ bits of Table 4.3. D_0 output is same as B_0 input. Hence, only the remaining 4 outputs need to be generated. Thus a single partially reversible RPS gate functions as a 4-bit binary to BCD converter. It can also be noted that the garbage count is zero.

The partially reversible RPS gate can also function as 'decimal adjust after BCD addition'. The 5-bit binary result obtained after the addition of 2 single digit BCD numbers is to be adjusted for its BCD equivalent. Part 1 of Table 4.4 contains the most significant 4 bits of the 5-bit binary result. The maximum possible binary value after the addition of 2 BCD numbers is '10010'. Excluding the least significant bit makes it '1001'. Hence Part 1 of Table 4.4 realizes the most significant 4-bits of an 'offset correction' function. Since the remaining combinations never occur in the most significant 4 bits of the binary result, Part 2 of Table 4.4 are 'don't cares'. Thus, a partially reversible RPS gate can be used to design a 'Decimal adjust' after the addition of BCD numbers. The partially reversible RPS gate has lesser logical complexity compared to the fully reversible RPS.

4.6.6 Reversible Implementations of BCD Adder

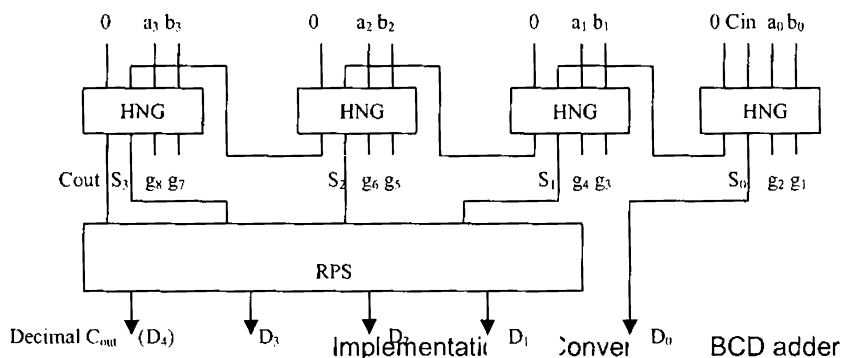
A BCD adder accepts 2 BCD inputs (4-bits each) and a carry input, C_{in} and produces 4-bit BCD Sum and a Decimal C_{out} . So the total number of inputs to the circuit is 9 and the number of outputs is 5. In the truth table of a BCD adder, there are 19 input combinations for which the output is same '10000'. At least five garbage bits are required to make a unique output

combination for each input combination. So the optimum realization of a reversible BCD adder will produce at the least 5 garbage outputs.

4.6.6.1 BCD adder using HNG-RPS Gates

The reversible implementation of BCD adder using HNG-RPS gates use 5 reversible gates and produces 8 garbage outputs. The reversible implementation of the 4-bit adder used in conventional BCD adder makes use of 4 HNG full adders producing 8 garbage outputs. A number of reversible full adders are available in literature, but the implementation of a full adder using HNG takes least number of gates with less complexity and produces least number of garbage outputs. This gate is preferred over TSG due to the lesser number of logical computations involved in it. The ‘six-correction’ circuit and the ‘4-bit binary adder’ are replaced by a single fully reversible RPS gate. The complete circuit of the reversible BCD adder is shown in Figure 4.45. The use of fully reversible RPS gate optimizes the offset correction part of the BCD adder architecture. The garbage count is also reduced to 8, which is very near to the optimum garbage count of 5 for a BCD adder. The delay for one stage of the new reversible BCD adder is that of 4 HNGs and one fully reversible RPS gate. This implementation makes use of only 2 types of gates, making it more suitable for regular implementation.

Partially reversible implementation of a BCD adder is done by replacing the fully reversible RPS gate in Figure 4.45 with partially reversible RPS gate. This reduces the computational complexity even though the number of reversible gates, the garbage count and number of levels of delay of the conventional BCD adder remain unchanged.



4.6.6.2 BCD adder using RPS gates

Figure 4.46 shows the implementation of a BCD adder using only fully reversible RPS gates. The implementation makes use of 9 RPS gates with 12 garbage outputs. The delay for one stage is that of 9 fully reversible RPS gates. For an N-digit BCD adder the delay is 9N. This implementation makes use of only RPS gates, making it suitable for regular implementation.

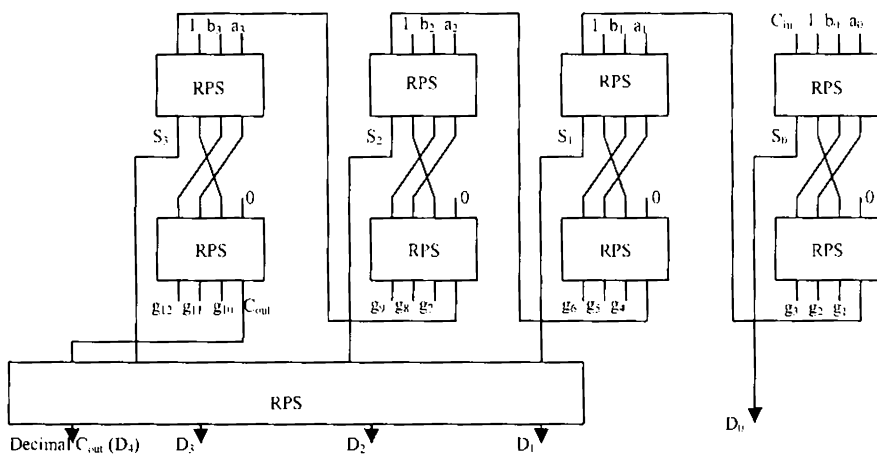


Figure 4.46: Conventional BCD adder using RPS gates

4.6.6.3 BCD adder using HNG gates

BCD adder is implemented using only 10 HNG gates with 19 garbage outputs as shown in Figure 4.47. The delay for one stage of Decimal C_{out} and BCD Sum is 7 and 10 HNG gates respectively. For an N-digit BCD adder the delay is $7N$ for Decimal C_{out} and $7N+3$ for final BCD Sum. Though this implementation uses only HNG gates, but has more gates and garbage compared to the implementation using RPS gates.

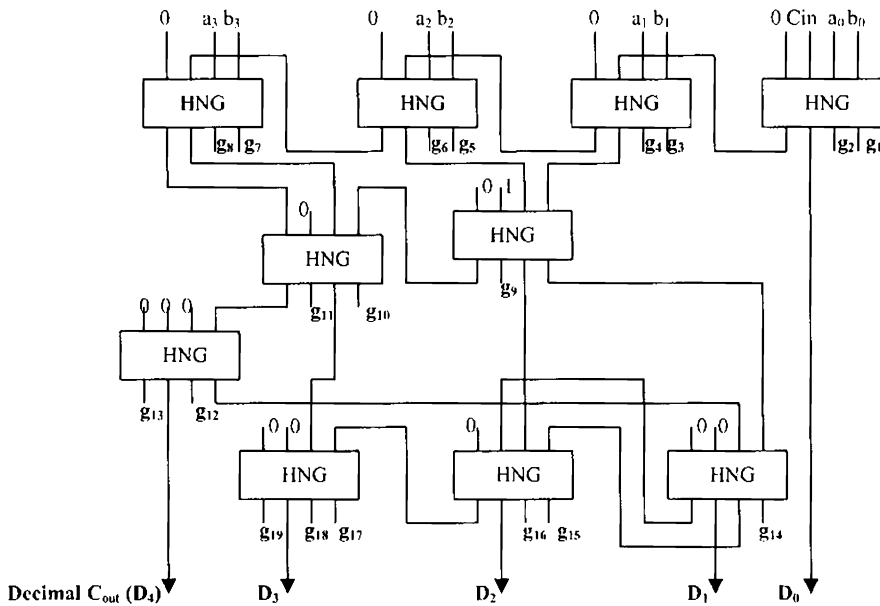


Figure 4.47: Conventional BCD adder using HNG

4.7 Reversible Error Correcting Code Generation and Detection

Error correcting codes are traditionally used to battle the corruption of transmitted data by channel noise. The encoded data or code words are sent

through the channel, and decoded at the receiving end. During decoding the errors are detected and corrected if the error is within the allowed, correctable, range. This range depends on the extra information, parity bits, added during encoding. Single Error Correcting (SEC) codes are generally used for this purpose. There are many ways to construct SEC codes, and one of the most commonly used codes is the Hamming Code. As power has become a first-order design consideration, researchers have begun looking at techniques to reduce power consumption in error correcting code generation and detection circuitry. Fault detection can be done using parity-preserving reversible logic gates. The feasibility of the parity-preserving approach in the design of reversible logic circuits was demonstrated by [B. Parhami, 2006] with examples of adder circuits. Parity checking is one of the oldest as well as one of the most widely used methods for error detection in digital systems. It's most common use is for detecting errors in the storage or transmission of information.

In this part of the chapter, a new reversible 4×4 HC gate (HCG) is presented for implementing hamming error coding and detection circuits. The parity of the outputs matches with that of the inputs in this gate. This can be used to generate the parity preserved / fault tolerant hamming code along with other parity preserving reversible logic gates. Parity preserving characteristic of such gates allows the detection of single fault at the circuit's primary outputs in reversible logic design.

HC gate (HCG) is shown in Figure 4.48. Table 4.5 shows the corresponding truth table. It is obvious from the truth table that the input pattern corresponding to a particular output pattern can be uniquely determined. The reversible Parity Preserving HC gate (PPHCG) is shown in

Figure 4.49. Table 4.6 shows the corresponding truth table. It can be verified from the truth table that the outputs preserve the input parity.

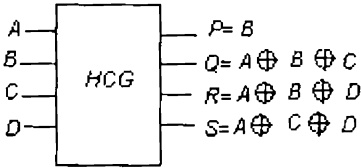


Figure 4.48: Reversible 4 x 4 HCG

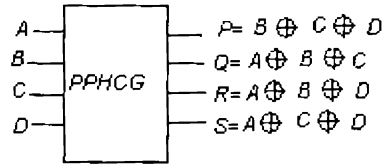


Figure 4.49: Reversible 4 x 4 PPHCG

Table 4.5: Truth Table of the 4 X 4 HCG

Inputs				Outputs			
A	B	C	D	P	Q	R	S
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	1
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	1	1	1	0
0	1	0	1	1	1	0	1
0	1	1	0	1	0	1	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	1	1
1	0	0	1	0	1	0	0
1	0	1	0	0	0	1	0
1	0	1	1	0	0	0	1
1	1	0	0	1	0	0	1
1	1	0	1	1	0	1	0
1	1	1	0	1	1	0	0
1	1	1	1	1	1	1	1

Table 4.6: Truth Table of the PPHC Gate

Inputs				Outputs			
A	B	C	D	P	Q	R	S
0	0	0	0	0	0	0	0
0	0	0	1	1	0	1	1
0	0	1	0	1	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	1	1	1	0
0	1	0	1	0	1	0	1
0	1	1	0	0	0	1	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	1	1
1	0	0	1	1	1	0	0
1	0	1	0	1	0	1	0
1	0	1	1	0	0	0	1
1	1	0	0	1	0	0	1
1	1	0	1	0	0	1	0
1	1	1	0	0	1	0	0
1	1	1	1	1	1	1	1

Figure 4.50 shows (7, 4) Hamming code generator designed using 4x4 HCG and three FGs. The three bits to be added are three even parity bits (P), where the parity bit is computed on different subsets of the message bits using Equations (4.48 – 4.50). It is seen that the circuit requires 4 gates at 2 levels and generates the 7-bit hamming code (H₇ to H₁) without any garbage outputs.

$$P_1 = D_0 \oplus D_1 \oplus D_3 \tag{4.48}$$

$$P_2 = D_0 \oplus D_2 \oplus D_3 \tag{4.49}$$

$$P_3 = D_1 \oplus D_2 \oplus D_3 \tag{4.50}$$

The reversible hamming code generator designed using F2G and FG gates is shown in Figure 4.51. On comparing the two implementations it is evident that the implementation using F2G and FG requires 6 reversible gates at 4 levels while the implementation using HCG and FG requires only 4 gates at 2 levels. However both implementations attain minimum number of garbage outputs. The types of gates used in these two implementations are not parity preserving gates except for F2G, and hence are not fault tolerant implementations.

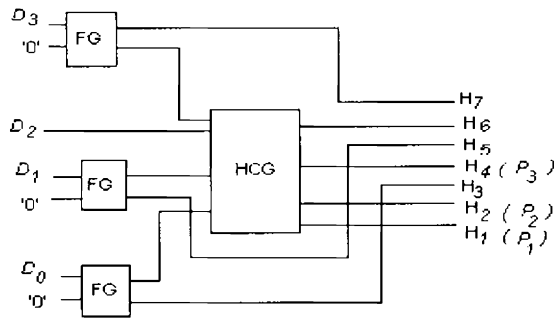


Figure 4.50: Reversible (7, 4) Hamming code generator using HCG

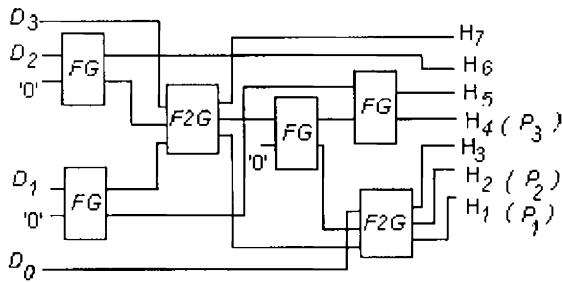


Figure 4.51: Reversible (7, 4) Hamming code generator using F2G

Figure 4.52 shows the implementation of reversible hamming code generator circuit designed using parity preserving gates. The implementation makes use of 5 gates at 2 levels and produces 5 garbage outputs. The input and output parity will be the same since the design is done using parity preserving gates. A single fault can be detected by checking the parity of the inputs and outputs at each level. The design uses 2 types of reversible gates. Figure 4.53 shows the implementation using six F2Gs at 4 levels with 4 garbage outputs. The implementation uses a single type of reversible gate, and produces less number of garbage outputs. But this results in increased delay, and makes use of more number of gates.

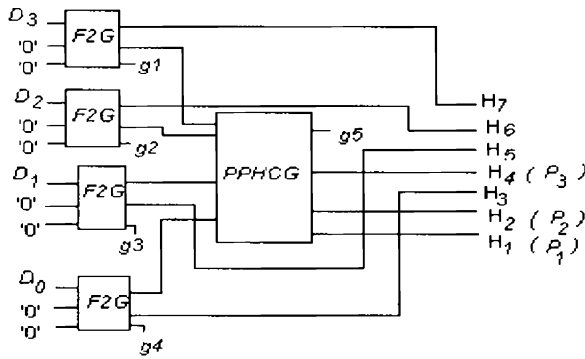


Figure 4.52: (7, 4) Hamming Code Generator using parity preserving gates

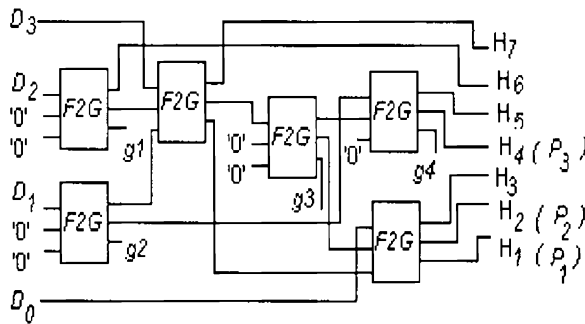


Figure 4.53: (7, 4) Hamming Code Generator using F2G

Figure 4.54 shows the (7, 4) hamming code error detector designed using 4×4 HCG and three FGs. It is seen that the circuit requires 4 gates at 2 levels and generates the check bits (C_3 to C_1) with 4 garbage outputs. Check bits are computed on different subsets of the hamming code bits using the Equations (4.51 – 4.53).

$$C_1 = H_1 \oplus H_3 \oplus H_5 \oplus H_7 \quad (4.51)$$

$$C_2 = H_2 \oplus H_3 \oplus H_6 \oplus H_7 \quad (4.52)$$

$$C_3 = H_4 \oplus H_5 \oplus H_6 \oplus H_7 \quad (4.53)$$

If all the check bits are zeros it indicates a 'no error condition', otherwise it indicates the position of error. This implementation is not done using parity preserving gates, and hence is not capable of detecting any faults in the circuit.

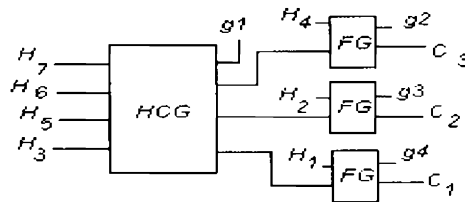


Figure 4.54: Reversible (7, 4) Hamming code error detector using HCG

Figure 4.55 shows the implementation of reversible hamming code error detector designed using parity preserving gates. The implementation uses 5 gates at 3 levels, and produces 6 garbage outputs. The advantage of this implementation is the use of only one type of reversible gate; but it results in increased delay and makes use of more number of gates. Since the implementation uses only one type of reversible gate, the design is more suitable for VLSI design.

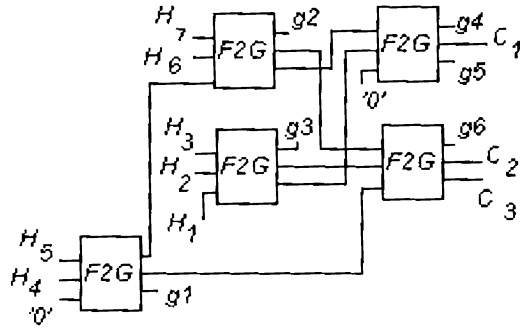


Figure 4.55: Reversible (7, 4) Hamming code error detector using F2G

4.8 Summary

In this chapter a reversible BCD adder implementation for a conventional BCD adder is presented first. The architecture is specially designed to make it suitable for reversible logic implementation. A modified version of decimal addition using reversible gates which results in further reduction in number of gates and garbage outputs with a fan-out of 1 is designed. It is demonstrated that the proposed design is highly optimized in terms of number of reversible gates and garbage outputs. The design makes use of 3 types of reversible gates. For VLSI implementation, circuits using only one type of standard reversible gate as the basic building block are adopted. This research also presents reversible implementations of three different fast decimal adders. An implementation for a reversible fault tolerant logic, using Fredkin gates, for quick addition of decimals (QAD), suitable for fast multi-digit BCD addition is presented. The performance comparison of reversible fast BCD adders with conventional BCD adder is also presented. This chapter also describes reversible implementations of several BCD adders using only Toffoli gate. Toffoli gates are also suitable for implementations of

Reed Muller expressions. VLSI implementations using only one type of modular building block can decrease system design and manufacturing cost.

This thesis also presents two novel 4×4 reversible logic gates : fully reversible RPS gate and partially reversible RPS gate. The new reversible 4-bit Binary to BCD converter circuit requires only one RPS gate and does not have any garbage outputs. A reversible BCD adder is designed in which the ‘6-correction circuit’ and the ‘final 4-bit binary adder’ are replaced by a single RPS gate. The performance comparisons of different reversible BCD adder designs are given in Section 6.4.

Different implementations for the reversible (7, 4) hamming error code generation and detection circuits are presented as well. It is demonstrated that the design using 4×4 HCG is highly optimized in terms of number of reversible gates and/or garbage outputs. This approach also provides a way of incorporating fault tolerance into reversible circuits without much extra design effort and with modest hardware overhead. The comparisons of different reversible hamming code generation and detection circuit designs are given in Section 6.4.

Chapter 5

Logic Synthesis using Multiplexers

This chapter presents an approach to obtain reduced hardware and/or delay for logic functions using multiplexer universal logic modules. Replication of single control line multiplexer is used as the only design unit for defining any logic function specified by minterms. A novel algorithm is formulated that does exhaustive branching to reduce the number of levels and/or modules required for implementing a logic function. The algorithm identifies single or double variable function at the control input of a multiplexer that will result in reduced number of levels and/or hardware. This approach can be adopted for the design of a more regular BCD digit multiplier.

5.1 Delay-Reduced Combinational Logic Synthesis using Multiplexers

The use of multiplexer as Universal Logic Module (ULM) for realization of logic functions has already been explored by researchers. An algorithm was developed by [A. Pal, 1986] to obtain single multiplexer realization of logic functions with a minimal size multiplexer. The limitation of this approach is that the size of the module changes with changes in function to be realized due to a non-modular implementation. An iterative method for cascade realization using 1-control line multiplexer was presented by [R. K. Gorai and A. Pal, 1990]. This method terminates if the function is not cascade realizable. Further, as the number of variables increases, the number of levels also increases drastically, resulting in increased delay. A programmed algorithm that implements logic functions using tree structure was presented by [A.E.A. Almaini, J.F. Miller and L Xu, 1992]. This algorithm does not explore all the possible branching options of the tree structure, and hence the delay of the circuit synthesized may not be minimal. Also, it does not guarantee the global optimality in all the cases. Genetic programming approach to synthesize logic functions using multiplexers was presented by [A.H. Aguirre, C.A.C. Coello and B.P. Buckles, 1999] and [A.H. Aguirre and C.A.C. Coello, 2004]. Even though the number of multiplexers used in the delivered circuit had an improvement over standard implementation, the circuit was not minimal or optimal.

In this chapter, a novel tree-structured exhaustive branching network using 1-control line multiplexer is designed. It implements logic functions described by minterms that reduces delay and/or hardware. A tree network is very suitable for VLSI realization because of the uniform interconnection structure and the repeated use of identical modules. A logic function with n variables can be implemented using 2^n-1 , 1-control line multiplexers in n levels in standard implementation. Any implementation using less than 2^n-1 number of modules and/or lesser number of levels can be considered as an improvement in cost and/or speed. VLSI implementations using only one type of modular building blocks can decrease system design and manufacturing cost. A multiplexer realization is a natural choice from the viewpoint of cost and speed.

5.2 N-ary Exhaustive Branching Technique

For a given number of input variables n , there is 2^{2^n} well defined number of functions available. Standard implementation of a tree network requires n levels to implement these functions. Almaini presented a programmed algorithm to reduce the complexity of the network in terms of number of modules and levels. In his approach 1's, 0's, x_i^* (where x_i^* is a variable x_i or its complement, $1 \leq i \leq n$) or an n -variable function can be given as a data input. The control inputs accept only variables. In this research, the performance is further improved by an exhaustive branching technique using n -variable functions at control input. Since functions are also given to the control input, the utilization of all branching options is possible. This

decreases the number of levels, and hence reduces delay for a logic function implementation using multiplexers.

The first level (output stage) will have a single multiplexer module, the second level will have a maximum of $(2^c + c)$ multiplexer modules, where c is the number of control inputs ($c = 1$ in this case). In general, the maximum number of modules in a level can be expressed as $(2^c + c)^{(L-1)}$ where L indicates the number of levels. The maximum number of modules in the complete network having L levels will be

$$\sum_{x=1}^L (2^c + c)^{(x-1)}$$

A network with 1 level can realize functions up to 3 variables, since there are 3 inputs as shown in Figure 5.1. By connecting x_i to the control input, the remaining x_j variables ($j \neq i$) or constants (0 or 1) can be connected to each of the 2 data input lines. So, there are 6 possible values for each data input line, resulting in 6^2 combinations. There are $3C_1$ combinations for selecting a variable as control input from the total of 3 variables. Hence, a total of $6^2 \times 3C_1$ combinations are possible for level 1. Among these, 24 are 3-variable functions, and only one level is required using the new exhaustive branching tree implementation. But, 3 levels are required for standard implementation and 2 levels for tree implementation.

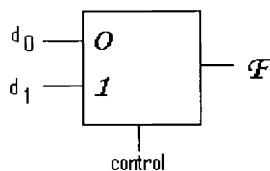


Figure 5.1: 1-control line multiplexer module

Functions with a maximum of 9 variables can be implemented using two level multiplexers with 3 control lines and 6 data lines. There are $9C_3 \times 3!$ combinations for selecting 3 variables as control inputs from the total of 9 variables. The remaining 6 variables and its complements or constants (0 or 1) at 6 data lines give rise to 14^6 functions. Hence $14^6 \times 9C_3 \times 3!$ input combinations are possible at this level. In the tree structure given by [A.E.A. Almaini, J.F. Miller and L Xu, 1992], at level 2, maximum number of variables possible is only 7, which results in $10^4 \times 7C_3 \times 3!$ combinations. The exhausting branching approach increases the number of variables and functions that can be implemented in 2 levels. As the levels increases this difference becomes more and more significant and hence reduction in delay can be achieved especially for functions with large number of variables. In general, with L levels the number of combinations possible in the proposed method is

$$\{ [2(y+1)]^y \} \times \{ z^L C z^{L-1} \} \times \{ z^{L-1}! \}$$

where $y = [z^L - z]$ and $z = 2^c + c$

Maximum number of variables at level L is

$$n_{\max} = z^L$$

For a given function if there are n dependent variables, the levels L required for implementation is given as

$$\lceil \log_{(2^c + c)} n \rceil \leq L \leq (n - 1)$$

Whereas in the tree structure, L can be in the range

$$\lceil \log_{(2^c)} n \rceil \leq L \leq (n - 1)$$

This clearly demonstrates a reduction in delay attained by this exhaustive branching technique. The following example illustrates the delay improvement achieved for a 9 variable function.

Consider the function,

$$F = x_8' x_7' x_3' x_1 + x_7' x_4' x_3' x_1 + x_8' x_7' x_4' x_1 + x_7' x_6' x_4' x_3' + x_8' x_7' x_6' x_4' + x_8' x_7' x_6' x_3' + x_9' x_8' x_4' x_2 + x_9' x_8' x_5' x_4' + x_9' x_4' x_3' x_2 + x_9' x_5' x_4' x_3$$

This function is implemented by a network of 4 modules using the proposed exhaustive branching technique that reduces the number of levels to 2, and is shown in Figure 5.2.

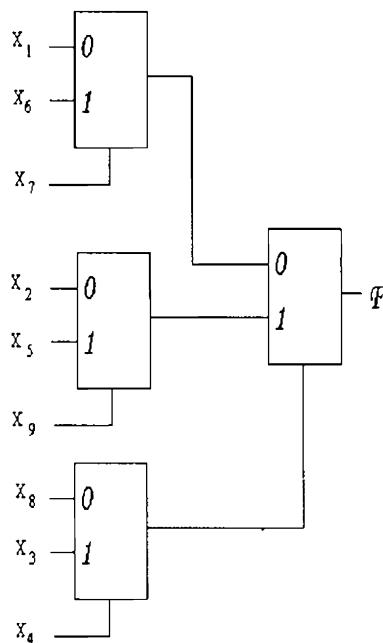


Figure 5.2: Implementation of a 9-variable function using exhaustive branching technique

5.3 Exhaustive Branching Technique

Behavior of a 1-control line multiplexer can be expressed as $F_s F_j + F_s F_k$, where F_s, F_j, F_k are functions of t variables ($1 \leq t \leq n$). The number of variables of F_s, F_j and F_k varies according to the complexity of the function to be realized. The maximum number of variables in F_s, F_j or F_k determines the delay of the network. The network terminates when F_s, F_j and F_k are 1's, 0's or x_i^* ($1 \leq i \leq n$). If all inputs except one terminates with a variable x_i^* or a logical constant and only one input continues into the next level, a cascade is generated where a single module is used in each level.

The algorithm presented aims to identify variables or functions of 2 variables at each control input, that eliminates as many branches as possible, and reduce the number of levels and modules required. This algorithm requires minterms as well as the number of variables of the function as inputs in order to implement the exact function. For example the minterms $\langle 0, 1, 2, 3 \rangle$ result in $F = 1$ for a two variable function; but for a 3 variable function $F = x_3$ where x_3 is the most significant variable.

The algorithm for exhaustive branching technique for a function given by minterms is as follows:

Step 1: Get the minterms and number of variables, n of the given function. Set the level, $L = 1$ and number of modules $M = 1$.

Step 2: List the minterms as minterm table. Check whether number of variables prior to L , $n - (L - 1) \leq 2$. If so, the tree is completed with any choice of remaining variables, and terminate.

Step 3: Check if there is any variable x_i for which the number of occurrences x_c in the minterm table is 0 or 2^{n-L} .

For $x_i = 1$, if $x_c = 0$ then data input 1 (d_1) = 0
 else if $x_c = 2^{n-L}$ then data input 1 (d_1) = 1.

For $x_i = 0$, if $x_c = 0$ then data input 0 (d_0) = 0
 else if $x_c = 2^{n-L}$ then data input 0 (d_0) = 1.

If both data inputs are constants, terminate.

Step 4: Check if there is any variable x_i for which number of occurrences, say x_c in the minterm table is 0 or 2^{n-L-1} . If so, check whether there is some variable x_j ($j \neq i$), for which x_j remains constant for that x_i . If so, terminate.

Step 5: $L = L + 1$, $M = M + 1$. Get the reduced minterm tables for each variable and find the x_i for which the following conditions are satisfied.

- (i) One reduced minterm table corresponds to a constant (0 or 1) or x_j^\bullet ($j \neq i$)
- (ii) The other reduced minterm table is a single module implementation by repeating step 4

Step 6: Get reduced minterm tables for each possible $(x_i \oplus x_j)$ or $x_i^\bullet x_j^\bullet$, and check whether the reduced minterm tables corresponds to constants or x_k^\bullet . If so, terminate.

Step 7: $M = M + 1$. Get the reduced minterm tables for each variable, and find the x_i for which the reduced minterm tables are single module implementations by repeating the step 4.

Step 8: Get the reduced minterm tables for each possible $(x_i \oplus x_j)$ or $x_i^*x_j^*$, and check whether the reduced minterm tables corresponds to a variable or constant, and a single module implementation by checking the conditions of step 5.

Step 9: $M = M + 1$. Get the reduced minterm tables for each possible $(x_i \oplus x_j)$ or $x_i^*x_j^*$, and check whether both reduced minterm tables are single module implementations by repeating the step 4.

Step 10: $L = L + 1$ and go to step 2.

Simulation is done up to 9-variable functions using 2 levels. The synthesis results obtained for various combinational logic functions implemented using 1-control line multiplexers with exhaustive branching algorithm are given in Section 6.5.

5.4 Summary

A novel algorithm for the synthesis of delay reduced multiplexer network is described. By suitable selection of variables or functions as control inputs, the number of modules and/or delay can be reduced. The reduction in number of modules results in reduced area of the synthesized network. Since

the number of variables is also given as input, exact functions are realized. The computation time is not always directly proportional to the number of variables, but increases with the complexity of the function to be realized. Since the topology of the delivered network is that of a tree, VLSI implementation of this network requires very few extra efforts in routing algorithms to redesign or for circuit layout. This approach can be adopted for the design of a more regular BCD digit multiplier.

Chapter 6

Simulation Results and Analysis

This chapter presents the simulation results of various multipliers and adders designed for decimal fused multiply-add unit as part of this research. This include simulation results of Double Digit Decimal Multiplier, BCD digit multiplier, fixed point multiplier using RPS algorithm, parallel decimal fixed point multipliers, floating point decimal multipliers and fused multiply-add unit. Performance analysis of different reversible BCD adders is also presented. The combinational logic synthesis results obtained using new exhaustive branching algorithm is presented. The salient features of the new system over other existing systems are compared and the results are tabulated.

6.1 Simulation Results and Analysis

This chapter presents the simulation results of various multipliers and adders designed for decimal fused multiply-add unit as part of this research.

6.2 Simulation Results of DFxP Multipliers

The designs for DFxP multipliers presented in Chapter 2 are implemented in VHDL using Leonardo Spectrum 0.18 micron, 1.8 V CMOS technology from Mentor Graphics Corporation with ASIC Library.

6.2.1 DDDM: Synthesis and Analysis

The design for Double Digit Decimal Multiplication (DDDM) is synthesised for 7-digit, 16-digit and 34-digit multipliers. The simulation results are given in Tables 6.1, 6.2 and 6.3. Figure 6.1 and Figure 6.2 give a comparison of area and delay respectively for various blocks of the DDDM for 7-digits.

Table 6.1: Area and Delay for various blocks of Double Digit Decimal Multiplier (7-digit × 7-digit)

Component (7 digit)	Area (μm^2)	% of total area	Delay (ns)	% of total delay
Secondary multiple generation block (SMG)	1355	10.67%	3.12	15.97%
Multiplier shift register (MSR)	648	5.09%	0.56	2.87%
Multiplexer blocks (MUX)	1270	9.98%	1.96	10.03%
Carry save addition block (CSA)	4894	38.49%	8.28	42.37%
Temporary Product register (TPR)	387	3.04%	0.44	2.25%
Decimal 4:2 compressor (4:2C)	2442	19.21%	5.45	27.89%
Partial Product shift register (PPR)	913	7.18%	0.79	4.05%
Decimal carry propagate adder (CPA)	844	6.64%	18.16	92.94%
Complete design	12713	100%	19.54	100%

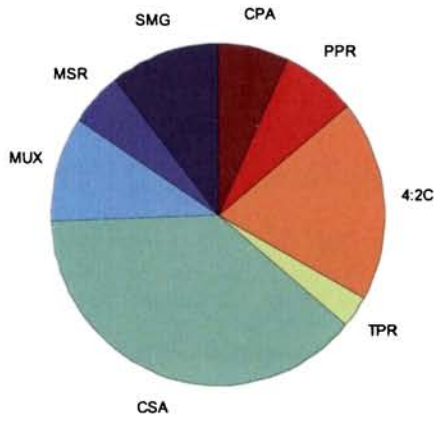


Figure 6.1: Area for different blocks of Double Digit Decimal Multiplier (7-digit x 7-digit)

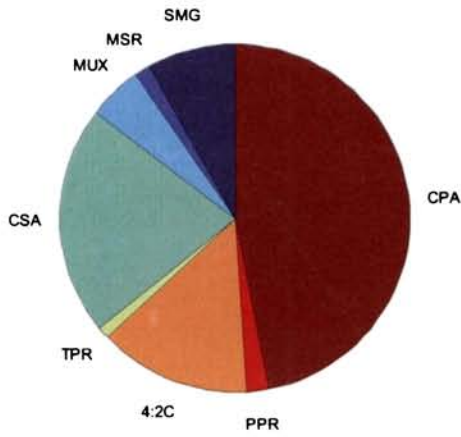


Figure 6.2: Delay for different blocks of Double Digit Decimal Multiplier (7-digit x 7-digit)

Table 6.2: Area and Delay for various blocks of Double Digit Decimal Multiplier (16-digit \times 16-digit)

Component (16 digit)	Area (μm^2)	% of total area	Delay (ns)	% of total delay
Secondary multiple generation block (SMG)	3106	11.7%	3.12	8.19%
Multiplier shift register (MSR)	648	2.44%	0.56	1.47%
Multiplexer blocks (MUX)	2694	10.15%	2.16	5.67%
Carry save addition block (CSA)	10550	39.75%	8.28	21.73%
Temporary Product register (TPR)	774	2.92%	0.44	1.15%
Decimal 4:2 compressor(4:2C)	4883	18.39%	5.45	14.30%
Partial Product shift register (PPR)	1838	6.92%	1.04	2.73%
Decimal carry propagate adder (CPA)	1688	6.36%	34.88	91.54%
Complete design	26539	100%	38.10	100%

Figure 6.3 and Figure 6.4 give a comparison of area and delay respectively for various blocks of the DDDM for 16-digits.

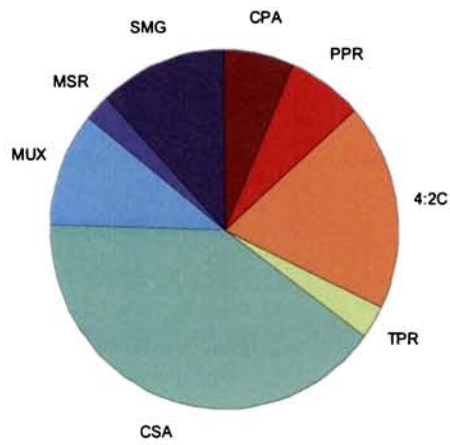


Figure 6.3: Area for different blocks of Double Digit Decimal Multiplier (16-digit x 16-digit)

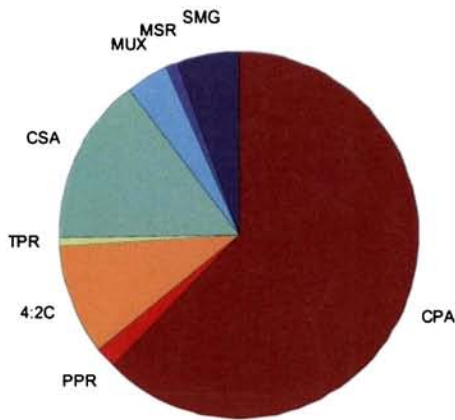


Figure 6.4: Delay for various blocks of Double Digit Decimal Multiplier (16-digit x 16-digit)

Table 6.3: Area and Delay for various stages of Double Digit Decimal Multiplier

(34-digit×34-digit)

Component (34 digit)	Area (μm^2)	% of total area	Delay (ns)	% of total delay
Secondary multiple generation block (SMG)	6607	11.86%	3.12	4.10%
Multiplier shift register (MSR)	2664	4.78%	0.73	0.96%
Multiplexer blocks (MUX)	5564	9.99%	2.18	2.86%
Carry save addition block (CSA)	21417	38.44%	8.28	10.88%
Temporary Product register (TPR)	1548	2.78%	0.44	0.59%
Decimal 4:2 compressor (4:2C)	10045	18.03%	5.45	7.16%
Partial Product shift register (PPR)	3858	6.93%	0.92	1.21%
Decimal carry propagate adder (CPA)	3471	6.23%	70.20	92.27%
Complete Design	55714	100%	76.08	100%

Figure 6.5 and Figure 6.6 give a comparison of area and delay respectively for various blocks of the DDDM for 34-digits.

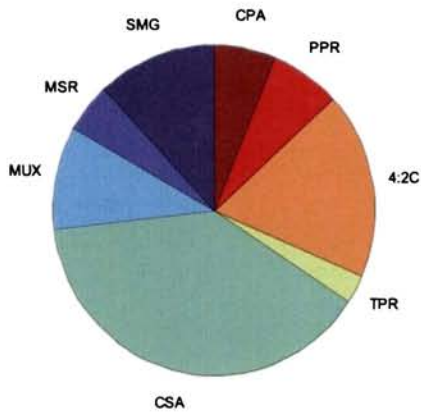


Figure 6.5: Area for various blocks of Double Digit Decimal Multiplier (34-digit×34-digit)

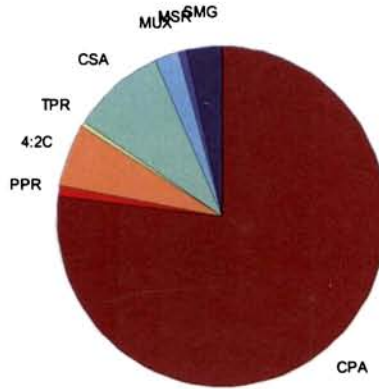


Figure 6.6: Delay for various blocks of Double Digit Decimal Multiplier (34-digit \times 34-digit)

Figure 6.7 and Figure 6.8 give a comparison of area and delay respectively for various blocks of the DDDM for different lengths. The comparisons show that the maximum area is occupied by the Carry Save Adder block, and maximum delay is for Decimal Carry Propagate Adder block for all lengths.

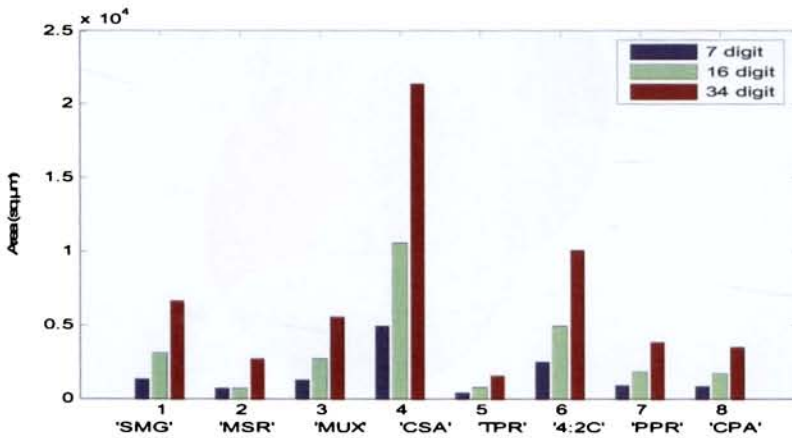


Figure 6.7: Area for various blocks of Double Digit Decimal Multipliers (7, 16, & 34-digits)

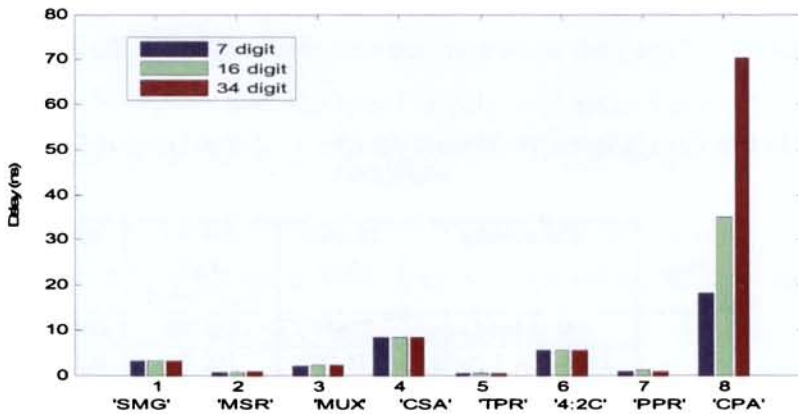


Figure 6.8: Delay for various blocks of DDDM (7, 16 & 34-digits)

34-digit multipliers are also designed as a combination of 17-digit multipliers. The simulation results are given in Table 6.4.

Table 6.4: Area and Delay for various blocks of 34-digit partitioned Double Digit Decimal Multiplier

Component (34 digit)	Area (μm^2)	% of total area	Delay (ns)	% of total delay
Secondary multiple generation block	6600	5.20%	3.12	3.03%
Multiplier shift register	1296	1.02%	0.56	0.54%
Multiplexer blocks	11336	8.93%	2.18	2.11%
17 digit Carry save addition block	37256	29.35%	5.66	5.49%
Temporary Product register	3268	2.57%	0.44	0.43%
17 digit Decimal 4:2 compressor block	20620	16.24%	5.45	5.29%
Partial Product shift register	8132	6.41%	0.82	0.79%
Decimal carry propagate adder	7128	5.62%	36.74	35.68%
34 digit carry save adder	5961	4.70%	2.83	2.75%
34 digit Decimal 4:2 compressor	9495	7.48%	5.45	5.29%
51 digit Decimal carry propagate adder	6773	5.34%	97.98	95.16%
Complete design	126908	100%	102.96	100%

Table 6.5 shows the comparison of the two designs for 34-digit DDDM.

Table 6.5: Comparisons for different designs of 34-digit Decimal Double Digit Multipliers

No: of Digits	Parameters	Double digit	Double digit partitioned	Ratio
34-Digit	Area (μm^2)	55714	126908	0.439
	Delay for 1 cycle (ns)	76.08	102.96	0.739
	Delay to complete 34 digit x 34 digit multiplication (ns)	1295.28	1029.6	1.26

It is noted that even though the area is almost double, the partitioned multiplier design gives a speed advantage of 1.26 times compared to the iterative DDDM for 34-digit multiplier. Table 6.6 indicates the comparison of area and delay parameters of DDDM with SDDM in [M. A. Erle and M. J. Schulte, 2003].

Table 6.6: Comparisons of DDDM and SDDM

No: of Digits	Parameters	Double digit	Single digit	Ratio
7-digit	Area (μm^2)	12713	8268	1.537
	Delay for 1 cycle (ns)	19.54	17.29	1.13
	Delay to complete 7 digit x 7 digit multiplication (ns)	90.8	130.4	0.696
16-digit	Area (μm^2)	26539	18717	1.42
	Delay for 1 cycle (ns)	38.10	36.24	1.05
	Delay to complete 16 digit x 16 digit multiplication (ns)	313.92	591.26	0.531
34-digit	Area (μm^2)	55714	39063	1.42
	Delay for 1 cycle (ns)	76.08	69.33	1.09
	Delay to complete 34 digit x 34 digit multiplication (ns)	1295.28	2388.4	0.542

It is noted that even though the area is increased by 50%, the speed to complete $n\text{-digit} \times n\text{-digit}$ multiplication is almost doubled as in the case of 16-digits and 34-digits. But, the speed is only 1.44 times for 7-digits because the number of cycles required is $\lceil (n/2)+1 \rceil$ which is equal to 5 cycles, compared to a single digit multiplier that requires 8 cycles.

Figure 6.9 and Figure 6.10 give a comparison of area and delay respectively of DDDM and SDDM for various lengths.

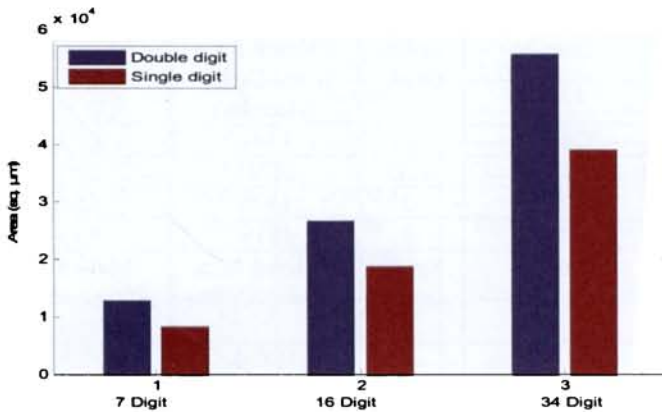


Figure 6.9: Area comparison of DDDM and SDDM

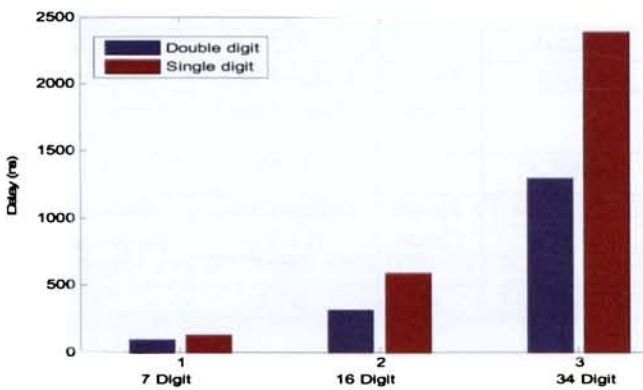


Figure 6.10: Delay comparison of DDDM and SDDM

The designs are synthesised in different families of various FPGAs as well and the results are tabulated. Table 6.7 shows the results for 7-digit DDDM on different families of various FPGAs.

Table 6.7: Simulation results of Double Digit Decimal Multipliers (7-digit) on FPGAs

FPGA	Family	Speed Grade	Utilized Area (Logic Cells)	Maximum Frequency(M Hz)	Delay (ns)
ALTERA	APEX 20 KE	-3	1109	22	47.04
	APEX 20 K	-3	1110	20.9	49.66
	ACEX 1 K	-3	1072	32.9	29.46
	FLEX 10 KE	-3	1072	25.2	37.54
ACTEL	Family	Speed Grade	Utilized Area (Core Cells* /Modules)	Maximum Frequency (M Hz)	Delay (ns)
	3200DX	-3	1544	10.1	98.08
	A500K	STD	2071*	20.7	47.6
	54SXA	-3	1526	15.6	63.78
	RT54SX	-1	1819	15.1	65.86
XILINX	Family	Speed Grade	Utilized Area (*Gates/CLBs)	Maximum Frequency(M Hz)	Delay (ns)
	Cool Runner	-6	*6804	0.5	2145.80
	Spartan	-4	504	18.7	54.64
	Spartan 2	-5	525	31.5	33.35
	Spartan XL	-4	504	18.7	54.64
	Virtex	-4	525	30.4	34.11
	Virtex E	-6	525	45.2	25.23
	XC9500XV	-7	4425	28.6	35.00
Quick Logic	Family	Speed Grade	Utilized Area (Logic Cells)	Maximum Frequency(M Hz)	Delay (ns)
	pASIC3	-1	1139	8.5	132.46
Lucent	Family	Speed Grade	Utilized Area (LUTs)	Maximum Frequency(M Hz)	Delay (ns)
	ORCA-3C/3T	-5	1074	20.7	50.37

Table 6.8 shows the simulation results of double digit multiplier for 16 digits on different families of various FPGAs.

Table 6.8: Simulation results of Double Digit Decimal Multipliers (16-digit) on FPGAs

FPGA	Family	Speed Grade	Utilized Area (Logic Cells)	Maximum Frequency(MHz)	Delay (ns)
ALTERA	APEX 20 KE	-3	2446	22	107.38
	APEX 20 K	-3	2447	20.9	113.60
	ACEX 1 K	-3	2405	32.9	34.95
	FLEX 10 KE	-3	2405	25.1	85.07
ACTEL	Family	Speed Grade	Utilized Area (Core Cells* /Modules)	Maximum Frequency (MHz)	Delay (ns)
	3200DX	-3	3348	10.1	206.39
	A500K	STD	4469*	20.1	83.14
	54SXA	-3	3357	15.0	127.20
	RT54SX	-1	3937	14.6	129.88
XILINX	Family	Speed Grade	Utilized Area (*Gates/ CLBs)	Maximum Frequency(MHz)	Delay (ns)
	Cool Runner	-15	*15588	0.5	2129.30
	Spartan 2	-6	1152	36.0	59.08
	Virtex	-6	1152	40.1	51.92
	Virtex E	-8	1152	58.6	36.75
Quick Logic	Family	Speed Grade	Utilized Area (Logic Cells)	Maximum Frequency(MHz)	Delay (ns)
	pASIC3	-1	2500	8.6	297.14

Table 6.9 shows the simulation results of double digit fixed point multiplier for 34 digits on different families of various FPGAs. The study reveals that efficient mapping of the double digit multiplier is achieved when Xilinx FPGA devices are used.

Table 6.9: Simulation results of Double Digit Decimal Multipliers (34-digit) on FPGAs

FPGA	Family	Speed Grade	Utilized Area (Logic Cells)	Maximum Frequency (M Hz)	Delay (ns)
ALTERA	APEX 20 KE	-3	12279	22	306.82
	APEX 20 K	-3	12283	20.9	325.57
	ACEX 1 K	-3	12191	32.9	112.54
	FLEX 10 KE	-3	12191	32.7	194.33
ACTEL	Family	Speed Grade	Utilized Area (Core Cells* /Modules)	Maximum Frequency (M Hz)	Delay (ns)
	3200DX	-3	17134	9.5	616.70
	A500K	STD	22048*	19.6	255.36
	54SXA	-3	16213	15.2	343.90
	RT54SX	-1	18759	14.7	346.58
XILINX	Family	Speed Grade	Utilized Area (*Gates/ CLBs)	Maximum Frequency (M Hz)	Delay (ns)
	Cool Runner	-15	*77218	0.5	2147.43
	Virtex E	-8	5765	59.9	99.49
Quick Logic	Family	Speed Grade	Utilized Area (Logic Cells)	Maximum Frequency (M Hz)	Delay (ns)
	pASIC3	-1	12427	8.6	870.73

The double digit decimal fixed point multiplier presented can be used in floating point multiplier circuits. Summarising the simulation results of DDDM design: It is noted that even though area is increased by 50%, the speed to complete n -digit \times n -digit multiplication is almost doubled. The design was validated using lengths of 7-digit, 16-digit, and 34-digit multipliers that are required for all the three formats of floating point decimal multiplication. The synthesized design has a latency of 90.8 ns, 313.92 ns, 1295.28 ns respectively for 7-digit, 16-digit, and 34-digit fixed point multipliers. Also, 34-digit multipliers are designed as a partitioned

combination of four 17-digit multipliers. It is seen that for the partitioned design, the speed is increased by 1.26 times compared to the iterative double digit decimal multiplier design. The area and delay comparisons for 7, 16 and 34-digit fixed point multipliers on different families of Xilinx, Altera, Actel and Quick logic FPGAs are also presented.

6.2.2 Simulation results of BCD Digit Multiplier

The design for BCD digit multiplication presented in Chapter 2 reduces the critical path delay and area that in turn allows for a fast multiplier design. A comparison of the new BCD digit multiplier design with the existing design by [Jaberipur and Kaivani, 2007] in terms of critical path area and delay is done and is tabulated in Table 6.10. The simulation is done using the logic synthesis tool Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library 0.18 micron, 1.8 V CMOS technology. The simulation results show significant improvement over the design by [Jaberipur and Kaivani, 2007].

Table 6.10: Comparison of area and delay of BCD digit multipliers

Type of Multiplier	Area (μm^2)	% reduction (area)	Delay (ns)	% reduction (delay)
Proposed Multiplier	489	8%	7.56	18%
Multiplier [Jaberipur Kaivani (2007)]	532		9.26	

The design is then extended to a Hex/Decimal multiplier that gives either a decimal output or a binary output depending on the requirement. A comparison of the Hex/Decimal multiplier design with one designed using the multiplier in [Jaberipur and Kaivani, 2007] in terms of area and critical path delay. The comparison shows that the proposed design has a reduction in delay of 16.52% with an extra hardware of 17.24% compared to the Hex/Decimal multiplier designed using the multiplier in [Jaberipur and Kaivani, 2007].

6.2.3 Simulation results of DFxP Multiplication using RPS Algorithm

The decimal fixed point multiplier using RPS algorithm presented in Chapter 2 was coded for a (7-digit \times 7-digit) multiplier in VHDL, and synthesized to evaluate the area and delay of the design. Synthesis was done using Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library of 0.18 micron, 1.8 V CMOS technology. An area and delay breakdown for an approximate contribution of major components of the design is given in Table 6.11. Even though the delay for the complete circuit is 30.2ns, the next cycle can start after 17.27 ns since the multi-operand BCD addition takes place simultaneously with the single digit multiplication of next set of inputs.

The new RPS algorithm generates and accumulates the partial products in an efficient manner for fixed point decimal multiplication. The design was validated using 7-digit \times 7-digit fixed point decimal multiplication that is required for a 32-bit floating point decimal multiplication. The latency for the multiplication of two n-digit BCD operands is (n+1) cycles, and a new

multiplication can begin every n cycle. The simulation results show that the RPS design gives a reduction in delay of 7.29% at the expense of area compared to the single digit implementation for 7-digit \times 7-digit fixed point multiplier. This iterative approach is suitable for high speed floating point multiplication since rounding can be initiated during the fixed point multiplication process.

Table 6.11: Area and Delay for different stages of Decimal Fixed Point Multiplier using RPS Algorithm (7-digit \times 7-digit)

Component	Area		Delay	
	μm^2	%	ns	%
Controller	2719	15.01%	4.85	16.05%
Single digit multiplier array	3465	19.15%	7.81	25.86%
BCD adder array	5791	32.01%	12.93	42.81%
Register array	6114	33.8%	4.61	15.26%
Fixed point multiplier	18089	100%	30.2	100%

6.2.4 Simulation Results for Parallel Decimal Multipliers

The Fixed Point Parallel Decimal Multiplier presented in Chapter 2 is realized for a (7-digit \times 7-digit) fixed point multiplication. The simulation results of this design are shown as 'Design 1' in Table 6.12. The simulation of the modified design for partial product generation using BCD digit multipliers

while keeping the same DCA and Carry counters for partial product accumulation by [Lang and Nannarelli, 2006] is also done. The realization of 7-digit multiplier using this modified design is simulated, and is shown as ‘Design 2’ in Table 6.12. The simulation results show that the design using Carry Counters (Design 2) has reduced area and delay compared to the design using Decimal 4:2 Compressors (Design 1) by 3.52% and 9.79% respectively. The designs are extended to 16-digit and 34-digit multipliers since they form integral components of 64-bit and 128-bit decimal floating point multipliers. The simulations are done using the logic synthesis tool Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library of 0.18 micron, 1.8 V CMOS technology. A comparison of area and critical path delay for ‘Design 1’ and ‘Design 2’ is done, and is tabulated in Table 6.13. Even though the area is more for 16-digit and 34-digit realizations in ‘Design 2’, the delay is always less than that for ‘Design 1’. Parallel multipliers are used when the constraint is delay compared to area, and hence ‘Design 2’ is the more suitable design. Comparison of 16-digit multiplier using ‘Design 2’ with the designs by [Lang and Nannarelli, 2006] shows a reduction in area and delay. The area occupied by the partial product generation block in ‘Design 2’ is only $141,504 \mu\text{m}^2$ while that for [Lang and Nannarelli, 2006] and [Dadda, Nannarelli and Milano, 2008] is $155,000 \mu\text{m}^2$. The delay for the generation of partial products is only 20.13% of total delay for ‘Design 2’, while the delay in the design of [Lang and Nannarelli, 2006] is 26.41%. So, there is a reduction of 8.71% in area and 6.28% in delay for ‘Design 2’ compared to the design of [Lang and Nannarelli, 2006].

Table 6.12: Area and Delay for different stages of Decimal Fixed Point Parallel Multipliers

Component		Area		Delay		
		μm^2	%	ns	%	
Generation of Partial Products		23961	52.24%	7.56	16.63%	
Decimal Carry Save Adder Block	DCA Block	7140	15.56%	2.83	6.23%	
	Rest of the levels	D4:2C Block (Design1)	12870	28.06%	21.19	46.61%
		DCA & CC (Design 2)	11256	25.43%	16.74	40.82%
Fixed Block Fast Decimal Adder		1898	4.14%	13.88	30.53%	
Fixed point multiplier (7-digit \times 7-digit)	D4:2C Block (Design1)	45869	100%	45.46	100%	
	DCA & CC (Design 2)	44255		41.01		

Table 6.13: Comparison of Decimal Fixed Point Parallel multipliers for different lengths

Multiplier using	Area (μm^2)			Delay (ns)		
	7-digit	16-digit	34-digit	7-digit	16-digit	34-digit
D4:2C (Design1)	45869	243449	1111817	45.46	52.78	67.18
DCA & CC (Design 2)	44255	256753	1147097	41.01	51.6	62.56

Partial product accumulation was also done column wise for a (7-digit \times 7-digit) fixed point multiplier. The area occupied and delay caused by major components for 7-digit \times 7-digit multiplier is given in Table 6.14.

Table 6.14: Area and Delay for different stages of Decimal Fixed Point Multiplier (7-digit×7-digit) Column Accumulation

Component	Area		Delay	
	μm^2	%	ns	%
Generation of Partial Products	23961	54.92%	7.56	16.63%
DCA & CC	16108	36.92%	17.62	47.34%
Fixed Block Fast Decimal Adder	3558	8.16%	12.04	30.53%
Fixed point multiplier	43627	100%	37.22	100%

Comparison of 7-digit multiplier using column accumulation with row accumulation ('Design 1' and 'Design 2') shows a reduction in area and delay. The results show that there is a reduction in delay of 22.14 % and 10.18% for column accumulation compared to that of 'Design 1' and 'Design 2' respectively. The decrease in area for column accumulation compared to 'Design 1' and 'Design 2' is by 5.14% and 1.44% respectively. This comparison shows that the better design for a parallel decimal multiplier is the one using single digit multipliers for partial product generation, and column wise approach for partial product accumulation using carry counters and decimal carry save adders.

6.3 Simulation results for Decimal Floating Point Units

6.3.1 Simulation results for Decimal Floating Point Multipliers

Two approaches for iterative decimal floating point multiplication are presented in Chapter 3. The first approach has a decimal fixed point multiplier using RPS algorithm. The DFP multiplier using RPS algorithm is synthesized for a 32-bit input using Leonardo Spectrum from Mentor Graphics

Corporation with ASIC Library of 0.18 micron, 1.8 V CMOS technology. An area and delay breakdown for an approximate contribution of major components of the design is given in Table 6.15.

Table 6.15: Area and Delay for different stages of DFP Multiplier using RPS Algorithm (32-bit)

Component	Area	Delay
	μm^2	ns
Decoding Logic	1439	3.26
Exponent generation	229	4.42
Exception handling	66	4.21
DFxP multiplier & Rounding Unit	25481	41.8
Encoding Logic	600	5.25
Total	27926	53.67

Even though the total area is the sum of area of different stages, the total delay for the complete circuit is only 53.67ns, which is less than the sum of delays of all stages. This is because of the inherent parallelism in the design. Table 6.16 shows the synthesis results of DFxP Multiplier and rounding unit.

Table 6.16: Area and Delay of Rounding Unit and DFxP Multiplier using RPS Algorithm (7-digit \times 7-digit)

Component	Area	Delay
	μm^2	ns
DFxP Multiplier	18089	30.2
Incrementer 1	3536	22.71
Incrementer 2	3536	22.71
Mux	205	0.24
Sticky bit	51	1.15
Rounding	64	1.96
Total	25481	41.8

Here, also the total delay is less than the sum of delays of each component as rounding starts before the DFP multiplication is completed. This is possible since the sticky bit (Sb), the round digit (R) and the guard digit (G) are generated on-the-fly during the DFP multiplication process. For a 7-digit \times 7-digit DFP multiplication, the sticky bit (Sb) is generated after the fourth cycle; round and guard digit generation is done in the next 2 cycles. The delay break up of different components of DFP multiplication is shown in Figure 6.11. Figure 6.12 gives a detailed delay breakup of the DFP Multiplier and the rounding unit. The delay break up for one DFP multiplication of 32-bit inputs is shown in Figure 6.13. DFP multiplication takes 9 cycles to complete one 32-bit DFP multiplication with worst cycle time being 19.97 ns. Hence, the total delay is 179.73 ns.

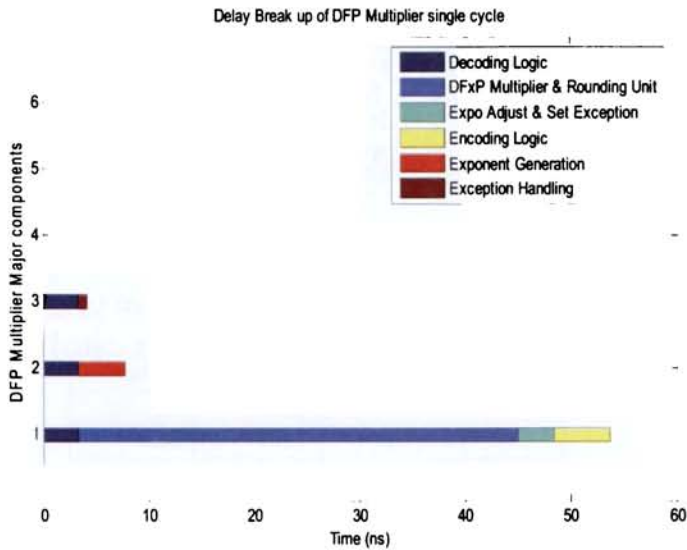


Figure 6.11: Delay Break up of different components of DFP Multiplier using RPS Algorithm

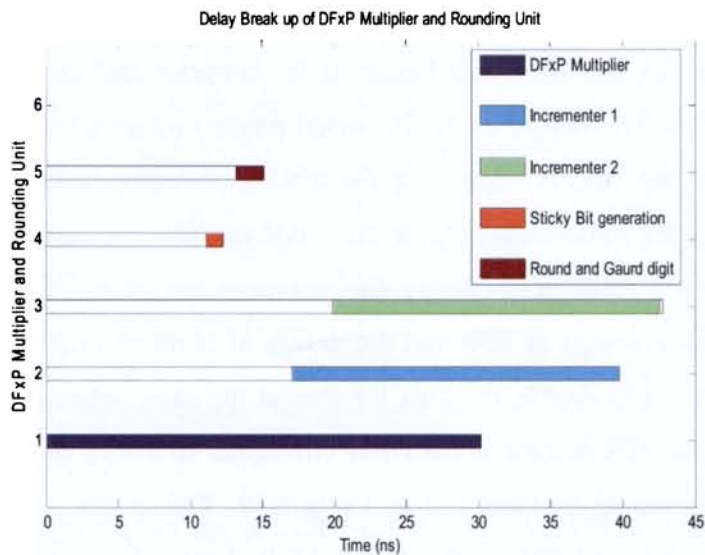


Figure 6.12: Delay Break up of DFP Multiplier (using RPS) and Rounding Unit

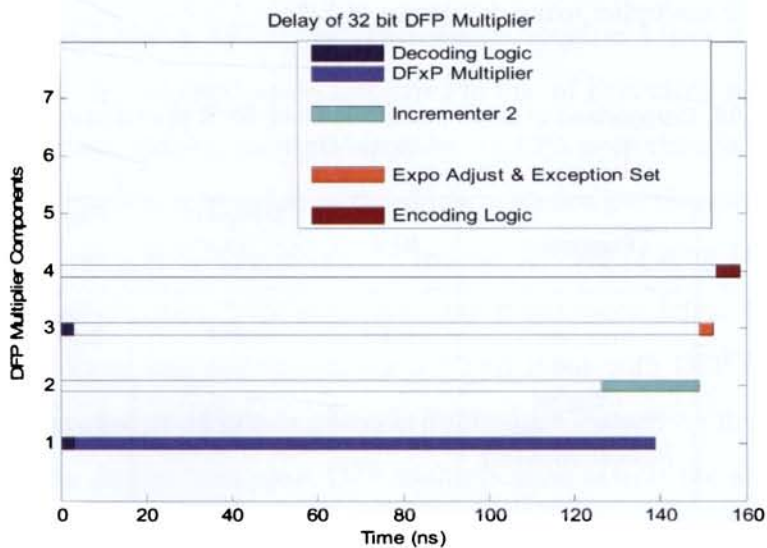


Figure 6.13: Delay break up of 32-bit DFP Multiplier using RPS algorithm

When multiplying two DFP numbers with n -digit significands, using RPS algorithm, the worst case latency is $(n+2)$ cycles, and initiation interval is $(n+1)$ cycles. The design differs in partial product generation and in rounding logic from the iterative approach for the DFP multiplication by Erle. For comparing the performance, both these designs are synthesized in the same environment. Table 6.17 shows the comparisons of the DFP multiplier designs using design of RPS and the design of [Erle *et al.*,2003]. The design using RPS needs double the area for almost the same latency. But when the design using RPS is used as the DFP Multiplier of a DFP multiplication, the speed is increased as tabulated in Table 6.18. This is because the rounding process is initiated before the DFP Multiplication cycles are over. This parallelism achieved in the DFP multiplier design using RPS algorithm decreases the delay of the critical path. This in turn reduces the worst cycle time. A delay reduction of 25.12% is achieved using this approach compared to the DFP multiplier using the design of Erle.

Table 6.17: Comparison of DFP multipliers using RPS algorithm with existing one (7-digit \times 7-digit)

Parameters	RPS	Design of Erle	Ratio
Area (μm^2)	18089	8268	2.18
Delay of 7-digit \times 7-digit multiplication (ns)	133.55	130.4	1.024

Table 6.18: Comparison of DFP Multipliers using RPS algorithm with existing one (32-bit \times 32-bit)

Parameters	RPS	DFP multiplier using design of Erle,	Ratio
Worst cycle time (ns)	19.97	26.67	0.749
Maximum frequency (MHz)	50	37.5	1.333
Delay of 32-bit DFP multiplication in terms of worst cycle time (ns)	179.73	240.03	0.749

The DFP multiplier using DDDM, for a 32-bit input is synthesized to find the area and delay associated with the design. Synthesis was done using Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library of 0.18 micron, 1.8 V CMOS technology. DFP multiplication takes 6 cycles to complete one 32-bit DFP multiplication with worst cycle time being 31.56 ns. Hence, the total delay is 189.36 ns. This design requires lesser number of clock cycles for DFP multiplication compared to that of Erle using single digit design. This in turn reduces the total delay by 21.10% even though the worst case cycle time is more compared to the single digit design. Compared to the first approach (using RPS algorithm), the second approach (using DDDM) is more regular and occupies 26% lesser area, but it has more delay. The delay comparison of these two approaches for a 32-bit input with DFP multiplier design using the design of Erle is given in Table 6.19. Extending the iterative DFxP multiplier design to support DFP multiplication affects the area, cycle time, latency, and initiation interval.

Table 6.19: Comparison of Iterative DFP Multipliers for 32-bit input

Parameters	DDDM	RPS	DFP multiplier using design of Erle
Worst cycle time (ns)	31.56	19.97	26.67
Maximum frequency (MHz)	31.68	50	37.5
Delay of 32-bit DFP multiplication in terms of worst cycle time (ns)	189.36	179.73	240.03

The parallel DFP multiplier is synthesized for a 32-bit input using Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library of 0.18 micron, 1.8 V CMOS technology. Synthesis results show that the parallel DFP multiplication takes 69.54ns to complete one 32-bit DFP multiplication. The parallel DFP design, the design using DDDM and DFP multiplier using design of Erle is synthesized in the same environment and the comparison of delays is given in Table 6.20.

Table 6.20: Comparison of DFP Multipliers (using parallel, DDDM & of Erle) for 32-bit Input

Parameters	Using Parallel multiplier	Using DDDM	DFP multiplier using design of Erle
Delay of 32-bit DFP multiplication (ns)	69.54	189.36	240.03

6.3.2 Simulation results of Decimal MAC unit

The fused multiply-add unit uses parallel and iterative multipliers and a floating point adder unit. The DFP adder is implemented using ripple carry BCD adders, Kogge-Stone adders and reduced delay BCD adders.

6.3.2.1 Simulation results of DFP Adders

Different designs of DFP adders are presented in Chapter 3. Analysis done on different implementations of DFP adders are tabulated in Table 6.21. Comparison of DFP adders shows that the 'reduced delay adder' achieves the highest speed.

Table 6.21: Comparison of DFP Adders (16- digit)

DFP Adders	Delay (ns)	Area (μm^2)
Ripple Carry Adder	124.62	65432
Kogge-Stone Adder	127.42	59358
Reduced Delay Adder	109.76	60327

Comparison of different designs of 32-bit DFP MAC unit is summarised in Table 6.22. Both iterative and parallel multiplier designs are used for different DFP MAC designs. DDDM design and design using RPS algorithm are the iterative multipliers used. For parallel multiplier, the 'Design 2' for row accumulation and the design for column accumulation are made use of. The 7-digit significands of 32-bit DFP inputs are multiplied by the 7-digit DFxP Multipliers. This result in 14-digit output since rounding of the result is not performed after multiplication. The adder unit accepts this as input and should be able to accommodate this length. Hence adder length should be

greater than 14 digits. The MAC designs presented in this research uses 16-digit adders. This is because, 16-digit adders form the integral part of a 64-bit DFP adder unit. Three different DFP adders are used in MAC implementation: Ripple carry adders, Kogge-stone adders and ‘reduced delay DFP adders’. Designs are simulated using Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library of 0.18 micron, 1.8 V CMOS technology. The comparison results show that for a high speed fused multiply-add unit the best suited multiplier is parallel multiplier using column accumulation fused with a ‘reduced delay DFP adder’. Such high speed units are of great importance in real time computations such as in military applications.

Table 6.22: Comparison of DFP MAC units (32-bit)

Unit		Delay (ns)	Area (μm^2)	
Decoding Unit		7.68	1845	
Multiplier	Iterative	DDDM	90.8	12713
		RPS Algorithm	133.55	18089
	Parallel	Row Accumulation	41.01	44255
		Column Accumulation	37.22	43627
Adder	Ripple Carry		124.62	65432
	Kogge-Stone		127.42	59358
	Reduced Delay DFP adder		109.76	60327
Correction Unit		54.17	9640	
Encoding Unit		5.25	600	

6.4 Performance Comparison of Reversible Circuits for Decimal Adders

Improved designs for reversible logic implementation of BCD adder are presented in Chapter 4. Results of analysis done on reversible implementations of BCD adders in [Hafiz Md. Hasan Babu and A. R. Chowdhury, 2005], [Himanshu. Thapliyal, S. Kotiyal and M.B Srinivas, 2006] compared with the two designs presented in Chapter 4 are tabulated in Table 6.23.

Even though this delay analysis will not give exact results because of the difference in complexity of the gates used, it gives a good estimate of the delay reduction attained by reversible implementation of BCD adders presented. The Table 6.23 also shows a comparison in terms of number of reversible gates and garbage outputs for the complete circuit. It is clear that the implementation in Figure 4.18 uses least number of gates, produces least number of garbage outputs and gives least delay compared to all other implementations.

Reversible implementations of three different fast decimal adders are also presented in Chapter 4.

A comparison of classical logic gate implementation of conventional, carry select and hybrid (with $m=4$) BCD adders is done to study how the speed and area of the different designs vary with number of digits. Designs are simulated with the logic synthesis tool Leonardo Spectrum from Mentor Graphics Corporation using ASIC Library. The critical path delay and area are normalized with respect to a full adder critical path delay of 1.98 ns and area of $38\mu\text{m}^2$. Figure 6.14 shows the delay, and Figure 6.15 shows the area-delay

product normalized to that of a full adder. The area overhead of carry select and hybrid adders is compensated by the speed advantage compared to the conventional adder.

Table 6.23: Comparative Analysis of the Reversible BCD Adders

Reversible BCD Adders	Complete Circuit			
	No: of gates	No: of garbage o/p	Delay of Decimal Cout for N-digit BCD adder	Delay of BCD Sum for N-digit BCD adder
BCD Adder in [Hafiz Md. Hasan Babu and A. R. Chowdhury, 2005]	NG-11 NTG-8 FG-3 Total-22	22	12N	12N+7
BCD Adder using TSG [Himanshu. Thapliyal, S. Kotiyal and M.B Srinivas, 2006] (fanout>1)	TSG-8 NG-3 Total-11	22	7N	7N+3
BCD Adder using TSG [Himanshu. Thapliyal, S. Kotiyal and M.B Srinivas, 2006] (fanout=1)	TSG-8 NG-3 FG-5 Total-16	22	9N	9N+4
BCD Adder presented in Figure 4.17 (fanout=1)	TSG-5 NG-3 FG-3 Total-11	13	7N	7N+3
BCD Adder presented in Figure 4.18 (fanout=1)	TSG-5 NG-3 FG-1 Total-9	11	7N	7N+1

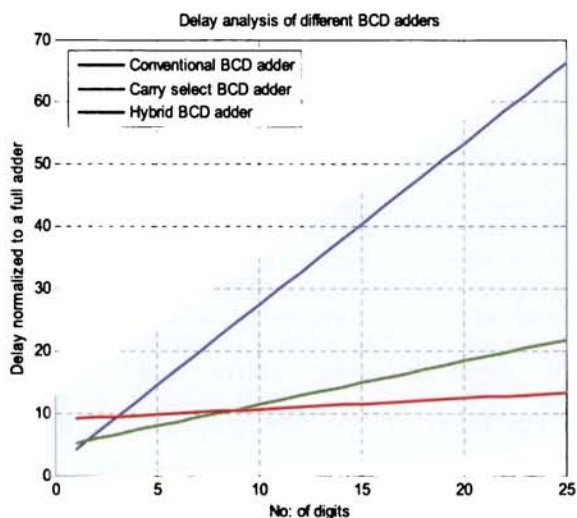


Figure 6.14: Delay analysis of Conventional, Carry Select and Hybrid BCD Adders using classical logic gates

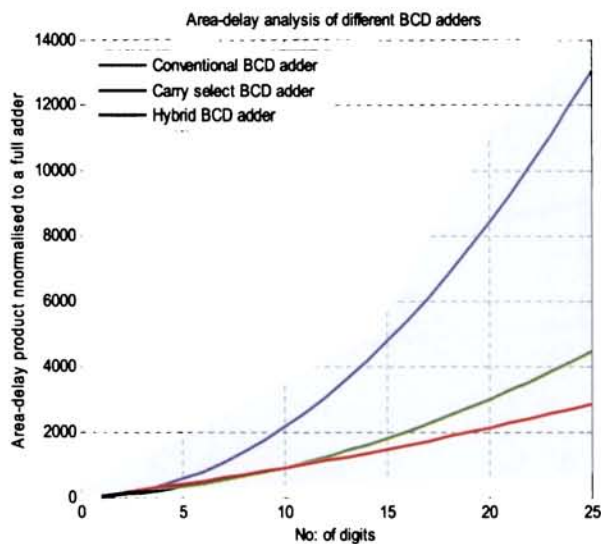


Figure 6.15: Analysis of area-delay product of Conventional, Carry Select and Hybrid BCD Adders using classical logic gates

Figure 6.16 demonstrates the speed up factor of carry select and hybrid BCD adders compared to conventional BCD adder as the number of digits increases. Hybrid decimal adder is three times faster than the conventional BCD adder as the number of digits increases above 12 while the carry select BCD adder attains a speed up factor of 2.5 at this level. It is noted that the hybrid adder attains speed up over carry select BCD adder only when the number of input digits increases above 8 for a classical logic gate implementation. The delay comparison graphs show that hybrid adder is 5 times faster than that for the conventional BCD adder, when the input word length is above 25.

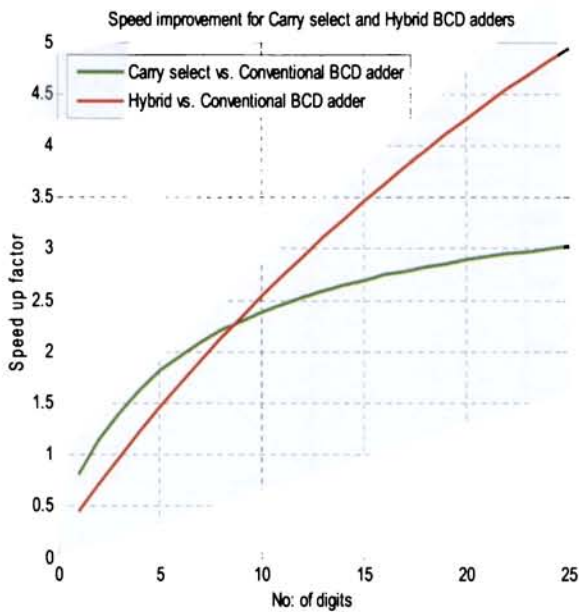


Figure 6.16: Speed up factor for Carry select and Hybrid BCD adders vs. conventional BCD adder for classical logic gates

Figure 6.17 shows a comparative delay analysis of conventional, carry select and hybrid BCD adder reversible implementations normalized to that of a Fredkin gate. Figure 6.18 demonstrates the speed up factor of reversible Fredkin gate implementations of carry select and hybrid BCD adders compared to conventional BCD adder. It can be noted that the hybrid adder attains speed over carry select BCD adder for all values of N in reversible implementation. Speed up factor of hybrid adder increases above 10 when the number of decimal digits is more than 25 for reversible logic implementation using Fredkin gates.

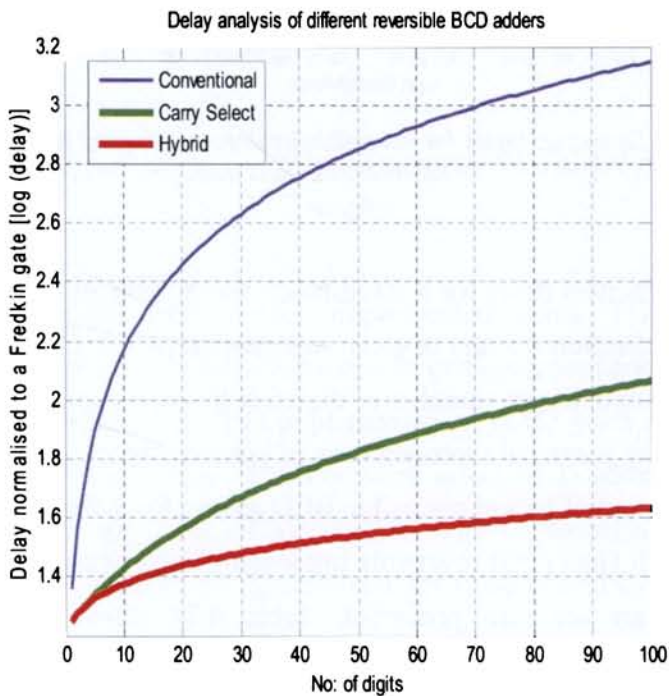


Figure 6.17: Delay analysis of reversible BCD adders using Fredkin gates

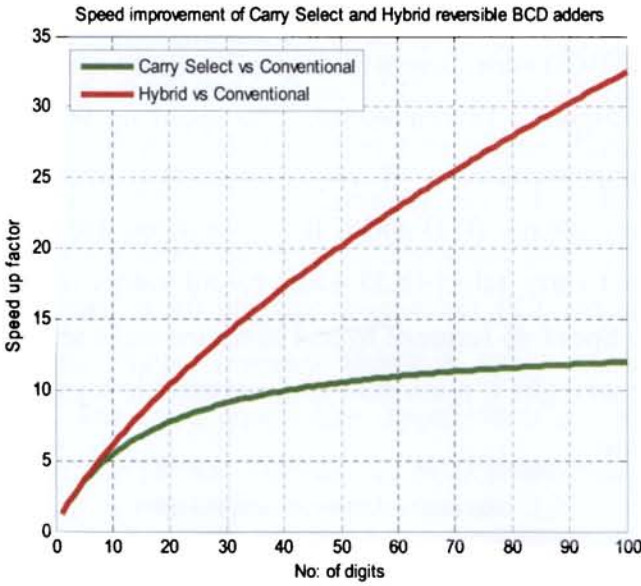


Figure 6.18: Speed up factor for reversible implementations of fast BCD adders vs. Conventional BCD adder

The shortest delay for a fixed block size hybrid BCD reversible adder is derived in Section 4.5 and is given in Equation (4.37). Optimum block size for different input length (number of digits) is given in Equation (4.36). Figure 6.19 shows the graphical representation of optimum block size (corresponding to shortest delay) of hybrid reversible BCD adders for different input lengths.

Toffoli Gate (TG) reversible implementations of conventional and fast decimal adders are also presented. Table 6.24 shows a comparison of implementations of different BCD adders using FRGs and TGs in terms of quantum cost, number of reversible gates, garbage outputs and delay. The percentage reduction attained in quantum cost is computed as $[QC(FRG) - QC(TG)] / QC(FRG)$.

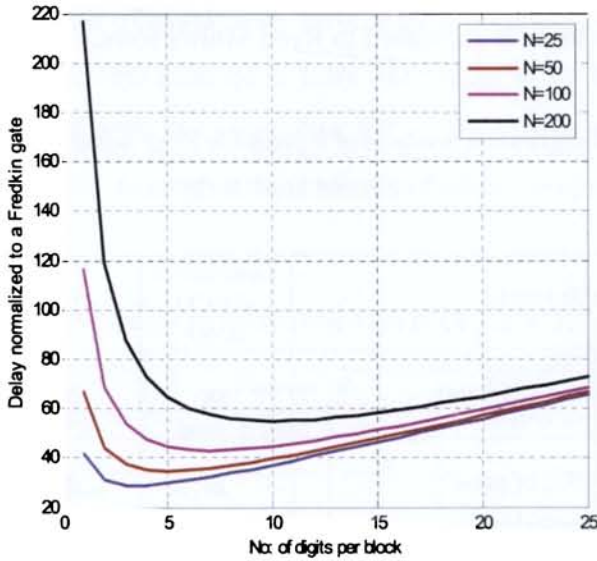


Figure 6.19: Optimum block size of Hybrid reversible BCD adder for different input lengths

Conventional BCD adder implemented using TGs shows 45% reduction in quantum cost, 67% reduction in garbage count and 18% reduction in gate count compared to FRG implementation. QAD gives a corresponding reduction of 38%, 63% and 15%. For carry select BCD adder the respective reduction factors are 37%, 64% and 12%. The implementation using TGs gives a reduction in delay for conventional BCD adder and QAD compared to FRG implementation.

The performance comparison of implementations using FRG and TG for different BCD adders reveals that the implementations using Toffoli gates are superior in terms of quantum cost, garbage count and gate count, compared to Fredkin gate implementations. Toffoli gates are also suitable for implementations of Reed Muller expressions. Hence decimal adders can be

implemented in reversible logic using lesser number of gates and garbage count when the logic is expressed in Reed Muller form.

Table 6.24: Comparative analysis of Reversible BCD Adders Implemented using Toffoli and Fredkin Gates

Reversible BCD adders		Convention al BCD adder	QAD	Carry select BCD adder	
Toffoli Implement ation	Quantum cost	58	78	79	
	No: of gates	28	34	35	
	Garbage count	12	16	15	
	Delay for an N-digit adder	Decimal Cout	9N	11+N	12+N
		BCD Sum	6+9N	18+N	19+N
Fredkin Implement ation	Quantum cost	105	126	126	
	No: of gates	35	42	42	
	Garbage count	36	45	44	
	Delay for an N-digit adder	Decimal Cout	11N	10+2N	11+N
		BCD Sum	6+11N	15+2N	17+N

RPS gates presented in Section 4.6 are substitutes for reversible implementations for 4-bit Binary to BCD converters. Table 6.25 shows the comparison of 4-bit Binary to BCD converters using various universal

reversible gates and RPS gates. Fully reversible RPS gate gives minimum delay, optimum number of gates with least logical complexity and optimum garbage for the implementation of a fully reversible 4-bit binary to BCD converter. The use of partially reversible RPS gate makes circuit partially reversible but decreases the logical complexity further compared to its fully reversible gate.

Table 6.25: Comparison of Reversible 4-Bit Binary to BCD Converters

Type of gate	Number of			Logical complexity
	Gates	level s	garbag e	
Any gate (One possible implementation)	NG-4 HNG-1	5	4	$13\alpha+10\beta+12\delta$
FRG	8	5	9	$16\alpha+32\beta+8\delta$
HNG	5	5	8	$25\alpha+10\beta$
Fully Reversible RPS	1	1	NIL	$8\alpha+10\beta+3\delta$
Partially Reversible RPS	1	1	NIL	$7\alpha+9\beta+3\delta$

α = A two input XOR gate calculation β = A two input AND gate calculation δ = A NOT calculation

Comparative analysis of BCD adders implemented using fully and partially reversible RPS gates, HNG gates and HNG-RPS combination is done for garbage count, number of gates, delay and logical complexity. Similar analysis is done on reversible implementations of BCD adders in [H.

Thapliyal, S. Kotiyal and M.B Srinivas, 2006] and [M. Haghparast and K. Navi, 2008], and are tabulated in Table 6.26.

Table 6.26: Comparative Analysis of Reversible BCD Adders for logical complexity

Reversible BCD Adders	Complete Circuit		Delay of Decimal Cout for an N-digit adder	Delay of BCD Sum for an N-digit adder	Logical complexity
	No: of gates	No: of garbage o/p			
BCD Adder using TSG [H. Thapliyal, S. Kotiyal and M.B Srinivas, 2006] (fanout>1)	TSG-8 NG-3 Total-11	22	7N	7N+3	$54\alpha+30\beta+33\delta$
BCD Adder using TSG [H. Thapliyal, S. Kotiyal and M.B Srinivas,2006] (fanout=1)	TSG-8 NG-3 FG-5 Total-16	22	9N	9N+4	$59\alpha+30\beta+33\delta$
BCD Adder using HNG [M. Haghparast and K. Navi, 2008] (fanout=1)	HNG-1 HNG-8 NG-2 FG-2 TG-1 Total-14	22	9N	9N+4	$49\alpha+21\beta+6\delta$
BCD Adder using HNG gates only (fanout=1)	HNG-10	19	7N	7N+3	$50\alpha+20\beta$
BCD Adder using RPS gates only (fanout=1)	Fully Reversible RPS-9 Total -9	12	9N	9N	$72\alpha+90\beta+27\delta$
BCD Adder using HNG-RPS (fanout=1)	HNG-4 Fully Reversible RPS-1 Total -5	8	5N	5N	$28\alpha+18\beta+3\delta$
Partial Reversible BCD Adder (fanout=1)	HNG-4 Partially Reversible RPS-1 Total -5	8	5N	5N	$27\alpha+17\beta+3\delta$

α = A two input XOR gate calculation, β = A two input AND gate calculation, δ = A NOT calculation

The implementations using combination of HNG-RPS gates give a reduction of 63% in garbage count compared to the implementation [H. Thapliyal, S. Kotiyal and M.B Srinivas, 2006] and [M. Haghparast and K. Navi, 2008] the reduction factor in garbage count is 63%.

Comparison in number of gates gives a reduction of 68% for [H. Thapliyal, S. Kotiyal and M.B Srinivas, 2006], and 64% for [M. Haghparast and K. Navi, 2008] respectively. BCD adder implementation using only RPS gates makes use of 9 RPS gates with 12 garbage outputs. This implementation gives a reduction of 10% in number of gates and 36% for garbage count when compared to the implementation using HNG gates only. It is shown that the reversible BCD adders presented have lower hardware complexity and it is much better and optimized in terms of number of reversible gates, garbage count and delay when compared with the existing counterparts. The logic designs are simulated in VHDL using the logic synthesis tool Leonardo Spectrum from Mentor Graphics Corporation.

Different implementations for the reversible (7, 4) hamming error coding and detection circuits are also presented. Table 6.27 shows the comparisons between different implementations of hamming code generation and detection circuits in terms of number of gates, garbage outputs and levels. It is seen that the design using 4×4 HCG is highly optimized in terms of number of reversible gates and/or garbage outputs. The approach using PPHCG also provides a way of incorporating fault tolerance into reversible circuits without much extra design effort and with modest hardware overhead.

Table 6.27: Comparison of Reversible Hamming Code Generation and Detection Circuits

Reversible 7-bit Hamming Code circuit	No. of reversible gates	No. of Garbage outputs	No. of levels
HC generator using HCG and FG	4	NIL	2
HC generator using F2G and FG	6	NIL	4
Parity preserving HC Generator using PPHC gate	5	5	2
Parity preserving HC Generator using F2G	6	4	4
HC error detector using HCG and FG	4	4	2
Parity preserving HC error detector using F2G	5	6	3

6.5 Logic synthesis simulation using Multiplexers

An exhaustive branching technique to obtain reduced hardware and/or delay for synthesizing logic functions using multiplexer universal logic modules is presented in Chapter 5. The technique is demonstrated using the following examples.

Example 1:

Implementation of the 4-variable function, $F = \sum (4, 7, 9, 10, 12, 13, 14, 15)$

Step 1: minterms=(4,7,9,10,12,13,14,15), $n=4$, $L=1$, $M=1$.

Step 2: Form a 4 bit binary minterm table as shown in Table 6.28.

Table 6.28: 4-bit binary minterm table

x_4	x_3	x_2	x_1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Here, $n = 4$ and $L = 1$. So, $n - (L - 1) \leq 2$ is not satisfied.

Step 3: Let $i = 4$. For $x_4 = 1$, $x_c = 6$ which is neither 0 nor $2^{n-L} (=8)$.

For $x_4 = 0$, $x_c = 2$ which is neither 0 nor $2^{n-L} (=8)$.

Similarly no other variable satisfies the conditions.

Step 4: Consider x_2 . Then, $x_c = 4$ which is $2^{n-L-1} (=4)$.

But none of the remaining variables are constant for x_2 . Checking the same conditions for other variables it is found that none of the variables satisfies the conditions.

Step 5: $L=2$, $M=2$. Consider x_3 .

The reduced minterm table for $x_3 = 0$ is given in Table 6.29 and for $x_3 = 1$ in Table 6.30 respectively.

Table 6.29: The reduced minterm table for $x_3 = 0$

x_4	x_2	x_1
1	0	1
1	1	0

Table 6.30: The reduced minterm table for $x_3 = 1$

x_4	x_2	x_1
0	0	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Here, none of the minterm tables corresponds to the condition (i). So try for another variable. No other variable satisfies the conditions.

Step 6: Therefore, consider $(x_2 \oplus x_1)$. The reduced minterm table for $(x_2 \oplus x_1) = 1$ is given in Table 6.31 and for $(x_2 \oplus x_1) = 0$ is given in Table 6.32 respectively.

Table 6.31: The reduced minterm table for $(x_2 \oplus x_1) = 1$

x_4	x_3
1	0
1	0
1	1
1	1

This table reduces to x_4 , for a 2 variable truth table with x_4 and x_3 as variables.

Table 6.32: The reduced minterm table for $(x_2 \oplus x_1) = 0$

x_4	x_3
0	1
0	1
1	1
1	1

This table reduces to x_3 , for a 2 variable truth table with x_4 and x_3 as variables. Since all conditions are satisfied, terminate.

The implementation has only 2 modules in 2 levels, as shown in the Figure 6.20, while in the tree implementation [A.E.A. Almaini, J.F. Miller and L Xu, 1992], the synthesized network will have minimum of 3 modules in 2 levels as in Figure 6.21.

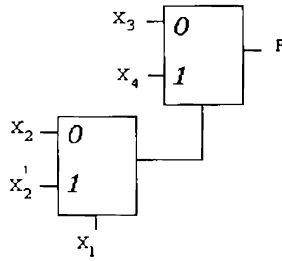


Figure 6.20: Exhaustive branched network implementation for
 $F = \Sigma(4, 7, 9, 10, 12, 13, 14, 15)$

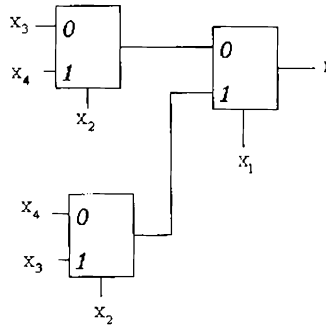


Figure 6.21: Tree implementation for
 $F = \Sigma(4, 7, 9, 10, 12, 13, 14, 15)$

It is noted that there is a reduction in number of modules which in turn reduces the area.

Example 2:

The implementation of a 4-variable function, $F = \sum (3, 5, 7, 9, 11, 15)$ has 3 modules using only 2 levels in this approach as shown in Figure 6.22, while the tree implementation [A.E.A. Almaini, J.F. Miller and L Xu, 1992] requires 3 modules in 3 levels as shown in Figure 6.23.

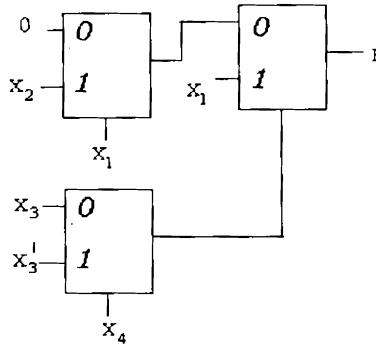


Figure 6.22: Exhaustive branched network implementation for $F = \sum (3, 5, 7, 9, 11, 15)$

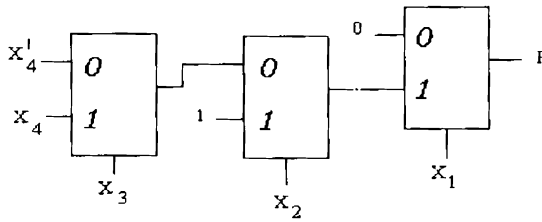


Figure 6.23: Tree implementation for $F = \sum (3, 5, 7, 9, 11, 15)$

Example 3:

The implementation of the 5-variable function, $F = \sum (3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$ is given in Figure 6.24.

The delivered network has only 4 modules in 2 levels, whereas the tree implementation requires more levels using at least 5 modules. One possible implementation is shown in Figure 6.25. This example clearly indicates the reduction in delay and hardware for a 5-variable function using the exhaustive branching technique.

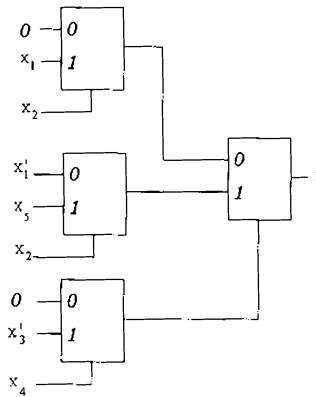


Figure 6.24: Exhaustive branched network implementation for $F = \sum (3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$

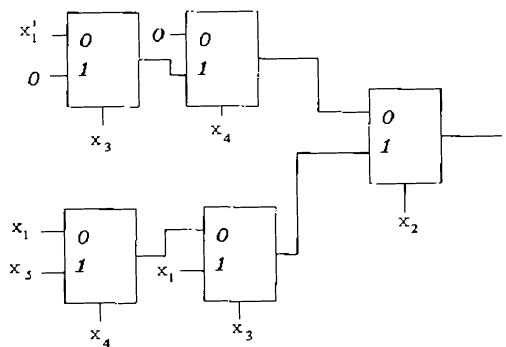


Figure 6.25: Tree implementation for $F = \sum (3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$

Table 6.33 shows the reduction in delay and/or hardware for certain functions. The reduction in number of modules required will lead to reduced power consumption.

Table 6.33: Comparison in terms of delay and hardware for standard implementation, tree implementation and exhaustive branched network implementation

Functions	Standard Implementation D / M	Tree Implementation D / M	Exhaustive Branched Implementation D / M
$F=\Sigma(1,2,4,7)$	3 / 7	2 / 3	2 / 2
$F=\Sigma(1,2,3,5,6,7,9,10,11,15)$	4 / 15	3 / 4	2 / 4
$F=\Sigma(4,7,9,10,12,13,14,15)$	4 / 15	2 / 3	2 / 2
$F=\Sigma(3,5,7,9,11,15)$	4 / 15	3 / 3	2 / 3
$F=\Sigma(3,7,8,15,19,23,24,26,27,31)$	5 / 31	3 / 5	2 / 4

D / M – Delay (number of levels) / Number of multiplexers

An algorithm for the synthesis of delay reduced multiplexer network is presented. By suitable selection of variables or functions as control inputs, the number of modules and/or delay is reduced. The reduction in number of modules results in reduced power consumption of the synthesized network. This technique can be used to design a BCD digit multiplier for a more regular implementation using multiplexers only.

6.6 Summary

A DFP MAC unit involves design of efficient DFP multiplier units. The speed of fixed point decimal multiplier in decimal DFP unit is increased by using DDDM. The latency for the multiplication of two n digit BCD operands is $\lceil (n/2)+1 \rceil$ cycles, and a new multiplication can begin every $n/2$ cycle. 34-digits multipliers are designed as a partitioned combination of 17-digits multipliers as well. It is seen that for the partitioned design, the speed is increased by 1.26 times compared to the iterative DDDM design. The area and delay analysis on different families of Xilinx, Altera, Actel and Quick logic FPGAs for DDDM reveals that efficient mapping is achieved when Xilinx FPGA devices are used.

Improved BCD digit multipliers for partial product generation are used in the iterative DFxP multiplier that employs novel RPS algorithm. The BCD digit multiplier design attains 8% savings in area and 18% savings in delay. The design leads to a more regular design, and does not require special registers for storing multiples of multiplicand. The design for a BCD digit decimal multiplier is extended to a Hex/Decimal digit multiplier. The comparison shows that the new design of Hex/Decimal multiplier has a reduction in delay of 16.52% with an extra hardware of 17.24%.

In the approach using RPS algorithm, partial products generated using BCD digit multipliers are accumulated from the least significant end in a column manner. The latency for the multiplication of two n -digit BCD operands is $(n+1)$ cycles, and a new multiplication can begin every n cycle. The simulation results show that the RPS design gives a reduction in delay compared to the single digit implementation for 7-digit \times 7-digit DFxP

multiplier. The iterative DFP multipliers are designed using floating point extensions of the iterative DFxP multiplier using DDDM and RPS algorithm. When multiplying two DFP numbers using RPS algorithm with n digit significands, the latency is $(n+2)$ cycles, and the initiation interval is $(n+1)$ cycles. The DFP design using DDDM for DFxP multiplication requires lesser number of clock cycles for DFP multiplication compared to RPS approach. The latency for the multiplication of DFP numbers with n -digit significands using DDDM is $\lceil (n/2) + 2 \rceil$ cycles, and a new multiplication can begin every $\lceil (n/2) + 1 \rceil$ cycle. These DFP multiplier designs are synthesized using Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library.

In the modified parallel DFxP multiplier design partial product accumulation is done using both row and column accumulations. The column accumulation approach gives a decrease of area and delay over the row accumulation. Parallel decimal multipliers for 7-digit, 16-digit and 34-digit are simulated and the results are tabulated. A 32-bit DFP parallel multiplier and compared with the iterative designs using DDDM and that using RPS algorithm.

The fused multiply-add unit uses parallel and iterative multipliers and a floating point adder unit. The DFP adder is implemented using ripple carry BCD adders, Kogge-Stone adders and 'reduced delay BCD adders'. Comparison of DFP adders shows that the 'reduced delay adder' is the fastest.

The improved reversible BCD adder implementations presented are compared with existing designs in literature. The performance comparison of carry select and hybrid BCD adders with conventional BCD adder is also done. When the number of digits is more than 25 the hybrid BCD adder can operate 5 times faster than conventional decimal adder using classical logic

gates. But for a reversible logic implementation using FRG gates the speed up factor of hybrid adder increases above 10 when the number of decimal digits is more than 25. It is shown that the optimum block size for a fixed block size hybrid BCD adder is given as $m_{opt} = \sqrt{0.5N}$ where N is the number of digits. Toffoli gate implementations are superior in terms of quantum cost, garbage count and gate count, compared to Fredkin gate implementations and are also suitable for implementations of Reed Muller expressions. The designs were verified using Reversible Circuit (RC) viewer/ analyzer.

A new reversible 4-bit Binary to BCD converter circuit that requires only one gate without garbage output is designed using a novel reversible 4×4 RPS gate or a partially reversible RPS gate. Six different implementations of the reversible (7, 4) Hamming code generation and error detection circuits are also presented. The new exhaustive branching algorithm to obtain reduced hardware and/or delay for logic functions using multiplexers can handle any number of variables for a completely specified logic function. A VLSI design using same modular building blocks can decrease system design and manufacturing cost for MAC implementation.

Chapter 7

Conclusion and Future Work

. Efficient designs are developed for high performance decimal floating point MAC unit as part of this research. The modified reversible BCD adder implementations presented are highly optimized in terms of number of reversible gates and garbage outputs. The new exhaustive branching algorithm obtains reduced hardware and/or delay for logic implementation using multiplexers for a VLSI design. The simulation results and the performance analysis show that the designs presented in this research are suitable for improving the performance of architectures for decimal computations. Hence these design techniques and circuits are dependable alternatives that could be used for high performance decimal processors. Suggestions for further work in this field are also presented.

7.1 Conclusion

Efficient design methods and architectures are developed for a high speed Decimal Floating Point (DFP) Multiply Accumulate (MAC) unit as part of this research. The decimal MAC unit has a multiplier fused with an adder module. The multiplier has to be an efficient, high speed multiplier for the MAC unit to achieve high performance. This research presents two novel techniques for iterative DFP multiplication. The first approach has a Decimal Fixed Point (DFxP) multiplier using a novel Double Digit Decimal Multiplication (DDDM) technique that performs two digit multiplications simultaneously. The speed to complete an n -digit \times n -digit multiplication is almost doubled compared to a single digit design at an expense of 50% increase in area for this design. The design was validated using lengths of 7-digit, 16-digit, and 34-digit multipliers that are required correspondingly for decimal32, decimal64, and decimal128 formats of DFP multiplications. In addition, 34-digit multiplier is designed as a partitioned combination of 17-digit multipliers. For the partitioned implementation, the speed is increased compared to the iterative DDDM implementation.

The second iterative approach does DFxP multiplication using a novel RPS algorithm. In this approach, partial products generated using BCD digit multipliers are accumulated from the least significant end in a column manner. This design leads to a more regular implementation, and does not require special registers for storing multiples of multiplicand. The simulation results show that the RPS design gives a reduction in delay compared to the single digit implementation for 7-digit \times 7-digit DFxP multiplier. This

iterative approach is suitable for high speed DFP multiplication since rounding can be initiated during the DFxP process.

A novel design for BCD digit multiplication that reduces the critical path delay and area is also presented in this research. The design for a BCD digit decimal multiplier is extended to a Hex/Decimal digit multiplier that gives either a decimal output or a hex output depending on the requirement. The comparison shows that the new design of Hex/Decimal multiplier has a reduction in delay compared to the existing design.

The iterative DFP multipliers using floating point extensions of the newly designed iterative DFxP multipliers use DDDM technique and RPS algorithm. A delay reduction is achieved using RPS algorithm because of the initiation of the rounding process during the DFxP multiplication. This parallelism decreases the worst case time period. The design using DDDM is more regular and occupies lesser area, but has more delay compared to the design using RPS algorithm.

The research also presents parallel DFP multiplier using modified parallel DFxP multiplier for significant digit multiplication. Parallel designs are adopted when latency and throughput are considered more important than area. The parallel DFP multiplier includes the floating point extensions of the parallel DFxP multiplier design. In this modified parallel DFxP multiplier design partial product accumulation is done using both row and column accumulations. The column accumulation approach gives a decrease of area and delay over the row accumulation.

Floating point MAC unit is designed for fused multiply-add operation with a single final rounding after add operation. The fused multiply-add unit uses parallel or iterative multipliers and a floating point adder unit. The DFP

adder is designed using ripple carry BCD adders, Kogge-Stone adders and 'reduced delay BCD adders'. Comparison of DFP adders shows that the 'reduced delay adder' achieves the highest speed.

In recent years, reversible logic has emerged as one of the most important approaches for power optimization. So, reversible logic is in demand in high-speed power aware circuits. The modified reversible BCD adder designs presented in this research are highly optimized in terms of number of reversible gates and garbage outputs. The comparison with existing designs in literature shows that the modified designs use least number of gates, produce least number of garbage outputs, and give least levels of delay. Speed of reversible design for carry select and hybrid BCD adders are compared with a conventional BCD adder. The results reveal that the hybrid BCD adder attains speed up over other two designs, for any input length in reversible implementation. This research also presents a reversible fault tolerant design using Fredkin gates (FRG) for conventional BCD adders, adders for QAD, and carry select BCD adders for multi-digit BCD addition. Toffoli Gate (TG) designs for multi-digit addition are also presented. The implementations using Toffoli gates give superior results in terms of quantum cost, garbage count and gate count, compared to Fredkin gate implementations.

A new reversible 4-bit Binary to BCD converter circuit that requires only one gate without garbage output is designed using a novel reversible 4×4 RPS gate as part of this research. A reversible BCD adder is also designed using this RPS gate that replaces the '6-correction circuit' and the 'final 4-bit binary adder'. In addition, a new partially reversible RPS gate that satisfies the reversibility criteria for BCD inputs is designed. This gate can further reduce the number of logical computations involved in BCD arithmetic reversible

circuits. The reversible implementations of BCD adder using combination of HNG-RPS (fully and partially) gates and using HNG gates are also presented in this research. A reduction in logical complexity is achieved by HNG-RPS design compared to the existing reversible BCD adder designs. It is also seen that the combination of HNG-RPS gates makes the BCD adder design more compact and reduces the number of garbage to a near optimal value.

Low power circuits designed in reversible logic for (7, 4) Hamming code generation and error detection circuits are presented in this research. The logic gates of a classical logic implementation of Hamming code generation and error detection circuits are replaced with reversible equivalents. Among these designs, the one using new 4×4 HCG is highly optimized in terms of number of reversible gates and/or garbage outputs. The use of new parity preserving HCG (PPHCG) gate provides a way of incorporating fault tolerance into reversible circuits with modest hardware overhead. Parity preservation by itself proves useful for ensuring the robustness of reversible logic circuits in their various application domains.

The new exhaustive branching algorithm suggested in this research obtains reduced hardware and/or delay for logic functions using multiplexers for a VLSI implementation. Exact functions are realized since the number of variables is also given as an input.

The simulation results and the performance analysis show that the design approaches and the architectures presented in this research are suitable for improving the performance of decimal computations. Hence these design techniques dependable alternatives that could be used for high performance decimal processors.

7.2 Suggestions for Future Work

Suggestions for further investigations in the field of architectures for decimal computations in continuation with the present work are listed below.

- The designs presented use BCD encoding for decimal representation which is the simplest and most popular code for decimal data. Alternatively, other decimal representations such as BCD Excess-3 (BCD XS3) representation may be used which may possibly allow more efficient decimal addition/subtraction.
- The use of specialized encodings such as 4221, 5211, 5421 codes that may improve the speed of computation in a decimal ALU can also be explored. If these specialized codes are suitable only for certain computations in the decimal ALU then the use of efficient intermediate encodings for processing decimal data may also be researched.
- Signed digit encodings that may reduce the number of partial product accumulations may be investigated. This may increase the speed of computations.
- The development of decimal/binary processor using these new designs may be explored.
- Future research may focus on incorporating more pipelining to improve the speed of multipliers and fused multiply-add computations.
- Custom layout design that takes advantage of high-speed and low-power circuit techniques may be developed.
- The reversible circuits presented in this research may be used to develop a decimal ALU for a reversible CPU.
- Further investigation into determining reversible implementations using logic synthesis methods [A. Agrawal and N. K. Jha, 2004],

[Dmitri Maslov, 2003], [P. Gupta, A. Agrawal, N.K. Jha, 2006], [G. Yang *et al.*, 2006] may be studied.

- Design modifications to reduce the total worst case delay may be investigated by varying the size of carry look-ahead blocks in a hybrid reversible adder.
- The investigation may focus on architectures for multipliers, comparators, etc using the fully and partially reversible RPS gates.
- Additionally, it is noted that there is a lack of simulation tools that support reversible gates, and this is most definitely an area worthy of attention.
- The exhaustive branching algorithm may be extended for the synthesis of incompletely specified functions. Research may be done for an alternative design of logic functions using different size multiplexers.

REFERENCES

- 1.A. Agrawal and N. K. Jha, "Synthesis of reversible logic," in Proc. Design Automation & Test in Europe Conf., Feb. 2004, pp. 21 384–21 385.
- 2.A.H. Aguirre and C.A.C. Coello "Using genetic programming and multiplexers for the synthesis of logic circuits", Engineering Optimization, Vol. 36, No. 4, August 2004, pp. 491-511
- 3.A.H. Aguirre, C.A.C. Coello and B.P. Buckles, "A genetic programming approach to logic function synthesis by means of multiplexers", Proceedings of First NASA/DOD workshop on Evolvable Hardware, IEEE Computer Society Press, Los Alanitos, California, July 1999, pp. 46-53
- 4.A.E.A. Almaini, J.F. Miller and L Xu, "Automated synthesis of digital multiplexer networks", IEE Proceedings-E, Vol. 139, No. 4, July 1992, pp. 329-334.
- 5.G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," IBM Journal of Research and Development, vol. 8, April 1964, pp. 87-101
- 6.Arazi, B., and Naccache, D.: 'Binary-to-decimal conversion based on the 282 1 by 5', Electron. Lett., 1992, 28, (23), pp. 2151–2152
- 7.J. V. Atanasoff, "Advent of Electronic Digital Computing," IEEE Annals of the History of Computing, vol. 6, no. 3, 1984, pp. 229-282
- 8.Md. M. H. Azad Khan, "Design of Full-adder With Reversible Gates", International Conference on Computer and Information Technology, Bangladesh, 2002, pp. 515-519

- 9.H. Md. H. Babu , A. R. Chowdhury, "Design of a compact reversible binary coded decimal adder circuit" *Journal of Systems Architecture* 52, 2006, pp. 272–282
- 10.H. Md. H. Babu and A. R. Chowdhury, "Design of a Reversible Binary Coded Decimal Adder by Using Reversible 4-bit Parallel Adder", *VLSI Design 2005*, Kolkata, India, Jan 2005, pp 255-260
- 11.H. Md. H Babu, Md. Rafiqul Islam, A. R. Chowdhury, S. M. Ali Chowdhury, "Synthesis of Full-adder Circuit Using Reversible Logic", *17th International Conference on VLSI design 2004*, India, 2004, pp. 757-760.
- 12.H. Md. H.Babu, Md. Rafiqul Islam, A. R. Chowdhury, S. M. Ali. Chowdhury, "On the Realization of Reversible Full-Adder Circuit", *International Conference on Computer and Information Technology*, Bangladesh, 2003, Vol. 2, pp. 880-883.
- 13.H. Md. H. Babu, Md. Rafiqul Islam, A. R. Chowdhury and S. M. Ali Chowdhury, "Reversible Logic Synthesis for Minimization of Fulladder Circuit", *IEEE Conference on Digital System Design 2003*, Euro-Micro'03, Turkey, 2003, pp. 50-54
- 14.F. Batista, "Decimal Data Type." *World Wide Web*. <http://www.python.org/dev/peps/pep-0327>, version 62268.
- 15.A.A. Bayrakci, A. Akkas "Reduced Delay BCD Adder" *IEEE International Conf. on. Application -specific Systems, Architectures and Processors*, ASAP, 2007, pp. 266-271
- 16.Bennett, C., "Logical Reversibility of Computation," *IBM Journal of Research and Development*, 17, 1973, 525-532.

- 17.G. Bohlender and T. Teufel, "BAP-SC: A Decimal Floating-Point Processor for Optimal Arithmetic", *Computer Arithmetic: Scientific Computation and Programming Languages*, Stuttgart, Germany: B. G. Teubner, 1987, pp. 31-58
- 18.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
- 19.J. J. Bradley, B. L. Stoffers, T. R. S. Jr., and M. A. Widen, "Simplified Decimal Multiplication by Stripping Leading Zeros," U.S. Patent, #4,615,016, Jun 1986
- 20.J.W.Bruce, M.A.Thornton, L.Shivakumariah, P.S.Kokate, X.Li, "Efficient Adder Circuits Based on a Conservative Logic Gate", *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, PA, USA, April 2002, pp 83-88
- 21.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, November 2001, pp. 1335– 1339
- 22.R. Campbell, "Makin' Numbers: Howard Aiken and the Computer, ch. "Aiken's First Machine: the IBM ASCC/Harvard Mark I", Cambridge, MA: MIT Press, 1999, pp. 31-63
- 23.E. O. Carbarnes, "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>.

24. Charles Tsen, Sonia González-Navarro, Michael Schulte, Brian Hickmann, Katherine Compton, "A Combined Decimal and Binary Floating-Point Multiplier," 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, 2009, pp.8-15
25. M. S. Cohen, T. E. Hull, and V. C. Hamacher, "CADAC: A Controlled-Precision Decimal Arithmetic Unit," IEEE Transactions on Computers, vol. C-32, April 1983, pp. 370-377
26. C. Cole, "The Remington Rand Univac LARC." World Wide Web. <http://www.computerhistory.info/Page4.dir/pages/LARC.dir/LARC.Cole.html>.
27. J. Copeland, "Colossus: Its Origins and Originators," IEEE Annals of the History of Computing, vol. 26, no. 4, , 2004, pp. 38-45
28. M. F. Cowlshaw, "Densely Packed Decimal Encoding", IEE Proceedings - Computers and Digital Techniques, vol. 149, May 2002, pp. 102-104
29. M. F. Cowlshaw, E. M. Schwarz, R. M. Smith, and C. F. Webb, "A Decimal Floating-Point Specification", 15th IEEE Symposium on Computer Arithmetic, IEEE Computer Society, July 2001, pp. 147-154
30. M. F. Cowlshaw, "Decimal Arithmetic FAQ." World Wide Web. <http://speleotrove.com/decimal/decifaq1.html>.
31. Dadda, L. Nannarelli, A. Politec. di Milano (2008) "A Variant of a Radix-10 Combinational Multiplier", IEEE International Symposium on Circuits and Systems, ISCAS May 2008, pp. 3370 – 3373
32. L. Dadda, 'Multioperand Parallel Decimal Adder: A Mixed Binary and BCD Approach', IEEE Transactions on Computers, Vol. 56, No. 9, Sept 2007

33. Dadda, L.: 'Some schemes for parallel multipliers', *Alta Frequenza*, 34, 1965, pp. 349–356
34. Decimal Arithmetic FAQ, 2009.04.02 IBM Corporation, <http://speleotrove.com/decimal/decifaq1.html>
35. Dmitri Maslov, "Reversible Logic Synthesis", PhD Dissertation, Computer Science Department, University of New Brunswick, Canada, Oct 2003
36. 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*, 51(1/2), 2007, pp. 217-228
37. R. Eissa, A. Mohamed, R. Samy, T. Eldeeb, Y. Farouk, M. Elkhoully, and H. Fahmy, "A Decimal Fully Parallel and Pipelined Floating Point Multiplier", *Asilomar Conference on Signals, Systems, and Computers*, 2008
38. 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, , November 2007, pp. 663-684
39. M. A. Erle, B. J. Hickmann and M. J. Schulte, "Decimal Floating-Point Multiplication", *IEEE Transactions on Computers*, Volume 58 , Issue 7, July 2009, pp. 902-916, ISSN:0018-9340
40. M. A. Erle, "Algorithms and Hardware designs for Decimal Multiplication", Ph. D Thesis, Nov 2008
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. Erle, M.A. Schwarz, E.M. Schulte, M.J, "Decimal multiplication with efficient partial product generation", 17th IEEE Symposium on Computer Arithmetic, 2005, ARITH-17 2005, June 2005, pp. 21- 28
43. M. A. Erle and M. J. Schulte, "Decimal Multiplication Via Carry-Save Addition," IEEE 14th International Conference on Application-specific Systems, Architectures and Processors, June 2003, pp. 348-358
44. ESA/390 Principles of Operation, ch. 8: Decimal Arithmetic Instructions. IBM, 2001.
45. H. A. H. Fahmy, R. Raafat, A. M. Abdel-Majeed, R. Samy, Tarek ElDeeb, Y. Farouk, "Energy and Delay Improvement via Decimal Floating Point Units," Proceedings on 19th IEEE Symposium on Computer Arithmetic, 2009, pp. 221-224
46. R. Feynman, "Quantum Mechanical Computers", Optical News, 1985, pp. 11-20
47. 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.
48. 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.
49. Free Software Foundation, "GNU C Compiler (GCC) 4.3 Release. "World Wide Web. <http://gcc.gnu.org/gcc-4.3>.
50. E. Fredkin, T. Toffoli, Conservative logic, International Journal of Theoretical Physics 21, 1982, pp. 219–253
51. A. Gough (Maintainer), "PERL BigInt Library." World Wide Web. <http://dev.perl.org/perl6/pdd/pdd14 bigint.html>, version 1.5

- 52.H. H. Goldstine and A. Goldstine, "The Electronic Numerical Integrator and Computer (ENIAC)", IEEE Annals of the History of Computing, vol. 18, no. 1, 1996, pp. 10-16
- 53.R.K. Gorai and A. Pal, "Automated synthesis of combinational circuits by cascade networks of multiplexers", IEE Proceedings-E, Vol. 137. No. 2, March 1990, pp. 164-170
- 54.P. Gupta, A. Agrawal, N.K. Jha, "An algorithm for synthesis of reversible logic circuits", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems", Volume: 25, Issue 11, Nov. 2006, pp. 2317-2330
- 55.M. Haghparast and K. Navi, "A Novel Reversible BCD Adder For Nanotechnology Based Systems", American Journal of Applied Sciences 5 (3): 2008, pp. 282-288, ISSN 1546-9239
- 56.R. Head, "Univac: a Philadelphia Story", IEEE Annals of the History of Computing, vol. 23, no. 3, 2001, pp. 60-63
- 57.Hickmann, B. Krioukov, A. Schulte, M. Erle, M. A, "Parallel IEEE P754 decimal floating-point multiplier", Proceedings of the IEEE International Conference on Computer Design 2007, October 2007, pp. 296-303
- 58.R. L. Hoffman and T. L. Schardt, "Packed Decimal Multiply Algorithm," IBM Technical Disclosure Bulletin, vol. 18, October 1975, pp. 1562-1563
- 59.T. E. Hull, M. S. Cohen, and C. B. Hall, "Specifications for a Variable Precision Arithmetic Coprocessor", 10th Symposium on Computer Arithmetic, IEEE, IEEE Computer Society, May 1991, pp. 127-131
- 60.IA-32 Intel Architecture Software Developer's Manual, vol. 2: Instruction Set Reference, ch. 3: Instruction Set Reference. Intel, 2001

61. IEEE Working Group of the Microprocessor Standards Subcommittee, IEEE Standard for Floating-Point Arithmetic. New York: The Institute of Electrical and Electronics Engineers, 2008
62. IEEE 754-2008 floating point standards, <http://www.rapidsharemegaupload.com/ieee-754-2008-rapids.htm>
63. I. P. S. of Japan, "Historic Computers in Japan: [NEC] NEAC 2201." [www.http://museum.ipsj.or.jp/en/computer/dawn/0018.html](http://www.museum.ipsj.or.jp/en/computer/dawn/0018.html).
64. Jaberipur, G.; Kaivani, A, "Binary-coded decimal digit multipliers", Computers & Digital Techniques, IET Volume 1, Issue 4, July 2007, pp. 377 – 381
65. JTC 1/SC 22/WG 4, ISO/IEC 1989: Information technology-Programming languages - COBOL. New York: American National Standards Institute, first ed., December 2002
66. G. Kane, PA-RISC 2.0 Architecture, ch. 7: Instruction Descriptions. Prentice Hall, 1996
67. R. D. Kenney and M. J. Schulte, 'High-Speed Multi-operand Decimal Adders' IEEE Transactions on Computers, vol. 54, No. 8, Aug 2005, pp. 953-963
68. R. D. Kenney, M. J. Schulte and M. A. Erle, "A High-Frequency Decimal Multiplier," IEEE 14th International IEEE international conference on Computer Design (ICCD'04, Oct 2004), pp. 22-29
69. D. E. Knuth, "The IBM 650: An Appreciation from the Field", IEEE Annals of the history of Computing, vol. 8, no. 1, 1986, pp. 50-55
70. Kogge, P. & Stone, H. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". IEEE Transactions on Computers, 1973, C-22, pp. 783-791

- 71.R. Landauer, "Irreversibility and Heat Generation in the Computational Process", IBM Journal of Research Development, 5, 1961, pp. 183-191
- 72.T. Lang and A. Nannarelli, "A radix-10 combinational multiplier" 40th Asilomar Conference on Signals, Systems, and Computers, Oct. 2006, pp. 313-317
- 73.R. H. Larson, "High Speed Multiply Using Four Input Carry Save Adder," IBM Technical Disclosure Bulletin, December 1973, pp. 2053-2054
- 74.T. Leser and M. Romanelli, "Programming and Coding for ORDVAC", October 1956. http://www.bitsavers.org/pdf/ordvac/ORDVAC_programming,Oct56.pdf
- 75.G. E. Moore, "Cramming More Components onto Integrated Circuits", Electronics, vol. 38, April 1965, pp. 114-117
- 76.Motorola, "Motorola M68000 Family Programmers Reference Manual." World Wide Web, 1992
<http://www.freescale.com/files/archives/doc/refmanual/M68000PRM.pdf>
- 77.Nation Master Encyclopedia, "Z1 (Computer)." World Wide Web.
[http://www.nationmaster.com/encyclopedia/Z1-\(computer\)](http://www.nationmaster.com/encyclopedia/Z1-(computer)).
- 78.T. Ohtsuki, Y. Oshima, S. Ishikawa, K. Yabe, and M. Fukuta, "Apparatus for Decimal Multiplication," U.S. Patent, #4,677,583, Jun 1987
- 79.A. Pal, "An algorithm for optimal logic design using multiplexers", IEEE Transactions on Computers, Vol. 35, No. 8, August 1986, pp. 755-757
- 80.B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs, Oxford University Press, 2000.
- 81.B. Parhami, "Fault Tolerant Reversible Circuits" Proc. 40th Asilomar Conf. Signals, Systems, and Computers, Pacific Grove, CA, Oct. 2006.

- 82.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, 1982, pp. 327-341
- 83.Rhyne, V.T.: 'Serial binary-to-decimal and decimal-to-binary conversion', *IEEE Trans. Comput.*, 19, (9), 1970, pp. 808-812
- 84.M. Schmookler and A. Weinberger, "High Speed Decimal Addition," *IEEE Trans. Computers*, vol. 20, no. 8, Aug. 1971, pp. 862-867
- 85.Schmookler, M.: 'High-speed binary-to-decimal conversion', *IEEE Trans. Comput*, 17, (5), 1968, pp. 506-508
- 86.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), 2009.
- 87.B. Shirazi, D.Y. Yun, and C.N. Zhang, "RBCD: Redundant Binary Coded Decimal Adder," *IEE Proc.—Part E*, vol. 136, no. 2, Mar. 1989
- 88.B. Shirazi, D.Y. Yun, and C.N. Zhang, "VLSI Designs for Redundant Binary-Coded Decimal Addition," *Proc. Seventh Ann. Int'l Conf. Computers and Comm.*, Mar. 1988, pp. 52-56
- 89.SilMinds, "Decimal Floating Point Arithmetic IP Cores Family." World Wide Web, 2008. <http://www.silminds.com/resources/SilMinds-DFPA-IP-CoresFamily.pdf>.
- 90.N. Stern, "From ENIAC to UNIVAC - An Appraisal of the Eckert-Mauchly Computers. Bedford, MA: Digital Press, 1981.
- 91.Sun Microsystems, "BigDecimal Java Class." World Wide Web. <http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html>.

92. A. Tsang and M. Olschanowsky, .A Study of DataBase 2 Customer Queries,. IBM Technical Report TR 03.413, IBM Santa Teresa Laboratory, San Jose, CA, April 1991.
- 93.H. Thapliyal, ,H. R. Arabnia, R. Bajpai ,K. K. Sharma, “Partial Reversible Gates (PRG) for Reversible BCD Arithmetic” International Conference on Computer Design (CDES'07), Las Vegas, U.S.A, June 2007, pp. 90-91
- 94.H. Thapliyal, S. Kotiyal and M.B Srinivas, “Novel BCD Adders and their Reversible Logic Implementation for IEEE 754r Format”, VLSI Design 2006, Hyderabad, India, Jan 4-7, 2006, pp. 387-392
- 95.H. Thapliyal and M.B Srinivas, “A Novel Reversible TSG Gate and Its Application for Designing Reversible Carry Look-Ahead and Other Adder Architectures”, Tenth Asia-Pacific Computer Systems Architecture Conference, Singapore, Oct 24 - 26, 2005
- 96.J. Thompson, M. J. Schulte, and N. Karra. “A 64-bit decimal floating-point adder”, Proceedings of the IEEE Computer Society Annual Symposium on VLSI, Lafayette, LA, Feb 2004, pp. 297-298
- 97.J. E. Thornton, “The CDC 6600 Project”, IEEE Annals of the History of Computing, vol. 2, no. 4, 1980, pp. 338-348
- 98.T. Toffoli., “Reversible Computing”, Tech memo MIT/LCS/TM-151, MIT Lab for Computer Science, 1980.
- 99.C. Tsen, S. Gonzalez-Navarro, and M. J. Schulte, “Hardware design of a binary integer decimal-based floating-point adder”, Proceedings of the IEEE International Conference on Computer Design, 2007.
- 100.Ueda, T.: “Decimal multiplying assembly and multiply module” .U.S. Patent 5379245, January 1995

101. A. Vazquez, E. Antelo and P. Montuschi (2007) "A New Family of High-Performance Parallel Decimal Multipliers", 18th IEEE Symposium on Computer Architecture, June 2007, pp. 195-204
102. Wallace, C.S.: 'A suggestion for fast multiplier', IEEE Trans. Electron. Comput., 13, 1964, pp. 14-17
103. L.K. Wang, M. J. Schulte, J. D. Thompson, and N. Jairam, "Hardware designs for decimal floating-point addition and related operations", IEEE Transactions on Computers, 58(3), March 2009.
104. L.-K. Wang and M. J. Schulte. Decimal floating-point adder and multifunction unit with injection-based rounding. In Proceedings of the 18th IEEE Symposium on Computer Arithmetic, Montpellier, France, June 2007.
105. 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>.
106. C. F. Webb. IBM z10: The next-generation mainframe microprocessor. IEEE Micro, 28(2):19-29, March/April 2008.
107. M. R. Williams, "The Origins, Uses, and Fate of the EDVAC", IEEE Annals of the History of Computing, vol. 15, no. 1, 1993, pp. 22-38
108. M. V. Wilkes, "Arithmetic on the EDSAC" IEEE Annals of the History of Computing, vol. 19, no. 1, 1997, pp. 13-15
109. G. Yang, Fei Xie, Xiaoyu Song, Hung, W.N.N., Perkowski, M.A., "A constructive Algorithm for Reversible Logic synthesis" IEEE Congress on Evolutionary Computation, July 2006, pp. 2416- 2421

LIST OF PUBLICATIONS OF THE AUTHOR

Book Chapters and International Journals:

- [1] Rekha K James, K. Poulouse Jacob and Sreela Sasi “Reversible Binary Coded Decimal Adders using Toffoli Gates”, **Advances in Computational Algorithms and Data Analysis, Springer, Netherlands, Book Series - Lecture Notes in Electrical Engineering**, ISSN 1876-1100, Volume 14, Book DOI 10.1007/978-1-4020-8919-0, Copyright 2008 ISBN 978-1-4020-8918-3 (Print) 978-1-4020-8919-0 (Online) DOI 10.1007/978-1-4020-8919-0_9, September, 2008, pp. 117-131
- [2] Rekha K James, K. Poulouse Jacob and Sreela Sasi, “Fast Reversible Binary Coded Decimal Adders”, **International Journal of Electrical, & Electronics Engineering (IJEEE)**, online journal, Vol. 3, No. 4, Fall 2008, pp. 254 – 266

Journal Papers Communicated:

- [3] Rekha K. James, K. Poulouse Jacob, Sreela Sasi, “Iterative and Parallel Decimal Floating Point Multipliers”, communicated to **IET Computers and Digital Techniques**, previously published as IEE Proceedings Computers and Digital Techniques.
- [4] Rekha K. James, K. Poulouse Jacob, Sreela Sasi, “Design of Compact Reversible Decimal Adder using RPS Gates”, Communicated to **IETE Journal of Research**

International conferences:

- [5] Rekha K. James, K. Poulouse Jacob, Sreela Sasi, “High Performance, Low Latency Double Digit Decimal Multiplier on ASIC and FPGA”, **International Symposium on Innovations in Natural Computing 2009**, Dec 12-13 2009, Cochin, India, In conjunction with World Congress in Nature and Biologically Inspired Computing (NaBIC2009), pp. 1445-1450
- [6] Rekha K. James, K. Poulouse Jacob, Sreela Sasi, “Double Digit Decimal Multiplier on XILINX FPGA”, **International Conference on Embedded Systems and Applications - ESA'09**: July 13-16, 2009, USA, pp. 47-53
- [7] Rekha K. James, K. Poulouse Jacob, Sreela Sasi, “An Alternate Approach to Enhance Parallel Decimal Multiplier Performance”, **Thirteenth IEEE VLSI Design and Test Symposium**, July 8-10, 2009, Bangalore, India, pp. 86-95
- [8] Rekha K. James, K. Poulouse Jacob, Sreela Sasi, “Performance Analysis of Double Digit Decimal Multiplier on Various FPGA Logic Families”, **V Southern Programmable Logic Conference (SPL 2009)**, Sao Carlos, Brazil, April 1-3, 2009, pp. 165-170
- [9] Rekha K. James, Shahana T K, K. Poulouse Jacob, Sreela Sasi, “Fixed Point Decimal Multiplication using RPS Algorithm”, **2008 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2008)**, Sydney, Australia, Dec 10 –12, 2008, ISBN:978-0-7695-3471-8, pp. 343-350
- [10] Rekha K. James, Shahana T K, K. Poulouse Jacob, Sreela Sasi, “Decimal Multiplication using compact BCD Multiplier”, **International Conference on Electronic Design (ICED 2008), IEEE Malaysia Chapter**, Penang Malaysia, Dec1–3,2008, Page(s):1-6, Digital Object Identifier 10.1109/ICED.2008.4786744

- [11] Rekha K. James, Shahana T K, K. Poullose Jacob, Sreela Sasi, "Quick Addition of Decimals using Reversible Conservative Logic", **15th International Conference on Advanced Computing & Communication (ADCOM 2007)**, ACS, 18- 21 December 2007, IIT Guwahati, India, pp. 191-196
- [12] Rekha K. James, Shahana T K, K. Poullose Jacob, Sreela Sasi, "A New Look at Reversible Logic Implementation of Decimal Adder", **IEEE International Symposium on System-on-Chip Tampere (SCO 2007)**, Finland Nov 20-22, 2007, page(s): 1-4, ISSN: 07EX1846, ISBN: 978-1-4244-1367-6
- [13] Rekha K. James, Shahana T K, K. Poullose Jacob, Sreela Sasi, "Fault Tolerant Error Coding and Detection using Reversible Gates", **IEEE TENCON 2007**, Oct 30 - Nov 2, 2007 Taiwan, page(s): 1-4, ISBN: 978-1-4244-1272-3
- [14] Rekha K. James, Shahana T K, K. Poullose Jacob, Sreela Sasi, "Performance Analysis of Reversible Fast Decimal Adders", **Lecture notes on International Conference on Computer Science and Applications (ICCSA'07)**-World Congress on Engineering and Computer Science 2007 (WCECS 2007).San Francisco, USA Oct 24-26 2007 ICCSA, pp. 234-239
- [15] Rekha K. James, Shahana T K, K. Poullose Jacob, Sreela Sasi, "Improved Reversible Logic Implementation of Decimal Adder" **11th IEEE VLSI Design and Test Symposium**, Aug 8-11 2007, India, pp. 70
- [16] Rekha K. James, Shahana T K, K. Poullose Jacob, Sreela Sasi, "Delay-Reduced Combinational Logic Synthesis using Multiplexers" **ESA'06 – The 2006 International Conference on Embedded Systems & Applications**, Las Vegas, Nevada, USA, June 26-29, 2006, pp. 105-110

IEEE Standard for Floating-Point Arithmetic

IMPORTANT NOTICE: *This standard is not intended to assure safety, security, health, or environmental protection in all circumstances. Implementers of the standard are responsible for determining appropriate safety, security, environmental, and health practices or regulatory requirements.*

This IEEE document is made available for use subject to important notices and legal disclaimers. These notices and disclaimers appear in all publications containing this document and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Documents". They can also be obtained on request from IEEE or viewed at <http://standards.ieee.org/IPR/disclaimers.html>.

1. Overview

1.1 Scope

This standard specifies formats and methods for floating-point arithmetic in computer systems—standard and extended functions with single, double, extended, and extendable precision—and recommends formats for data interchange. Exception conditions are defined and standard handling of these conditions is specified.

1.2 Purpose

This standard provides a method for computation with floating-point numbers that will yield the same result whether the processing is done in hardware, software, or a combination of the two. The results of the computation will be identical, independent of implementation, given the same input data. Errors, and error conditions, in the mathematical processing will be reported in a consistent manner regardless of implementation.

1.3 Inclusions

This standard specifies:

- Formats for binary and decimal floating-point data, for computation and data interchange.
- Addition, subtraction, multiplication, division, fused multiply add, square root, compare, and other operations.
- Conversions between integer and floating-point formats.
- Conversions between different floating-point formats.
- Conversions between floating-point formats and external representations as character sequences.
- Floating-point exceptions and their handling, including data that are not numbers (NaNs).

1.4 Exclusions

This standard does not specify:

- Formats of integers.
- Interpretation of the sign and significand fields of NaNs.

1.5 Programming environment considerations

This standard specifies floating-point arithmetic in two radices, 2 and 10. A programming environment may conform to this standard in one radix or in both.

This standard does not define all aspects of a conforming programming environment. Such behavior should be defined by a programming language definition supporting this standard, if available, and otherwise by a particular implementation. Some programming language specifications might permit some behaviors to be defined by the implementation.

Language-defined behavior should be defined by a programming language standard supporting this standard. Then all implementations conforming both to this floating-point standard and to that language standard behave identically with respect to such language-defined behaviors. Standards for languages intended to reproduce results exactly on all platforms are expected to specify behavior more tightly than do standards for languages intended to maximize performance on every platform.

Because this standard requires facilities that are not currently available in common programming languages, the standards for such languages might not be able to fully conform to this standard if they are no longer being revised. If the language can be extended by a function library or class or package to provide a conforming environment, then that extension should define all the language-defined behaviors that would normally be defined by a language standard.

Implementation-defined behavior is defined by a specific implementation of a specific programming environment conforming to this standard. Implementations define behaviors not specified by this standard nor by any relevant programming language standard or programming language extension.

Conformance to this standard is a property of a specific implementation of a specific programming environment, rather than of a language specification.

However a language standard could also be said to conform to this standard if it were constructed so that every conforming implementation of that language also conformed automatically to this standard.

1.6 Word usage

In this standard three words are used to differentiate between different levels of requirements and optionality, as follows:

- **may** indicates a course of action permissible within the limits of the standard with no implied preference (“may” means “is permitted to”)
- **shall** indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (“shall” means “is required to”)
- **should** indicates that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (“should” means “is recommended to”).

Further:

- **might** indicates the possibility of a situation that could occur, with no implication of the likelihood of that situation (“might” means “could possibly”)
- **see** followed by a number is a cross-reference to the clause or subclause of this standard identified by that number
- **NOTE** introduces text that is informative (that is, is not a requirement of this standard).

2. Definitions, abbreviations, and acronyms

2.1 Definitions

For the purposes of this standard, the following terms and definitions apply.

2.1.1 applicable attribute: The value of an attribute governing a particular instance of execution of a computational operation of this standard. Languages specify how the applicable attribute is determined.

2.1.2 arithmetic format: A floating-point format that can be used to represent floating-point operands or results for the operations of this standard.

2.1.3 attribute: An implicit parameter to operations of this standard, which a user might statically set in a programming language by specifying a constant value. The term attribute might refer to the parameter (as in “rounding-direction attribute”) or its value (as in “roundTowardZero attribute”).

2.1.4 basic format: One of five floating-point representations, three binary and two decimal, whose encodings are specified by this standard, and which can be used for arithmetic. One or more of the basic formats is implemented in any conforming implementation.

2.1.5 biased exponent: The sum of the exponent and a constant (bias) chosen to make the biased exponent’s range nonnegative.

2.1.6 binary floating-point number: A floating-point number with radix two.

2.1.7 block: A language-defined syntactic unit for which a user can specify attributes. Language standards might provide means for users to specify attributes for blocks of varying scopes, even as large as an entire program and as small as a single operation.

2.1.8 canonical encoding: The preferred encoding of a floating-point representation in a format. Applied to declets, significands of finite numbers, infinities, and NaNs, especially in decimal formats.

2.1.9 canonicalized number: A floating-point number whose encoding (if there is one) is canonical.

2.1.10 cohort: The set of all floating-point representations that represent a given floating-point number in a given floating-point format. In this context -0 and $+0$ are considered distinct and are in different cohorts.

2.1.11 computational operation: An operation that can signal floating-point exceptions, or that produces floating-point results, or that produces integer results by rounding them to fit destination formats according to a rounding direction rule. Comparisons are computational operations.

2.1.12 correct rounding: This standard’s method of converting an infinitely precise result to a floating-point number, as determined by the applicable rounding direction. A floating-point number so obtained is said to be correctly rounded.

2.1.13 decimal floating-point number: A floating-point number with radix ten.

2.1.14 declet: An encoding of three decimal digits into ten bits using the densely-packed-decimal encoding scheme. Of the 1024 possible declets, 1000 canonical declets are produced by computational operations, while 24 non-canonical declets are not produced by computational operations, but are accepted in operands.

2.1.15 denormalized number: *See: subnormal number.*

2.1.16 destination: The location for the result of an operation upon one or more operands. A destination might be either explicitly designated by the user or implicitly supplied by the system (for example, intermediate results in subexpressions or arguments for procedures). Some languages place the results of intermediate calculations in destinations beyond the user’s control; nonetheless, this standard defines the result of an operation in terms of that destination’s format and the operands’ values.

2.1.17 dynamic mode: An optional method of dynamically setting attributes by means of operations of this standard to set, test, save, and restore them.

2.1.18 exception: An event that occurs when an operation on some particular operands has no outcome suitable for every reasonable application. That operation might signal one or more exceptions by invoking the default or, if explicitly requested, a language-defined alternate handling. Note that *event*, *exception*, and *signal* are defined in diverse ways in different programming environments.

- 2.1.19 exponent:** The component of a finite floating-point representation that signifies the integer power to which the radix is raised in determining the value of that floating-point representation. The exponent e is used when the significand is regarded as an integer digit and fraction field, and the exponent q is used when the significand is regarded as an integer; $e = q + p - 1$ where p is the precision of the format in digits.
- 2.1.20 extendable precision format:** A format with precision and range that are defined under user control.
- 2.1.21 extended precision format:** A format that extends a supported basic format by providing wider precision and range.
- 2.1.22 external character sequence:** A representation of a floating-point datum as a sequence of characters, including the character sequences in floating-point literals in program text.
- 2.1.23 flag:** See: status flag.
- 2.1.24 floating-point datum:** A floating-point number or non-number (NaN) that is representable in a floating-point format. In this standard, a floating-point datum is not always distinguished from its representation or encoding.
- 2.1.25 floating-point number:** A finite or infinite number that is representable in a floating-point format. A floating-point datum that is not a NaN. All floating-point numbers, including zeros and infinities, are signed.
- 2.1.26 floating-point representation:** An unencoded member of a floating-point format, representing a finite number, a signed infinity, a quiet NaN, or a signaling NaN. A representation of a finite number has three components: a sign, an exponent, and a significand; its numerical value is the signed product of its significand and its radix raised to the power of its exponent.
- 2.1.27 format:** A set of representations of numerical values and symbols, perhaps accompanied by an encoding.
- 2.1.28 fusedMultiplyAdd:** The operation `fusedMultiplyAdd(x, y, z)` computes $(x \times y) + z$ as if with unbounded range and precision, rounding only once to the destination format.
- 2.1.29 generic operation:** An operation of this standard that can take operands of various formats, for which the formats of the results might depend on the formats of the operands.
- 2.1.30 homogeneous operation:** An operation of this standard that takes operands and returns results all in the same format.
- 2.1.31 implementation-defined:** Behavior defined by a specific implementation of a specific programming environment conforming to this standard.
- 2.1.32 integer format:** A format not defined in this standard that represents a subset of the integers and perhaps additional values representing infinities, NaNs, or negative zeros.
- 2.1.33 interchange format:** A format that has a specific fixed-width encoding defined in this standard.
- 2.1.34 language-defined:** Behavior defined by a programming language standard supporting this standard.
- 2.1.35 NaN:** not a number—a symbolic floating-point datum. There are two kinds of NaN representations: quiet and signaling. Most operations propagate quiet NaNs without signaling exceptions, and signal the invalid operation exception when given a signaling NaN operand.
- 2.1.36 narrower/wider format:** If the set of floating-point numbers of one format is a proper subset of another format, the first is called narrower and the second wider. The wider format might have greater precision, range, or (usually) both.
- 2.1.37 non-computational operation:** An operation that is not computational.
- 2.1.38 normal number:** For a particular format, a finite non-zero floating-point number with magnitude greater than or equal to a minimum $b^{e_{min}}$ value, where b is the radix. Normal numbers can use the full precision available in a format. In this standard, zero is neither normal nor subnormal.
- 2.1.39 not a number:** See: NaN.
- 2.1.40 payload:** The diagnostic information contained in a NaN, encoded in part of its trailing significand field.

2.1.41 precision: The maximum number p of significant digits that can be represented in a format, or the number of digits to that a result is rounded.

2.1.42 preferred exponent: For the result of a decimal operation, the value of the exponent q which preserves the quantum of the operands when the result is exact.

2.1.43 preferredWidth method: A method used by a programming language to determine the destination formats for generic operations and functions. Some **preferredWidth** methods take advantage of the extra range and precision of wide formats without requiring the program to be written with explicit conversions.

2.1.44 quantum: The quantum of a finite floating-point representation is the value of a unit in the last position of its significand. This is equal to the radix raised to the exponent q , which is used when the significand is regarded as an integer.

2.1.45 quiet operation: An operation that never signals any floating-point exception.

2.1.46 radix: The base for the representation of binary or decimal floating-point numbers, two or ten.

2.1.47 result: The floating-point representation or encoding that is delivered to the destination.

2.1.48 signal: When an operation on some particular operands has no outcome suitable for every reasonable application, that operation might signal one or more exceptions by invoking the default handling or, if explicitly requested, a language-defined alternate handling selected by the user.

2.1.49 significand: A component of a finite floating-point number containing its significant digits. The significand can be thought of as an integer, a fraction, or some other fixed-point form, by choosing an appropriate exponent offset. A decimal or subnormal binary significand can also contain leading zeros.

2.1.50 status flag: A variable that can take two states, raised or lowered. When raised, a status flag might convey additional system-dependent information, possibly inaccessible to some users. The operations of this standard, when exceptional, can as a side effect raise some of the following status flags: inexact, underflow, overflow, divideByZero, and invalid operation.

2.1.51 subnormal number: In a particular format, a non-zero floating-point number with magnitude less than the magnitude of that format's smallest normal number. A subnormal number does not use the full precision available to normal numbers of the same format.

2.1.52 supported format: A floating-point format provided in the programming environment and implemented in conformance with the requirements of this standard. Thus, a programming environment might provide more formats than it supports, as only those implemented in accordance with the standard are said to be supported. Also, an integer format is said to be supported if conversions between that format and supported floating-point formats are provided in conformance with this standard.

2.1.53 trailing significand field: A component of an encoded binary or decimal floating-point format containing all the significand digits except the leading digit. In these formats, the biased exponent or combination field encodes or implies the leading significand digit.

2.1.54 user: Any person, hardware, or program not itself specified by this standard, having access to and controlling those operations of the programming environment specified in this standard.

2.1.55 width of an operation: The format of the destination of an operation specified by this standard; it will be one of the supported formats provided by an implementation in conformance to this standard.

2.2 Abbreviations and acronyms

LSB	least significant bit
MSB	most significant bit
NaN	not a number
qNaN	quiet NaN
sNaN	signaling NaN

3. Floating-point formats

3.1 Overview

3.1.1 Formats

This clause defines floating-point formats, which are used to represent a finite subset of real numbers (see 3.2). Formats are characterized by their radix, precision, and exponent range, and each format can represent a unique set of floating-point data (see 3.3).

All formats can be supported as **arithmetic formats**; that is, they may be used to represent floating-point operands or results for the operations described in later clauses of this standard.

Specific fixed-width encodings for binary and decimal formats are defined in this clause for a subset of the formats (see 3.4 and 3.5). These **interchange formats** are identified by their size (see 3.6) and can be used for the exchange of floating-point data between implementations.

Five **basic formats** are defined in this clause:

- Three binary formats, with encodings in lengths of 32, 64, and 128 bits.
- Two decimal formats, with encodings in lengths of 64 and 128 bits.

Additional arithmetic formats are recommended for extending these basic formats (see 3.7).

The choice of which of this standard's formats to support is language-defined or, if the relevant language standard is silent or defers to the implementation, implementation-defined. The names used for formats in this standard are not necessarily those used in programming environments.

3.1.2 Conformance

A conforming implementation of any supported format shall provide means to initialize that format and shall provide conversions between that format and all other supported formats.

A conforming implementation of a supported arithmetic format shall provide all the operations of this standard defined in Clause 5, for that format.

A conforming implementation of a supported interchange format shall provide means to read and write that format using a specific encoding defined in this clause, for that format.

A programming environment conforms to this standard, in a particular radix, by implementing one or more of the basic formats of that radix as both a supported arithmetic format and a supported interchange format.

3.2 Specification levels

Floating-point arithmetic is a systematic approximation of real arithmetic, as illustrated in Table 3.1. Floating-point arithmetic can only represent a finite subset of the continuum of real numbers. Consequently certain properties of real arithmetic, such as associativity of addition, do not always hold for floating-point arithmetic.

Table 3.1—Relationships between different specification levels for a particular format

Level 1	$\{-\infty \dots 0 \dots +\infty\}$	Extended real numbers.
many-to-one ↓	<i>rounding</i>	↑ projection (except for NaN)
Level 2	$\{-\infty \dots -0\} \cup \{+0 \dots +\infty\} \cup \text{NaN}$	Floating-point data—an algebraically closed system.
one-to-many ↓	<i>representation specification</i>	↑ many-to-one
Level 3	$(\text{sign}, \text{exponent}, \text{significand}) \cup \{-\infty, +\infty\} \cup \text{qNaN} \cup \text{sNaN}$	Representations of floating-point data.
one-to-many ↓	<i>encoding for representations of floating-point data</i>	↑ many-to-one
Level 4	0111000...	Bit strings.

The mathematical structure underpinning the arithmetic in this standard is the extended reals, that is, the set of real numbers together with positive and negative infinity. For a given format, the process of *rounding* (see 4) maps an extended real number to a *floating-point number* included in that format. A *floating-point datum*, which can be a signed zero, finite non-zero number, signed infinity, or a NaN (not-a-number), can be mapped to one or more *representations of floating-point data* in a format.

The representations of floating-point data in a format consist of:

- triples $(\text{sign}, \text{exponent}, \text{significand})$; in radix b , the floating-point number represented by a triple is $(-1)^{\text{sign}} \times b^{\text{exponent}} \times \text{significand}$
- $+\infty$, $-\infty$.
- qNaN (quiet), sNaN (signaling).

An *encoding* maps a representation of a floating-point datum to a bit string. An encoding might map some representations of floating-point data to more than one bit string. Multiple NaN bit strings should be used to store retrospective diagnostic information (see 6.2).

3.3 Sets of floating-point data

This subclause specifies the sets of floating-point data representable within all floating-point formats; the encodings for specific representations of floating-point data in interchange formats are defined in 3.4 and 3.5, and the parameters for interchange formats are defined in 3.6.

The set of finite floating-point numbers representable within a particular format is determined by the following integer parameters:

- b = the radix, 2 or 10
 - p = the number of digits in the significand (precision)
 - e_{max} = the maximum exponent e
 - e_{min} = the minimum exponent e
- e_{min} shall be $1 - e_{\text{max}}$ for all formats.

The values of these parameters for each basic format are given in Table 3.2, in which each format is identified by its radix and the number of bits in its encoding. Constraints on these parameters for extended and extendable precision formats are given in 3.7.

Within each format, the following floating-point data shall be represented:

- Signed zero and non-zero floating-point numbers of the form $(-1)^s \times b^e \times m$, where
 - s is 0 or 1.
 - e is any integer $e_{min} \leq e \leq e_{max}$.
 - m is a number represented by a digit string of the form $d_0 \cdot d_1 d_2 \dots d_{p-1}$ where d_i is an integer digit $0 \leq d_i < b$ (therefore $0 \leq m < b$).
- Two infinities, $+\infty$ and $-\infty$.
- Two NaNs, qNaN (quiet) and sNaN (signaling).

These are the only floating-point data represented.

In the foregoing description, the significand m is viewed in a scientific form, with the radix point immediately following the first digit. It is also convenient for some purposes to view the significand as an integer, in which case the finite floating-point numbers are described thus:

- Signed zero and non-zero floating-point numbers of the form $(-1)^s \times b^q \times c$, where
 - s is 0 or 1.
 - q is any integer $e_{min} \leq q + p - 1 \leq e_{max}$.
 - c is a number represented by a digit string of the form $d_0 d_1 d_2 \dots d_{p-1}$ where d_i is an integer digit $0 \leq d_i < b$ (c is therefore an integer with $0 \leq c < b^p$).

This view of the significand as an integer c , with its corresponding exponent q , describes exactly the same set of zero and non-zero floating-point numbers as the view in scientific form. (For finite floating-point numbers, $e = q + p - 1$ and $m = c \times b^{1-p}$.)

The smallest positive *normal* floating-point number is $b^{e_{min}}$ and the largest is $b^{e_{max}} \times (b - b^{1-p})$. The non-zero floating-point numbers for a format with magnitude less than $b^{e_{min}}$ are called *subnormal* because their magnitudes lie between zero and the smallest normal magnitude. They always have fewer than p significant digits. Every finite floating-point number is an integral multiple of the smallest subnormal magnitude $b^{e_{min}} \times b^{1-p}$.

For a floating-point number that has the value zero, the sign bit s provides an extra bit of information. Although all formats have distinct representations for $+0$ and -0 , the sign of a zero is significant in some circumstances, such as division by zero, but not in others (see 6.3). Binary interchange formats have just one representation each for $+0$ and -0 , but decimal formats have many. In this standard, \emptyset and ∞ are written without a sign when the sign is not important.

Table 3.2—Parameters defining basic format floating-point numbers

parameter	Binary format ($b=2$)			Decimal format ($b=10$)	
	binary32	binary64	binary128	decimal64	decimal128
p , digits	24	53	113	16	34
e_{max}	-127	+1023	+16383	+384	+6144

3.4 Binary interchange format encodings

Each floating-point number has just one encoding in a binary interchange format. To make the encoding unique, in terms of the parameters in 3.3, the value of the significand m is maximized by decreasing e until either $e = emin$ or $m \geq 1$. After this process is done, if $e = emin$ and $0 < m < 1$, the floating-point number is subnormal. Subnormal numbers (and zero) are encoded with a reserved biased exponent value.

Representations of floating-point data in the binary interchange formats are encoded in k bits in the following three fields ordered as shown in Figure 3.1:

- 1-bit sign S
- w -bit biased exponent $E = e + bias$
- $(t - p - 1)$ -bit trailing significand field digit string $T = d_1 d_2 \dots d_{t-p-1}$; the leading bit of the significand, d_0 , is implicitly encoded in the biased exponent E .

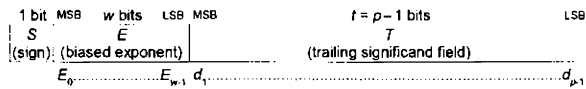


Figure 3.1—Binary interchange floating-point format

The values of k , p , t , w , and $bias$ for binary interchange formats are listed in Table 3.5 (see 3.6).

The range of the encoding's biased exponent E shall include:

- every integer between 1 and $2^n - 2$, inclusive, to encode normal numbers
- the reserved value 0 to encode ± 0 and subnormal numbers
- the reserved value $2^n - 1$ to encode $\pm\infty$ and NaNs.

The representation r of the floating-point datum, and value v of the floating-point datum represented, are inferred from the constituent fields as follows:

- If $E = 2^n - 1$ and $T \neq 0$, then r is qNaN or sNaN and v is NaN regardless of S (see 6.2.1).
- If $E = 2^n - 1$ and $T = 0$, then r and $v = (-1)^S \times (+\infty)$.
- If $1 \leq E \leq 2^n - 2$, then r is $(S, (E - bias), (1 - 2^{1-p} \times T))$; the value of the corresponding floating-point number is $v = (-1)^S \times 2^{E - bias} \times (1 + 2^{1-p} \times T)$; thus normal numbers have an implicit leading significand bit of 1.
- If $E = 0$ and $T \neq 0$, then r is $(S, emin, (0 + 2^{1-p} \times T))$; the value of the corresponding floating-point number is $v = (-1)^S \times 2^{emin} \times (0 + 2^{1-p} \times T)$; thus subnormal numbers have an implicit leading significand bit of 0.
- If $E = 0$ and $T = 0$, then r is $(S, emin, 0)$ and $v = (-1)^S \times (+0)$ (signed zero, see 6.3).

NOTE—Where k is either 64 or a multiple of 32 and ≥ 128 , for these encodings all of the following are true (where $\text{round}()$ rounds to the nearest integer):

$$\begin{aligned}
 k - 1 &= w + t - w + p = 32 \times \text{ceiling}((p - \text{round}(4 \times \log_2(p) - 13)) / 13) / 32 \\
 w &= k - t - 1 = k - p - \text{round}(4 \times \log_2(k)) - 13 \\
 t &= k - w - 1 = p - 1 = k - \text{round}(4 \times \log_2(k)) + 12 \\
 p &= k - w - t + 1 = k - \text{round}(4 \times \log_2(k)) + 13 \\
 emax &= bias - 2^{(w-1)} - 1 \\
 emin &= 1 - emax = 2^{(w-1)}
 \end{aligned}$$

3.5 Decimal interchange format encodings

3.5.1 Cohorts

Unlike in a binary floating-point format, in a decimal floating-point format a number might have multiple representations. The set of representations a floating-point number maps to is called the floating-point number's *cohort*; the members of a cohort are distinct *representations* of the same floating-point number. For example, if c is a multiple of 10 and q is less than its maximum allowed value, then (s, q, c) and $(s, q+1, c/10)$ are two representations for the same floating-point number and are members of the same cohort.

While numerically equal, different members of a cohort can be distinguished by the decimal-specific operations (see 5.3.2, 5.5.2, and 5.7.3). The cohorts of different floating-point numbers might have different numbers of members. If a finite non-zero number's representation has n decimal digits from its most significant non-zero digit to its least significant non-zero digit, the representation's cohort will have at most $p - n + 1$ members where p is the number of digits of precision in the format.

For example, a one-digit floating-point number might have up to p different representations while a p -digit floating-point number with no trailing zeros has only one representation. (An n -digit floating-point number might have fewer than $p - n + 1$ members in its cohort if it is near the extremes of the format's exponent range.) A zero has a much larger cohort: the cohort of +0 contains a representation for each exponent, as does the cohort of -0.

For decimal arithmetic, besides specifying a numerical result, the arithmetic operations also select a member of the result's cohort according to 5.2. Decimal applications can make use of the additional information cohorts convey.

3.5.2 Encodings

Representations of floating-point data in the decimal interchange formats are encoded in k bits in the following three fields, whose detailed layouts and canonical (preferred) encodings are described below.

- 1-bit sign S .
- A $w+5$ bit combination field G encoding classification and, if the encoded datum is a finite number, the exponent q and four significand bits (1 or 3 of which are implied). The biased exponent E is a $w+2$ bit quantity $q + bias$, where the value of the first two bits of the biased exponent taken together is either 0, 1, or 2.
- A t -bit trailing significand field T that contains $J \times 10$ bits and contains the bulk of the significand. When this field is combined with the leading significand bits from the combination field, the format encodes a total of $p = 3 \times J + 1$ decimal digits.

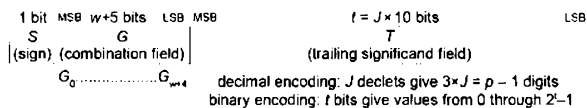


Figure 3.2—Decimal interchange floating-point formats

The values of k , p , t , w , and $bias$ for decimal interchange formats are listed in Table 3.6 (see 3.6).

The representation r of the floating-point datum, and value v of the floating-point datum represented, are inferred from the constituent fields as follows:

- If G_0 through G_4 are 11111, then v is NaN regardless of S . Furthermore, if G_5 is 1, then r is sNaN; otherwise r is qNaN. The remaining bits of G are ignored, and T constitutes the NaN's payload, which can be used to distinguish various NaNs.

The NaN payload is encoded similarly to finite numbers described below, with G treated as though all bits were zero. The payload corresponds to the significand of finite numbers, interpreted as an

integer with a maximum value of $10^{(3 \times J) - 1}$, and the exponent field is ignored (it is treated as if it were zero). A NaN is in its preferred (canonical) representation if the bits G_6 through G_{w+4} are zero and the encoding of the payload is canonical.

- b) If G_0 through G_4 are 11110 then r and $v = (-1)^s \times (+\infty)$. The values of the remaining bits in G , and T , are ignored. The two canonical representations of infinity have bits G_5 through $G_{w+4} = 0$, and $T = 0$.
- c) For finite numbers, r is $(S, E - \text{bias}, C)$ and $v = (-1)^s \times 10^{(E - \text{bias}) \times C}$, where C is the concatenation of the leading significant digit or bits from the combination field G and the trailing significant field T , and where the biased exponent E is encoded in the combination field. The encoding within these fields depends on whether the implementation uses the decimal or the binary encoding for the significand.
 - 1) If the implementation uses the *decimal* encoding for the significand, then the least significant w bits of the exponent are G_5 through G_{w+1} . The most significant two bits of the biased exponent and the decimal digit string $d_0 d_1 \dots d_{p-1}$ of the significand are formed from bits G_6 through G_4 and T as follows:
 - i) When the most significant five bits of G are 110xx or 1110x, the leading significant digit d_0 is $8 + G_6$, a value 8 or 9, and the leading biased exponent bits are $2G_2 + G_3$, a value 0, 1, or 2.
 - ii) When the most significant five bits of G are 0xxxx or 10xxx, the leading significant digit d_0 is $4G_2 + 2G_3 + G_4$, a value in the range 0 through 7, and the leading biased exponent bits are $2G_0 + G_1$, a value 0, 1, or 2. Consequently if T is 0 and the most significant five bits of G are 00000, 01000, or 10000, then $v = (-1)^s \times (+0)$.

The $p-1 = 3 \times J$ decimal digits $d_1 \dots d_{p-1}$ are encoded by T , which contains J deplets encoded in densely-packed decimal.

A canonical significand has only canonical deplets, as shown in Tables 3.3 and 3.4. Computational operations produce only the 1000 canonical deplets, but also accept the 24 non-canonical deplets in operands.

- 2) Alternatively, if the implementation uses the *binary* encoding for the significand, then:
 - i) If G_0 and G_1 together are one of 00, 01, or 10, then the biased exponent E is formed from G_6 through G_{w+1} and the significand is formed from bits G_{w+2} through the end of the encoding (including T).
 - ii) If G_0 and G_1 together are 11 and G_2 and G_3 together are one of 00, 01, or 10, then the biased exponent E is formed from G_2 through G_{w+3} and the significand is formed by prefixing the 4 bits $(8 + G_{w+4})$ to T .

The maximum value of the binary-encoded significand is the same as that of the corresponding decimal-encoded significand; that is, $10^{(p-1)} - 1$ (or $10^{(p-1)} - 1$ when T is used as the payload of a NaN). If the value exceeds the maximum, the significand c is non-canonical and the value used for c is zero.

Computational operations generally produce only canonical significands, and always accept non-canonical significands in operands.

NOTE.—Where k is a positive multiple of 32, for these encodings all of the following are true:

$$\begin{aligned}
 k &= 1 \mid 5 \mid w \mid t && 32 \times \text{ceiling}((p+2)/9) \\
 w &= k \mid t \mid 6 && k/16 + 4 \\
 t &= k \mid w \mid 6 && 15 \times k/16 - 10 \\
 p &= 3 \times t/10 + 1 && 9 \times k/32 - 2 \\
 \text{emax} &= 3 \times 2^{(w-1)} \\
 \text{emin} &= 1 - \text{emax} \\
 \text{bias} &= \text{emax} + p - 2.
 \end{aligned}$$

Decoding densely-packed decimal: Table 3.3 decodes a dectet, with 10 bits $b_{(6)}$ to $b_{(9)}$, into 3 decimal digits $d_{(1)}$, $d_{(2)}$, $d_{(3)}$. The first column is in binary and an “x” denotes a “don’t care” bit. Thus all 1024 possible 10-bit patterns shall be accepted and mapped into 1000 possible 3-digit combinations with some redundancy.

Table 3.3—Decoding 10-bit densely-packed decimal to 3 decimal digits

$b_{(6)}, b_{(7)}, b_{(8)}, b_{(3)}, b_{(4)}$	$d_{(1)}$	$d_{(2)}$	$d_{(3)}$
0 x x x x	$4b_{(6)} + 2b_{(1)} + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$4b_{(7)} + 2b_{(8)} + b_{(9)}$
1 0 0 x x	$4b_{(6)} + 2b_{(1)} + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$8 + b_{(9)}$
1 0 1 x x	$4b_{(6)} + 2b_{(1)} + b_{(2)}$	$8 + b_{(5)}$	$4b_{(3)} + 2b_{(4)} + b_{(9)}$
1 1 0 x x	$8 + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$4b_{(6)} + 2b_{(1)} + b_{(9)}$
1 1 1 0 0	$8 + b_{(2)}$	$8 + b_{(5)}$	$4b_{(6)} + 2b_{(1)} + b_{(9)}$
1 1 1 0 1	$8 + b_{(2)}$	$4b_{(6)} + 2b_{(1)} + b_{(5)}$	$8 + b_{(9)}$
1 1 1 1 0	$4b_{(6)} + 2b_{(1)} + b_{(2)}$	$8 + b_{(5)}$	$8 + b_{(9)}$
1 1 1 1 1	$8 + b_{(2)}$	$8 + b_{(5)}$	$8 + b_{(9)}$

Encoding densely-packed decimal: Table 3.4 encodes 3 decimal digits $d_{(1)}$, $d_{(2)}$, and $d_{(3)}$, each having 4 bits which can be expressed by a second subscript $d_{(1,a,3)}$, $d_{(2,a,3)}$, and $d_{(3,a,3)}$, where bit 0 is the most significant and bit 3 the least significant, into a dectet, with 10 bits $b_{(6)}$ to $b_{(9)}$. Computational operations generate only the 1000 canonical 10-bit patterns defined by Table 3.4.

Table 3.4—Encoding 3 decimal digits to 10-bit densely-packed decimal

$d_{(1,0)}, d_{(2,0)}, d_{(3,0)}$	$b_{(6)}, b_{(1)}, b_{(2)}$	$b_{(3)}, b_{(4)}, b_{(5)}$	$b_{(6)}$	$b_{(7)}, b_{(8)}, b_{(9)}$
0 0 0	$d_{(1,1,3)}$	$d_{(2,1,3)}$	0	$d_{(3,1,3)}$
0 0 1	$d_{(1,1,3)}$	$d_{(2,1,3)}$	1	0, 0, $d_{(3,3)}$
0 1 0	$d_{(1,1,3)}$	$d_{(3,1,2)}, d_{(2,3)}$	1	0, 1, $d_{(3,3)}$
0 1 1	$d_{(1,1,3)}$	1, 0, $d_{(2,3)}$	1	1, 1, $d_{(3,3)}$
1 0 0	$d_{(3,1,2)}, d_{(1,3)}$	$d_{(2,1,3)}$	1	1, 0, $d_{(3,3)}$
1 0 1	$d_{(2,1,2)}, d_{(1,3)}$	0, 1, $d_{(2,3)}$	1	1, 1, $d_{(3,3)}$
1 1 0	$d_{(3,1,2)}, d_{(1,3)}$	0, 0, $d_{(2,3)}$	1	1, 1, $d_{(3,3)}$
1 1 1	0, 0, $d_{(1,3)}$	1, 1, $d_{(2,3)}$	1	1, 1, $d_{(1,3)}$

The 24 non-canonical patterns of the form 01x11x111x, 10x11x111x, or 11x11x111x (where an “x” denotes a “don’t care” bit) are not generated in the result of a computational operation. However, as listed in Table 3.3, these 24 bit patterns do map to values in the range 0 through 999. The bit pattern in a NaN trailing significand field can affect how the NaN is propagated (see 6.2).

3.6 Interchange format parameters

Interchange formats support the exchange of floating-point data between implementations. In each radix, the precision and range of an interchange format is defined by its size; interchange of a floating-point datum of a given size is therefore always exact with no possibility of overflow or underflow.

This standard defines binary interchange formats of widths 16, 32, 64, and 128 bits, and in general for any multiple of 32 bits of at least 128 bits. Decimal interchange formats are defined for any multiple of 32 bits of at least 32 bits.

The parameters p and $emax$ for every interchange format width are shown in Table 3.5 for binary interchange formats and in Table 3.6 for decimal interchange formats. The encodings for the interchange formats are as described in 3.4 and 3.5.2; the encoding parameters for each interchange format width are also shown in Tables 3.5 and 3.6.

Table 3.5—Binary interchange format parameters

Parameter	binary16	binary32	binary64	binary128	binary(k) ($k \geq 128$)
k , storage width in bits	16	32	64	128	multiple of 32
p , precision in bits	11	24	53	113	$k - \text{round}(4 \times \log_2(k)) + 13$
$emax$, maximum exponent e	15	127	1023	16383	$2^{(k/p)-1} - 1$
<i>Encoding parameters</i>					
$bias, E - e$	15	127	1023	16383	$emax$
sign bit	1	1	1	1	1
w , exponent field width in bits	5	8	11	15	$\text{round}(4 \times \log_2(k)) - 13$
t , trailing significand field width in bits	10	23	52	112	$k - w - 1$
k , storage width in bits	16	32	64	128	$1 + w + t$

The function $\text{round}()$ in Table 3.5 rounds to the nearest integer.

For example, binary256 would have $p = 237$ and $emax = 262143$.

Table 3.6—Decimal interchange format parameters

Parameter	decimal32	decimal64	decimal128	decimal(k) ($k \geq 32$)
k , storage width in bits	32	64	128	multiple of 32
p , precision in digits	7	16	34	$9 \times k / 32 - 2$
$emax$	96	384	6144	$3 \times 2^{(k/16)-3}$
<i>Encoding parameters</i>				
$bias, E - q$	101	398	6176	$emax + p - 2$
sign bit	1	1	1	1
$w+5$, combination field width in bits	11	13	17	$k/16 + 9$
t , trailing significand field width in bits	20	50	110	$15 \times k / 16 - 10$
k , storage width in bits	32	64	128	$1 + 5 + w + t$

For example, decimal256 would have $p = 70$ and $emax = 1572864$.

3.7 Extended and extendable precisions

Extended and extendable precision formats are recommended for extending the precisions used for arithmetic beyond the basic formats. Specifically:

- An **extended precision format** is a format that extends a supported basic format with both wider precision and wider range.
- An **extendable precision format** is a format with a precision and range that are defined under user control.

These formats are characterized by the parameters b , p , and $emax$, which may match those of an interchange format and shall:

- provide all the representations of floating-point data defined in terms of those parameters in 3.2 and 3.3
- provide all the operations of this standard, as defined in Clause 5, for that format.

This standard does not require an implementation to provide any extended or extendable precision format. Any encodings for these formats are implementation-defined, but should be fixed width and may match those of an interchange format.

Language standards should define mechanisms supporting extendable precision for each supported radix. Language standards supporting extendable precision shall permit users to specify p and $emax$. Language standards shall also allow the specification of an extendable precision by specifying p alone; in this case $emax$ shall be defined by the language standard to be at least $1000 \times p$ when p is ≥ 237 bits in a binary format or p is ≥ 51 digits in a decimal format.

Language standards or implementations should support an extended precision format that extends the widest basic format that is supported in that radix. Table 3.7 specifies the minimum precision and exponent range of the extended precision format for each basic format.

Table 3.7—Extended format parameters for floating-point numbers

Parameter	Extended formats associated with:				
	binary32	binary64	binary128	decimal64	decimal128
p digits \geq	32	64	128	22	40
$emax \geq$	1023	16383	65535	6144	24576

NOTE 1—For extended formats, the minimum exponent range is that of the next wider basic format, if there is one, while the minimum precision is intermediate between a given basic format and the next wider basic format.

NOTE 2—For interchange of binary floating-point data, the width k in bits of the smallest standard format that will allow the encoding of a significand of at least p bits is given by:

$$k = 32 \times \text{ceiling}((p + \text{round}(4 \times \log_2(p + \text{round}(4 \times \log_2(p) - 13)) - 13))/32), \text{ where } \text{round}() \text{ rounds to the nearest integer and } p \geq 113; \text{ for smaller values of } p, \text{ see Table 3.5.}$$

For interchange of decimal floating-point data, the width k in bits of the smallest standard format that will allow the encoding of a significand of at least p digits is given by:

$$k = 32 \times \text{ceiling}((p + 2)/9), \text{ where } p \geq 1.$$

In both cases the chosen format might have a larger precision (see 3.4 and 3.5.2).

4. Attributes and rounding

4.1 Attribute specification

An attribute is logically associated with a program block to modify its numerical and exception semantics. A user can specify a constant value for an attribute parameter.

Some attributes have the effect of an implicit parameter to most individual operations of this standard; language standards shall specify

- rounding-direction attributes (see 4.3)

and should specify

- alternate exception handling attributes (see 8).

Other attributes change the mapping of language expressions into operations of this standard; language standards that permit more than one such mapping should provide support for:

- preferredWidth attributes (see 10.3)
- value-changing optimization attributes (see 10.4)
- reproducibility attributes (see 11).

For attribute specification, the implementation shall provide language-defined means, such as compiler directives, to specify a constant value for the attribute parameter for all standard operations in a block; the scope of the attribute value is the block with which it is associated. Language standards shall provide for constant specification of the default and each specific value of the attribute.

4.2 Dynamic modes for attributes

Attributes in this standard shall be supported with the constant specification of 4.1. Particularly to support debugging, language standards should also support dynamic-mode specification of attributes.

With dynamic-mode specification, a user can specify that the attribute parameter assumes the value of a dynamic-mode variable whose value might not be known until program execution. This standard does not specify the underlying implementation mechanisms for constant attributes or dynamic modes.

For dynamic-mode specification, the implementation shall provide language-defined means to specify that the attribute parameter assumes the value of a dynamic-mode variable for all standard operations within the scope of the dynamic-mode specification in a block. The implementation initializes a dynamic-mode variable to the default value for the dynamic mode. Within its language-defined (dynamic) scope, changes to the value of a dynamic-mode variable are under the control of the user via the operations in 9.3.1 and .

The following aspects of dynamic-mode variables are language-defined; language standards may explicitly defer the definitions to implementations:

- The precedence of static attribute specifications and dynamic-mode assignments.
- The effect of changing the value of the dynamic-mode variable in an asynchronous event, such as in another thread or signal handler.
- Whether the value of the dynamic-mode variable can be determined by non-programmatic means, such as a debugger.

NOTE —A constant value for an attribute can be specified and meet the requirements of 4.1 by a dynamic mode specification with appropriate scope of that constant value.

4.3 Rounding-direction attributes

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while signaling the inexact exception, underflow, or overflow when appropriate (see 7). Except where stated otherwise, every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the attributes in this clause.

The rounding-direction attribute affects all computational operations that might be inexact. Inexact numeric floating-point results always have the same sign as the unrounded result.

The rounding-direction attribute affects the signs of exact zero sums (see 6.3), and also affects the thresholds beyond which overflow (see 7.4) and underflow (see 7.5) are signaled.

Implementations supporting both decimal and binary formats shall provide separate rounding-direction attributes for binary and decimal, the binary rounding direction and the decimal rounding direction. Operations returning results in a floating-point format shall use the rounding-direction attribute associated with the radix of the results. Operations converting from an operand in a floating-point format to a result in integer format or to an external character sequence (see 5.8 and 5.12) shall use the rounding-direction attribute associated with the radix of the operand.

NaNs are not rounded (but see 6.2.3).

4.3.1 Rounding-direction attributes to nearest

In the following two rounding-direction attributes, an infinitely precise result with magnitude at least $b^{emax}(b \cdot \frac{1}{2}b^1)^p$ shall round to $\cdot e$ with no change in sign; here *emax* and *p* are determined by the destination format (see 3.3). With:

- *roundTiesToEven*, the floating-point number nearest to the infinitely precise result shall be delivered; if the two nearest floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with an even least significant digit shall be delivered
- *roundTiesToAway*, the floating-point number nearest to the infinitely precise result shall be delivered; if the two nearest floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with larger magnitude shall be delivered.

4.3.2 Directed rounding attributes

Three other user-selectable rounding-direction attributes are defined, the directed rounding attributes *roundTowardPositive*, *roundTowardNegative*, and *roundTowardZero*. With:

- *roundTowardPositive*, the result shall be the format's floating-point number (possibly $+\infty$) closest to and no less than the infinitely precise result
- *roundTowardNegative*, the result shall be the format's floating-point number (possibly $-\infty$) closest to and no greater than the infinitely precise result
- *roundTowardZero*, the result shall be the format's floating-point number closest to and no greater in magnitude than the infinitely precise result.

4.3.3 Rounding attribute requirements

An implementation of this standard shall provide *roundTiesToEven* and the three directed rounding attributes. A decimal format implementation of this standard shall provide *roundTiesToAway* as a user-selectable rounding-direction attribute. The rounding attribute *roundTiesToAway* is not required for a binary format implementation.

The *roundTiesToEven* rounding-direction attribute shall be the default rounding-direction attribute for results in binary formats. The default rounding-direction attribute for results in decimal formats is language-defined, but should be *roundTiesToEven*.

5. Operations

5.1 Overview

All conforming implementations of this standard shall provide the operations listed in this clause for all supported arithmetic formats, except as stated below. Each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format (see 4 and 7). Clause 6 augments the following specifications to cover ± 0 , $\pm\infty$, and NaN. Clause 7 describes default exception handling.

In this standard, operations are written as named functions; in a specific programming environment they might be represented by operators, or by families of format-specific functions, or by operations or functions whose names might differ from those in this standard.

Operations are broadly classified into four groups according to the kinds of results and exceptions they produce:

- General-computational operations produce floating-point or integer results, round all results according to Clause 4, and might signal the floating-point exceptions of Clause 7.
- Quiet-computational operations produce floating-point results and do not signal floating-point exceptions.
- Signaling-computational operations produce no floating-point results and might signal floating-point exceptions; comparisons are signaling-computational operations.
- Non-computational operations do not produce floating-point results and do not signal floating-point exceptions.

Operations in the first three groups are referred to collectively as “computational operations”.

Operations are also classified in two ways according to the relationship between the result format and the operand formats:

- homogeneous operations, in which the floating-point operands and floating-point result are all of the same format
- *formatOf* operations, which indicate the format of the result, independent of the formats of the operands.

Language standards might permit other kinds of operations and combinations of operations in expressions. By their expression evaluation rules, language standards specify when and how such operations and expressions are mapped into the operations of this standard. Operations (except re-encoding operations) do not have to accept operands or produce results of differing encodings.

In the operation descriptions that follow, operand and result formats are indicated by:

- *source* to represent homogeneous floating-point operand formats
- *source1*, *source2*, *source3* to represent non-homogeneous floating-point operand formats
- *int* to represent integer operand formats
- *boolean* to represent a value of *false* or *true* (for example, 0 or 1)
- *enum* to represent one of a small set of enumerated values
- *logBFormat* to represent a type for the destination of the *logB* operation and the scale exponent operand of the *scaleB* operation
- *integralFormat* to represent the scale factor in scaled products (see 9.4)
- *decimalCharacterSequence* to represent a decimal character sequence
- *hexCharacterSequence* to represent a hexadecimal-significand character sequence
- *conversionSpecification* to represent a language dependent conversion specification
- *decimal* to represent a supported decimal floating-point type
- *decimalEncoding* to represent a decimal floating-point type encoded in decimal

- *binaryEncoding* to represent a decimal floating-point type encoded in binary
- *exceptionGroup* to represent a set of exceptions as a set of *booleans*
- *flags* to represent a set of status flags
- *binaryRoundingDirection* to represent the rounding direction for binary
- *decimalRoundingDirection* to represent the rounding direction for decimal
- *modeGroup* to represent dynamically-specifiable modes
- *void* to indicate that an operation has no explicit operand or has no explicit result; the operand or result might be implicit.

formatOf indicates that the name of the operation specifies the floating-point destination *format*, which might be different from the floating-point operands' formats. There are *formatOf* versions of these operations for every supported arithmetic format.

intFormatOf indicates that the name of the operation specifies the integer destination format.

In the operation descriptions that follow, languages define which of their types correspond to operands and results called *int*, *intFormatOf*, *characterSequence*, or *conversionSpecification*. Languages with both signed and unsigned integer types should support both signed and unsigned *int* and *intFormatOf* operands and results.

5.2 Decimal exponent calculation

As discussed in 3.5, a floating-point number might have multiple representations in a decimal format. Therefore, decimal arithmetic involves not only computing the proper numerical result but also selecting the proper member of that floating-point number's cohort.

Except for the quantize operation, the value of a floating-point result (and hence its cohort) is determined by the operation and the operands' values; it is never dependent on the representation or encoding of an operand.

The selection of a particular representation for a floating-point result is dependent on the operands' representations, as described below, but is not affected by their encoding.

For all computational operations except *quantize* and *roundToIntegralExact*, if the result is inexact the cohort member of least possible exponent is used to get the maximum number of significant digits. If the result is exact, the cohort member is selected based on the preferred exponent for a result of that operation, a function of the exponents of the inputs. Thus for finite x , depending on the representation of zero, $0+x$ might result in a different member of x 's cohort. If the result's cohort does not include a member with the preferred exponent, the member with the exponent closest to the preferred exponent is used.

For *quantize* and *roundToIntegralExact*, a finite result has the preferred exponent, whether or not the result is exact.

In the descriptions that follow, $Q(x)$ is the exponent q of the representation of a finite floating-point number x . If x is infinite, $Q(x)$ is $+\infty$.

Index

A

Arithmetic processor 8

ASIC 9

B

BCD adder 3,40,41,48,50,53,78,81,85,90,93,102,104,107,110-116,140,183,192,210

BFP 4

BID 12

Binary arithmetic 3,4,5,6,10,

Binary floating point 51

BCD Digit Multiplier 36,169

Binary Multiplier 37

Binary to BCD Converter 39,126,128

C

Carry counter (CC) 59,60,63,172,174

Carry propagate adder (CPA) 74,157,159,161,162,163

Carry select BCD 17,110,111,112,124,209

Carry save adder (CSA)

Combination field 11,12,15

Conventional BCD 17,99,100,103,109,110,115,117, 119,120,121, 125,186, 187, 189,191,202,209

Critical path delay 16,18,37,43,71,128,169,170,183,208

Carry Save Adder Block 31

Carry Select BCD Adder 110

Column Accumulation 60

Combination Field 15

Computer Arithmetic Systems 7

D

DDDM 16,25,26,35,36,62,67,84,157,159,161,207

Decimal carry save adder (DCA) 56,57,58,172,174

DPD 12,13,69,71,75,77

DSP 42

Decimal arithmetic 3
Decimal Carry Propagate Adder 33
Decimal Encodings 10
Decimal Fixed point multiplication 23
Decimal Floating Point Multipliers 67,174
Densely Packed Decimal Encoding 13
DFP Adders 77
DFP MAC Unit 75,181
DFP Multiplication 68,74,75
Double Digit Decimal Multiplication 26,157

E

ENIAC 3, 7
Error detection 17, 76 , 104, 136, 203, 210
Exception 8, 71, 72, 175
Exhaustive branching 17,18,146,147,148,149,150,152

F

Fault tolerant 17,98,135,137,140,193,209
FPGA 62,166,167,168,169,201
Fredkin gates 17,187,189,190,203,209
Fully Reversible RPS Gate 125

G

Garbage output 118, 183,188,193,194,203, 209,210

H

Hamming code 17,135,136,137,140,141,193,194,203,210
HNG 17,103,128,129,132,133,134,191,192,193,210
Hex/Decimal Multiplier 42
Hybrid BCD Adder 113
Hybrid Reversible BCD Adder 115

I

IEEE 754-2008 - 6,9,10,11,12,15,16,75,78,84
Iterative DFxP Multipliers 23

K

Kogge-Stone Adder 80

L

Latency 16,25,35,37,54,56,62,85,168,170,178,179,201,202,208

Low power 16,17,18,89,210

LUT 24,166

Logic synthesis using multiplexers 145,194

M

Minterms 17,146,150,151,194

Multiplexer 209

Multi-operand Decimal Adders 44

Multiplexer Block 29

Multiplier Shift Register 29

N

NaN 10,11,15,72,76

Nanotechnology 16

N-ary Exhaustive Branching Technique 146,149

New Reversible RPS Gate 124

P

Parallel DFP multiplier 16,18,56,67,75,84,180,202, 208

Partially reversible 125,129,130,131,132,141,191,192,203,209,212

Patriot missile 5

PPRM 91

Parallel DFxP Multipliers 56,172

Parity Preserving Reversible Carry Select BCD Adder 112

Parity Preserving Reversible Quick Decimal Adder 107

Partial Product Register 33

Partially Reversible RPS Gate 129

Q

QAD 17,104,105,106,112,120,121,124,140,189,190

Quantum computing 16,90

Quantum cost 116,117,118,120,124,188,189,190,203,209
Quick Decimal Adder 104

R

RBCD 452
Reed muller 91, 141
Rounding 69, 171, 175, 177, 181, 208
RPS algorithm 16, 18, 36, 46, 52, 53, 55, 62, 67, 68, 170
Reduced Delay BCD Adder 81
Reversible Decimal Adders 99
Reversible Error Correcting Code 134
Reversible Fast Decimal Adders 104
Reversible Full Adders 93
Reversible Gates 90
Reversible Logic 89
Ripple Carry BCD Adder 79
Row Accumulation 56
RPS Algorithm 46

S

SDDM 35, 164, 165
Secondary multiple 25, 26
Secondary multiple generation block 26, 157, 159, 161, 163

T

Throughput 16, 25, 56, 84, 208
Toffoli gate 17, 91, 95, 100, 116, 140, 188, 189, 203
Trailing significand field 11, 12
TSG 92, 96, 100, 101, 132, 184, 192

U

Universal Logic Module (ULM) 145

V

VHDL 9, 147, 170, 193