FPGA IMPLEMENTATION OF RADIX-10 PARALLEL DECIMAL MULTIPLIER

A Dissertation Submitted In Partial Fulfilment of the Required for the Degree of

MASTER OF TECHNOLOGY

In

VLSI Design

Submitted By

GEETA

Roll no. 601361009

Under the guidance of

(Ms. Sakshi Bajaj)

Assistant Professor, ECED

T.U. Patiala

Department of Electronics and Communication Engineering

THAPAR UNIVERSITY

(Established under the section 3 of UGC Act, 1956)

PATIALA 147004 (PUNJAB)

July, 2015

DECLARATION

I hereby declare that the work which is being presented in dissertation titled **"FPGA IMPLEMENTATION OF RADIX-10 PARALLEL DECIMAL MULTIPLIER"** in partial fulfillment of the requirement for the award of degree of Master of Technology in VLSI Design submitted in Electronics and Communication Engineering Department of Thapar university, Patiala is an authentic record of my study carried out as under the guidance of **Ms. Sakshi Bajaj** (Assistant Professor), ECED during 2013-2015.

Date: 13/07/14

Geeta

Geeta
Roll no.601361009

It is certified that the above statement made by the student is correct to the best of my Knowledge and belief.

(Sakshi Bajaj)

Assistant Professor, ECED.
Thapar University,
Patiala 147004, (Punjab)

Countersigned by:

(Dr.Sanjay Sharma)

Head
ECED, Thapar University,
Patiala, 147004

Dean of Academics Affairs
ECED, Thapar University,
Patiala, 147004

ACKNOWLEDGEMENT

First of all, I would like to express my gratitude to **Ms. Sakshi Bajaj** mam, Assistant Professor, Electronics and Communication Engineering Department, Thapar University, Patiala for his patient guidance and support throughout this report. I am truly very fortunate to have the opportunity to work with him. I found this guidance to be extremely valuable.

I am also thankful to our **Head of the Department, Dr. Sanjay Sharma** as well as **PG Coordinator, Dr. Amit Kumar Kohli** Associate professor, **Dr. Alpana Agarwal** Associate professor, **Dr. Anil Arora** Assistant professor, Electronics and Communication Engineering Department, entire faculty and staff of Electronics and Communication Engineering Department and then friends who devoted their valuable time and helped me in all possible ways towards successful completion of this work.

I thank all those who have contributed directly or indirectly to this work.

Lastly, I would also like to thank my parents for their years of unyielding love and encourage. They have always wanted the best for me and I admire their determination and sacrifice.

GEETA

ABSTRACT

Multipliers are being increasingly used in DSP processors, filters, communications systems etc. With the rising complications of technology, high-speed systems are in great demand. On comparison with other operation in an arithmetic logic unit the multiplier consumes more time and power. Hence the demand to design or implement multipliers with optimal speed, power and area has increased.

This dissertation includes the implementation of decimal multipliers which are arranged in parallel, with the idea of reducing delay. The partial products are generated in parallel by using signed digit radix-10 recoding of the multiplier and a simplified box of multiplicand multiples. Number of partial products are reduced by creating a tree structure of partial products. The same is developed by using a new algorithm known as decimal multi-operand carry save addition. This uses unconventional decimal coded number systems, which largely improves the area and latency of the prior or existing design. It includes optimized digit recorders, decimal carry-save adders (CSA's) combining different decimal-coded operands, and carry free adders implemented by special designed bit counters (a design methodology that combines all these techniques to obtain efficient reduction trees with different area and delay trade-offs for any number of partial products generated). The generation of partial products are developed parallel by using signed-digit (SD) radix-10 recordings of the multiplier and a simplified set of multiplicand multiples. Evaluation results for 16-digit operands show that the proposed architectures have interesting area-delay figures compared to conventional Booth radix-4 and radix-8 parallel binary multipliers and outperform the figures of previous alternatives for decimal multiplication. The modules have been designed in Verilog HDL, simulated and synthesized using Xilinx 14.5.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

Z_i	Decimal integer operand
i	Decimal digit
X	Multiplicand
Y	Multiplier
r_i	Weight of the j bits
P	Final Product
ys_{i-1}	Sign signal
yb_i	Hot one code
V	OR
.	AND
$\oplus$	XOR
Lm_{shift}	Left arithmetic binary shift of m bit
$w_{i,j}$	Bits of the BCD
$x_{i,j}$	Bits of the (4221)
S	Sign encoding
H	Hot one 10’s complement encoding
X	Regular (4221) digit
F	extra digit position to support the width of multiplicand
V	Regular (4221) digit
B	Regular (5211) digit
W	Carry vector coded in (5211)
$s_{i,j}$	Sum bit of a full adder
$h_{i,j}$	Carry bit of full adder

ABBRIVEATIONS

DFP	Decimal floating point
DFUs	Decimal floating units
DSP	Digital signal processor
FPGA	Field programmable gate array
HDL	Hardware description language
IEEE	Institute of Electrical and Electronics Engineers
ISE	Integrated Software Environment
LSB	Least significant digit
LUT	Look up table
MAC	Multiply and accumulate
MSB	Most significant digit
RTL	Register transfer level
RAM	Random access memory
VLSI	Very large scale integration
XST	Xilinx synthesis technology
CSA	Carry save adder
HA	Half adder
PP	Partial product
BCD	Binary coded decimal
SD	Signed digit
CALs	Carry look ahead adder
ulp	Unit in the last place

EDIF	Electronics design interchange format
NGC	Netlist Constraints File
ILA	Integrated Logic Analyzer
VIO	Virtual Input /Output
ATC2	Agilent Trace Core 2
IBA	Integrated Bus Analyzer

CHAPTER 1

INTRODUCTION

1.1 Introduction To Multiplier

Multipliers are widely used in many high performance systems. They are used as small blocks in large systems like Digital signal processors, filters, communication systems, microprocessors etc. DSP‟s have operations like convolution operations, transform operations, filtering operations etc. All these operations need multiplication unit. So multiplier plays an important role in DSP‟s. In older microprocessors, multiplication is performed by repeated addition but in recent microprocessors multiplication instructions are available. In communication systems for baseband signal transmission multiplier is used. Filter contains large number of multipliers, so use of fast multipliers is necessary for high speed applications. Multipliers also find application in 3D graphics and in video processing for displaying streamline images. The time taken by multiplier in calculating final result are very important and plays major role in determining its performance. In DSP chips multiplication time plays important role in calculating instruction cycle time. So there is a need of high speed multiplier. A multiplier is a digital circuit, which is used to multiply two numbers.

Digital multiplier multiplies two numbers in a similar manner as it is done in mathematics i.e. addition of shifted partial products. Multiplication of unsigned numbers can be done by using AND gates and full adders. AND gates are used for generating the partial products and full adders are used for adding the generated partial products. For signed number multiplication the negative numbers are first converted into its 2’s complement representation. This makes sure that the all the partial products are positive. There are three types of digital multipliers i.e. serial, parallel and serial-parallel multiplier. In serial multiplier, both the operands are entered serially, therefore it requires less area. Due to less area requirement of serial multiplier it has less hardware cost and

low power consumption. But the speed of the serial multiplier is poor because the operands are entered serially. In parallel multiplier operation is carried out in parallel which increases the speed of the multiplier. But parallel multipliers occupy larger area and more complex as compared to serial multipliers. High power consumption is also one problem in parallel multipliers. Parallel multipliers can further classified into array multipliers and tree multipliers. Most widely used array multipliers are Braun multiplier, booth multiplier, Robertson multiplier, Baugh-Wooley multiplier etc. Serial-parallel multipliers are used to take advantage of high speed operation of parallel multiplier and small area of serial multiplier. In serial-parallel multipliers, one operand is entered serially while other is stored in parallel. This requires less area and also enhances the speed of the multiplier.

In today scenario hardware is becoming a topic of interest for decimal floating point (DFP).software DFP implementation are slower than hardware implementation although this satisfy the precision requirement and this could not stratify the demands of future financial high performance, user-oriented applications and commercial [1].The revision of IEEEE 754 [2] standard for floating point arithmetic specifications for DFP arithmetic that can be executed in software, hardware or combination of both. The IBM Z_9 architecture perform the basic operation dedicated hardware. The IBM power6 microprocessor [3] oriented to servers and work station for high performance DFUs the key factor is multiplication. Most of the binary floating point unit FPUs uses parallel binary multipliers for high performance [4].however complexity in the generation of multiplicand multiplies and in adequacy in representing decimal values for the system release on binary signals, results in hard to implement decimal multiplication. This leads to complication in generation and reduction of partial product, thus decimal adders are representing in a parallel fashion while commercial implementation of sequential decimal multiplier [5,6,7].these implementation are released on iterative algorithm for decimal integer multiplication [7,8] and therefore leads to low performance several techniques has been proposed for the improvement of sequential decimal multiplier [9,10,11]. The first implementation of the parallel decimal multiplier is show in [12].some other parallel decimal multiplier architecture are shown in [8]. These architecture are further advanced to support binary multiplication.in this paper represented the method for the efficient implementation of decimal parallel multiplication using a parallel generation and reduction of the partial product by a carry-save addition tree.

1.2 Organization of Thesis

The report is organized as follows:

- Chapter 2- Describe the applications of the multiplier and gives a brief about the Parallel decimal multiplier.
- Chapter 3- Discusses the IEEE 754 standard, Floating point multiplication algorithm And the adders used in floating point multiplication.
- Chapter 4- FPGA flow using Xilinx is discussed in this chapter.
- Chapter 5- Implementation results are described and report is concluded.
- Chapter 6- Conclusion and future scope of the implemented design is discussed here.

CHAPTER 2

LITERATURE REVIEW

Liu han [13]: Describe decimal multiplication is considered as one of the most complicated dyadic operations, which requires high-cost hardware implementation. Therefore, the processor industry has opted to use the sequential decimal multipliers to reduce the high cost of parallel architectures. However, the main drawback of iterative multipliers is their high latency. The focus has been on reducing the latency of decimal sequential multipliers while maintaining a low cost of area. Consequently, a high-frequency sequential decimal multiplier is proposed whose cycle time is reduced to the latency of a binary half-adder plus that of a decimal multiply-by-two operation, which overall is less than that of a decimal carry-save adder. The sequential multiplier works with a higher clock frequency than the fastest previous decimal multiplier which in turn leads to overall latency advantage.

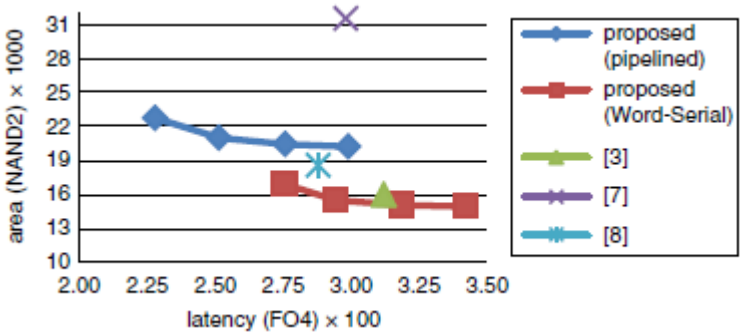


Figure 2.1: Delay-constrained comparison graph [13]

Kenney, R.D [11]: Presented the decimal arithmetic is regaining popularity in the computing community due to the growing importance of commercial, financial, and Internet-based applications, which process decimal data .iterative decimal multiplier, which is operates at high clock frequencies and scales well to large operand sizes. The multiplier uses a new decimal representation for intermediate products, which allows for a very fast two stage iterative multiplier design. Decimal multipliers, which are

synthesized using a 0.11 micron CMOS standard cell library, operate at clock frequencies close to 2 GHz. The latency of the proposed design to multiply two n-digit BCD operands is (n + 8) cycles with a new multiplication able to begin every (n + 1) cycles.

Vazquez.A and antelo [12]: Introduce the two novel architecture for parallel decimal multiplier.Our multiplier are based on a new algorithm for decimal carry-save multioperand addition that uses a novel BCD–4221 recoding for decimal digits. It significantly improves the area and latency of the partial product reduction tree with respect to previous proposals. This paper present three schemes for fast and efficient generation of partial products in parallel. The recoding of the BCD–8421 multiplier operand into minimally redundant signed–digit radix–10, radix–4 and radix–5 representations using new recorder's reduces the complexity of partial product generation. In addition, SD radix–4 and radix–5 recoding's allow the reuse of a conventional parallel binary radix–4 multiplier to perform combined binary decimal multiplications. Evaluation results show that the proposed architectures have interesting area–delay figures compared to conventional Booth radix–4 and radix–8parallel binary multipliers and other representative alternatives for decimal multiplication.

Vazquea, A [14]: Presented an high-performance decimal floating-point units (DFUs) is demanding efficient implementations of parallel decimal multipliers. The parallel generation of partial products is performed using signed-digit radix-10 or radix-5 recordings of the multiplier and a simplified set of multiplicand multiples. The reduction of partial products is implemented in a tree structure based on a decimal multi operand carry-save addition algorithm that uses unconventional (non BCD) decimal-coded number systems. The present the new improvements to reduce the latency of the previous designs, which include: optimized digit recorders for the generation of 2n-tuples (and 5-tuples), decimal carry-save adders (CSAs) combining different decimal-coded operands, and carry-free adders implemented by special designed bit counters. A design methodology that combines all these techniques to obtain efficient reduction trees with different area and delay trade-offs for any number of partial products generated.

M. F. Cowlishaw [15]: Decimal arithmetic is the norm in human calculations, and human-centric applications must use a decimal floating-point arithmetic to achieve the same results. Initial benchmarks indicate that some applications spend 50% to 90% of their time in decimal processing, because software decimal arithmetic suffers a 100× to 1000× performance penalty over hardware. Existing designs, however, either fail to

conform to modern standards or are incompatible with the established rules of decimal arithmetic. Decimal floating- point which not only provides the strict results which are necessary for commercial applications but also meets the constraints and requirements of the IEEE 854 standard.

M.A. Erle [16]: Proposed a fixed-point decimal multiplication that utilizes a simple recoding scheme to produce signed-magnitude representations of the operands thereby greatly simplifying the process of generating partial products for each multiplier digit. The partial products are generated using a digit-by-digit multiplier on a word by- digit basis, first in a signed-digit form with two digits per position, and then combined via a combinational circuit. As the signed-digit partial products are developed one at a time while traversing the recoded multiplier operand from the least significant digit to the most significant digit, each partial product is added along with the accumulated sum of previous partial products via a signed-digit adder. This work is significantly different from other work employing digit-by-digit multipliers due to the efficiency gained by restricting the range of digits throughout the multiplication process.

G. Jaberipur [17]: Describe contributions to digit recurrence decimal division hardware and focus on techniques for improving the performance of quotient digit selection (QDS) as the most complex part. Design D1 uses the digit set for quotient digits. Another design (D2) uses mixed binary/decimal carry-save manipulation of the few most significant digits of partial remainders. Motivated by successful combined arithmetic algorithms such as hybrid adders, the authors offer a decimal division scheme that takes advantage of the best design options of D1 and D2 with due modifications that significantly enhance the division speed. In particular, they configure the architectures of QDS and partial remainder computation paths in favor of reduced balanced latencies of both. Furthermore, they remove the rounding cycle by cost-free auto-rounding, which is an exclusive advantage of the digit set. The authors of D1 and D2 have used logical effort (LE) and circuit synthesis to evaluate their dividers, respectively. Therefore for a fair comparison, the authors evaluate the proposed design (D3) with both methods. The LE-based D3/D1 comparison shows 21% more speed at the cost of 6% more area, whereas the synthesis-based D3/D2 comparison results in 46% less latency and 23% less area.

Kaivani, A [18]: Multiplication, as one of the four basic operations embedded in arithmetic processors, is nowadays experiencing being spotlighted by the hardware designers involved in the revived decimal arithmetic. The decimal hardware units usually

employ the sequential implementation for this operation, due to the high area cost of the parallel decimal multipliers. However, the main drawback of this iterative method is in regard to its high latency. The intention of ameliorating this problem, proposes a high-frequency sequential decimal Multiplier. The cycle time of the proposed multiplier is determined by a decimal carry-save adder which is about 22% less than that of the fastest previous design.

Schmookler .M [19]: Describe Parallel decimal arithmetic capability is becoming increasingly attractive with new applications of computers in a multiprogramming environment. The direct production of decimal sums offers a significant improvement in addition over methods requiring decimal correction. The several techniques useful in designing decimal adders which are both high speed and economical. One such technique is the direct production of decimal sums without the need of first producing the binary sums. Another technique is the refinement of carry look-ahead to directly produce the decimal carries. These techniques offer significant improvement over the well-known method of decimal correction. They also permit the design of a parallel decimal arithmetic unit which is competitive to a binary arithmetic unit in performance and cost.

Tom´as Lang and Alberto Nannarelli [20]: presented a combinational decimal multiply unit which can be pipelined to reach the desired throughput. With respect to previous implementations of decimal multiplication, the proposed unit is combinational (parallel) and not sequential, has a simpler recoding of the operands which reduces the number of partial product pre computations and uses counters to eliminate the need of the decimal equivalent of a 4:2 adder. The results of the implementation show that the combinational decimal multiplier offers a good compromise between latency and area when compared to other decimal multiply units and to binary double-precision multipliers.

M.A erle [21]: describe the fixed-point decimal multiplication that utilize decimal carry-save addition to reduce the critical path delay. First, a multiplier that stores a reduced number of multiplicand multiples and uses decimal carry-save addition in the iterative portion of the design is presented. Then, a second multiplier design is proposed with several notable improvements including fast generation of multiplicand multiples that do not need to be stored, the use of decimal (4:2) compressors, and a simplified decimal carry- propagate addition to produce the final product. When multiplying two n-digit operands to produce a 2n-digit product, the improved multiplier design has a worst-case

latency of n + 4 cycles and an initiation interval of n + 1 cycles. Three data-dependent optimizations, which help reduce the multipliers' average latency.

R.D Kenney [22]: introduced and analyzed three different techniques for performing fast decimal addition on multiple binary coded decimal (BCD) operands. Two of the techniques cover BCD correction values and correct intermediate results while adding the input operands. The first involves single addition. The second involves two additions. The third technique 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 more significant digit. Multioperand adder designs are constructed and synthesized for four to 16 input operands. Analyses are performed on the synthesis results and the merits of each technique are discussed. Finally, these techniques are compared to several previous techniques for high-speed decimal addition.

L.Dadda [23] : Problem of Multioperand Parallel Decimal Addition with an approach that uses binary arithmetic and suggested adoption of binary-coded decimal (BCD) numbers. This involves correcting of the result in order to obtain the BCD result or a binary-to-decimal (BD) conversion. They opted for the latter approach, which is significantly efficient for a large number of addends. Conversion demands a relatively small area and allows fast operation. The BD conversion moreover creates an easy alignment of the sums of adjacent columns. BCD digit adders using fast carry-free adders and the conversion problem using a parallel scheme involving elementary conversion cells is designed. They developed spreadsheets for adding several BCD digits and for simulating the BD conversion as a design tool.

B.shirazi [24]: major advantage of the binary coded decimal (BCD) system in providing rapid binary decimal conversion has been discussed. They focused on the short coming of the BCD system is that BCD arithmetic operations, that they are slow and require complex hardware. They concluded that the performance of BCD operations can be enhanced using a redundant binary coded decimal (RBCD) representation which leads to carry-free operations. This paper introduces the VLSI design of an RBCD adder. The design consists of two small PLA's and two 4-bit binary adders for one digit of the RBCD adder. The addition delay is constant for n-digit RBCD addition (no carry propagation delay). The VLSI time and space complexities of the design as well as its layoutare presented. In addition, BCD to RBCD conversions have been carried out in a constant

time. However, RBCD to BCD conversion requires a carry-ripple operation which can be accomplished with a complexity equivalent to that of the carry-look-ahead circuitry.

CHAPTER

3

PARALLEL DECIMAL MULTIPLIER

3.1 Fixed-Point Decimal Multiplication

The decimal integer of any operand $Z = \sum_{i=0}^{d-1} Z_i 10^i$ can be represented as a positive weighted 4-bit vector as

$$Z_i = \sum_{i=0}^{3} z_{i,j} r_j \qquad (3.1)$$

Where $z_{i,j}$ is the jth bit of the ith digit , $Z_i \in [0,9]$ is the ith decimal digit and $r_i \geq 1$ is the weight of the jth bit .table 1 shows the set of coded decimal no. system that consist BCD (with $r_j = 2^j$) the other codespresented in table 3.1 used for signifying different decimal operand ,as required by the methods of represented in this paper. The codes are referred by their weight bits as (r_3, r_2, r_1, r_0).The 4-bit vector Z_i in the coded decimal number (r_3, r_2, r_1, r_0) is represented by Z_i ($r_3 r_2 r_1 r_0$).

The multiplicand $X = \sum_{i=0}^{d-1} X_i 10^i$ and multiplier $Y = \sum_{i=0}^{d-1} Y_i 10^i$ are unsigned decimal integer d-digital BCD words. Multiplication consists of the three aspects generation of partial product, reduction (addition) of partial product and finally carry propagation addition (conversion to non-redundant 2d digit BCD representation $P = \sum_{i=0}^{2i-1} p_i 10^i$ .The decimal floating-point multiplication consist of exponent addition, rounding $P = X \times Y$, sign calculation, handling and exception detection.

The decimal digits in decimal multiplication, increments the numbers of multiplicands multiplies and hence more complex than binary multiplication in representing decimal values in system based on binary logic.

For the generation of decimal partial product two approaches are used. The first approach performs a digit by digit multiplication of the input using look up table methods [8] [21]. Recently, a signed-digit radix-10 recoding (from [0, 9] to [-5, 5]) is recommended for the magnitude reduction of operand. This recommendation simplifies and speeds up the generation of partial product than, a signed-digit partial product are represented using combinational logic and simplifies tables. This methods doesn't suits with serial multiplication, since parallel partial product demands for more hardware. The next approach produces and retain all the multiplicand multiple. The multiple are then reduced by multipliers. This approach requires wide range decimal carry propagate additions to produce multiplicand multiples (3X 4X 5X 6X 7X 8X 9X).In [6] only even multiplies (2X 4X 6X 8X) are calculated and retain. Odd multiples (3X 5X 7X) are discovered on demands. A condensed set up BCD multiples (X 2X 4X 5X) is produced in [9] without the carry propagation. The multiples are obtained from this set.

First step to improve decimal multiplication is by reduction of partial product using carry propagate addition (direct decimal addition) [19]. Decimal carry save addition approach uses to present sum and carry [22], [11], [8], [7] or a BCD sum word and carry bit per digit. The first group implements decimal addition by mixing binary CSAs with combinational logic. In [7] a scheme of 3:2 binary CSAs is used to add the partial product iteratively. In order to convert BCD into decimal digit +6, +12 digit are added to obtain the carries and sum digit. The other approach uses a binary CSAs tree monitored by a single decimal corrections[22].recently in [23] uses a binary to BCD conversion and a binary carry- free adder to add N d-digit BCD operands. A second group of method [9] [20] uses 4bit radix-10 carry propagate adder to execute decimal carry save addition. In [9], a serial multiplier is executed using an array of carry look ahead adders (CLAs). A CSA tree using these radix-10 CLAs is executed in the combinational decimal multiplier.

In order to reduce all decimal partial product, efficient operand decimal tree adders are required. Most of the schemes uses parallel network with binary CSAs tree full adders. Due to simple logic cell and faster operation [22, 23]. The multioperand decimal tree adder are executed using binary CSAs tree but the operand decimal tree are coded in decimal coding then BCD. These multioperand CSAs trees are explained section**.**

Table 3.1

Decimal coding

Z_i	$Z_i(BCD)$	$Z_i(5421)$	$Z_i(4221)$	$Z_i(5211)$	$Z_i(4311)$	$Z_i(3321)$
0	0000	0000	0000	0000	0000	0000
1	0001	0001	0001	0001 0010	0001 0010	0001
2	0010	0010	0100 0010	0100 0011	0011	0010
3	0011	0011	0101 0011	0101 0110	0100	0100 1000 0011
4	0100	0100	0110 1000	0111	1000 0110 0101	1001 0101
5	0101	1000	0111 1001	1000	1001 0111 1010	1010 0110
6	0110	1001	1010 1100	1010 1001	1011	1100 1011 0111
7	0111	1010	1011 1101	1011 1100	1100	1101
8	1000	1011	1110	1110 1101	1110 1101	1110
9	1001	1100	1111	1111	1111	1111

3.2 Decimal Parallel Multipliers

In this section an overview of recommended architecture. The design consist of partial product generation and their reduction of partial product explained in section 3.3 and 3.4.these architecture uses codes for 4221 and 5221 then BCD to present partial product. This leads to reduction of decimal partial product.

3.2.1 Radix-10 Architecture

Figure 3.1 shows the architecture of d-digit SD-radix multiplier this multiplier consist of three stages generation of partial product [decimal] coded in 42211 (generation of multiplicand multiplies and SD radix-10 encoding of the multiplier),reduction of partial product and a finally carry propagate addition (BCD).The multiplier is encoded into d-SD radix-10 digit and an additional bit to generate d+1 partial products.SD radix-10 digit controls 5:1 mux to select a positive multioperand multiplies 0,X,2X,3x,4X,5X)coded in 4221.the output bits of mux is inverted by XOR gate. When the sign of SD radix-10 is negative, to find the partial product. Next, the d+1 partial product are coded (4221) decimal digit using P:2 CSAs trees explained section 3.4.4 .the digits to be condensed for each column ranges from p=d+1 to p=2.thus,the two 2d-digit operands S and H are formed from d+1 partial products.

The resultant product is 2d-digit BCD word denoted by P=2H+S. S and H are required to be progressed before being added. S is recoded from (4221) to BCD excess-6 (BCD plus 6).The H x2 multiplication is executed along with the recoding of S. The X2 requires a (4221) to (5421) digit recorder and a 1-bit wired left shift to obtain 2H. Finally BCD carry – propagate addition is obtained by using quaternary tree (Q-T) adder [14].

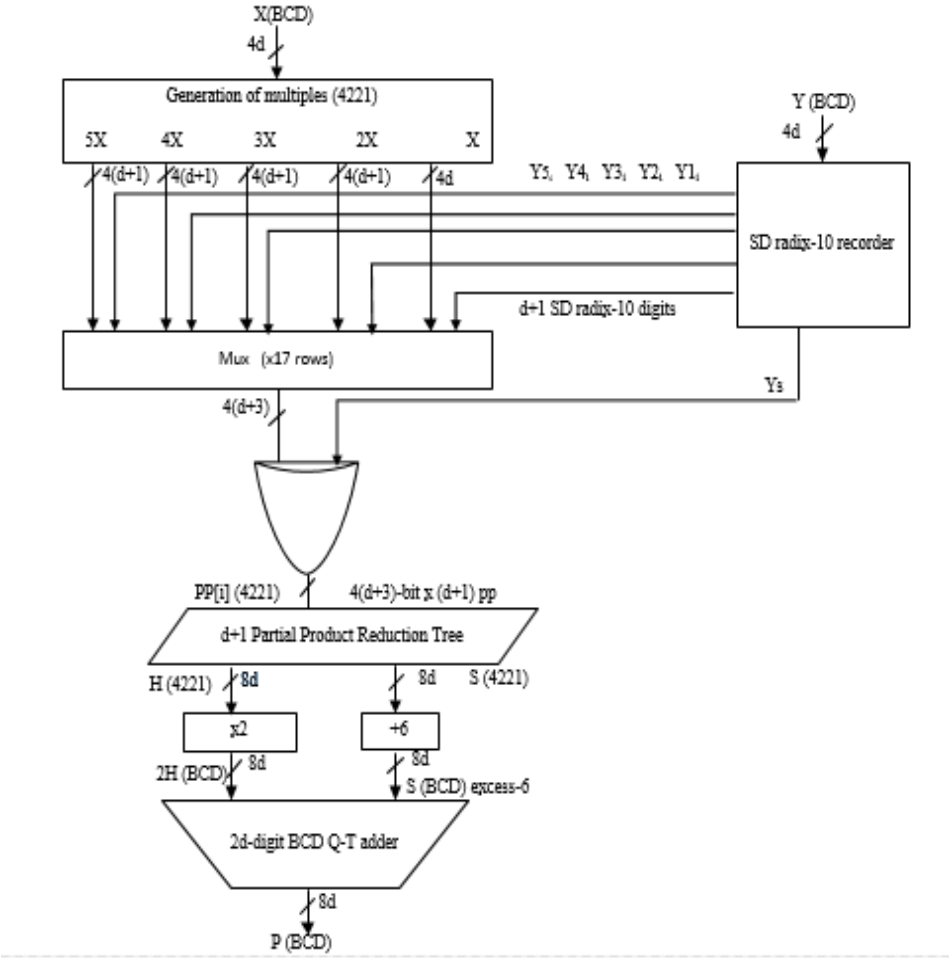


Figure .3.1: Combinational SD radix-10 architecture

3.2.1.1 SD radix-10 recoding

The block diagram of generation of partial product by using the SD radix-10 recoding. This recoding transforms a BCD digit $Y_i \in \{0,.....,9\}$ into a SD radix-10 $Yb_i \in \{-5,.....,5\}$.The value of the recoded digit Yb_i depends upon the value of Y_i and on a signal ys_{i-1} (sign signal) this shows if y_{i-1} is greater than or equal to S. Thus, the d-digit BCD multiplier Y is recoded into the d+1 digit SD radix-10 multiplier is refer by

$$Yb = \sum_{i=0}^{d} Yb_i 10^i \text{ with } Yb_d = ys_{d-1} \in \{0,1\}.$$

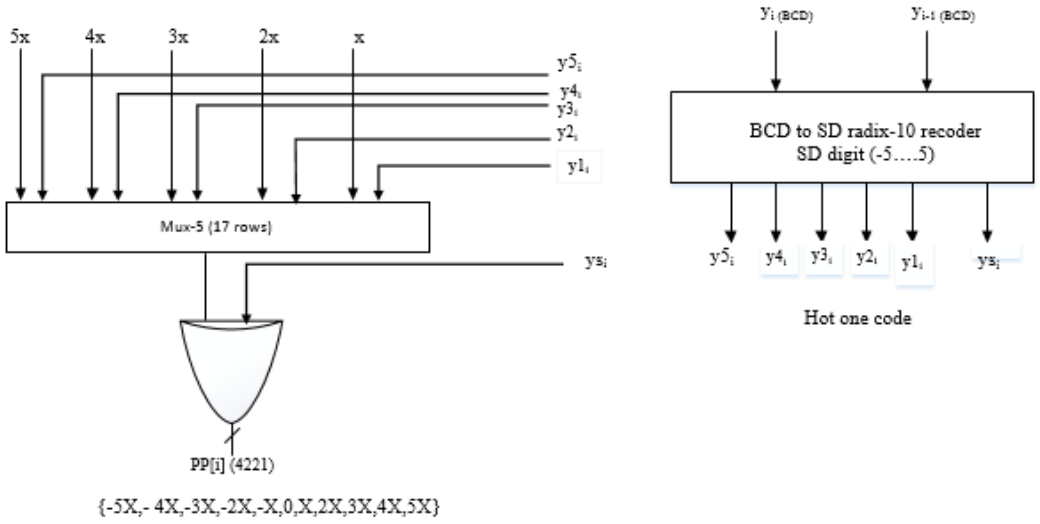


Figure 3.2: Partial product generation for SD radix-10.[2]

Each digit Yb_i generates a partial product PP $[i]$ selecting the proper multiplicand multiple coded in (4221). This is performed in a similar way to a modified booth recoding. Represented as five "hot one code" signals $\{y1_i, y2_i, y3_i, y4_i, y5_i\}$ and a sign bit ys_i .these signals are attained directly from the BCD multiplier digits Y_i using the following logical expressions:

$$ys_i = y_{i,3} \, V \, y_{i,2}.(y_{i,1} \, V \, y_{i,0}) \qquad (3.2(a))$$

$$y5_i = y_{i,2} \, V \, \overline{y_{i,1}}.(y_{i,0} \oplus ys_{i-1}) \qquad (3.2(b))$$

$$y4_i = ys_{i-1}.y_{i,0}.(y_{i,2} \oplus y_{i,1}) \, V \overline{ys_{i-1}}.y_{i,2}.\overline{y_{i,0}} \qquad (3.2(c))$$

$$y3_i = y_{i,1}.(y_{i,0} \oplus ys_{i-1}) \qquad (3.2(d))$$

$$y2_i = \overline{ys_{i-1}}.\overline{y_{i,0}}.(y_{i,0} \, V \, \overline{y_{i,2}}.y_{i,1}) \quad V \, ys_{i-1}.\overline{y_{i,3}}.y_{i,0}.\overline{y_{i,2} \oplus y_{i,1}} \qquad (3.2(e))$$

$$y1_i = \overline{y_{i,2} V y_{i,1}}.(y_{i,0} \oplus ys_{i-1}) \qquad (3.2(f))$$

Symbols V , $\cdot$, and $\oplus$ specifies Boolean operators OR, AND, and XOR, respectively. The five "hot one code" signals are used as an control signals for 5:1 muxes to select the positive d+1 digit multiples {0,X,2X,3X,4X,5X}.To find the partial product ,the

nominated positive multiple is 10's complemented if ys_i bit is one. This is done simply by a bit inversion of the positive (4221) decimal –coded multiple using a row of XOR gates that are controlled by ys_i .The accumulation of one ulp (unit in the last place) is done enclosing a tail – encoded bit ys_i (hot one) to the next significant partial product $PP[i]$. To remove a sign extension, and thus, to reduce the difficulty of the partial product reduction tree ,the Partial product sign bits ys_i are coded at every leading position into two digits as:

$$(pp[i]_{d+2}, pp[i]_{d+1}) = \left\{ \begin{array}{ll} (\overline{ys_0}, ys_0\ ys_0\ ys_0\ ys_0), & i=0, \\ (0,111\overline{ys_i}), & 0<i<d-1 \\ (0,0000), & i=d-1. \end{array} \right\} \quad (3.3)$$

Therefore, every partial product $PP[i]$ have (d+3) digit length.

3.3 Generation of multiplicand multiples

The block diagram for the generation of the positive multiplicand multiples $(X, 2X, 3X, 4X, 5X)$ for SD radix-10 recoding. All these multiples are encoded (4221). The X BCD multiplicand can be easily recoded to (4221) using the logical expressions

$$(w_{i,3}, w_{i,2}, w_{i,1}, w_{i,0}) = \left(x_{i,3}\ V\ x_{i,2}, x_{i,3}, x_{i,3}\ V\ x_{i,1}, x_{i,0}\right) \quad (3.4)$$

where, $x_{i,j}$ and $w_{i,j}$ are, respectively, the bit of the BCD and (4221) representation of X , respectively. The generation of multiples is as follows:

Multiples $2X$: Table 1 show the BCD digit recoded into (5421) decimal coding. An $L1_{shift}$ is executed to the recoded multiplicand, getting the 2X BCD multiples .Then by using expression 3.4 the 2X BCD multiples is recoded again.

Multiples $4X$ **:** It is obtained as $2X$ x2, where the $2X$ multiple is coded in (4221).The second $2X$ operation performed by is $L1_{shift}$,to encoded from (4221) to (5211).

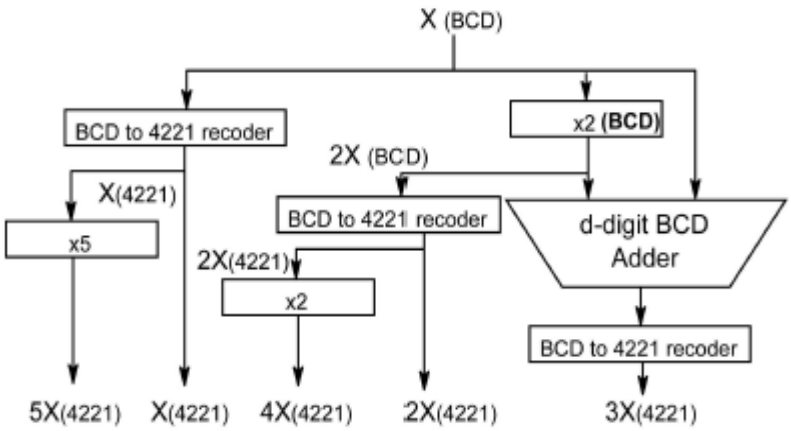


Figure 3.3: Generation of multiplicand Multiples for SD radix-10.[2]

Multiples $5X$ **:**Each BCD digit is first recoded to the (5421) decimal coding is shown in table 1.It is obtained by using $L3_{shift}$ of the (4221) multiplicand reproducing the code in 5211. Then recoded from(5221) to (4221) as shown in figure 3.3

f**Multiples** $3X$: It is obtained by a carry-propagate addition of BCD multiples X and $2X$ in a d- digit BCD adder (implemented as a quaternary- tree decimal adder [21]) by using equation 3.4 the BCD sum digit are recoded into (4421).

			$x10^2$	$x10^1$	$x10^0$
			4221	4221	4221
X (4221)	35		0000	0001	1001
		x5 ↓			
$L3_{shift}$			5211	5211	5211
$5X$ (5211)	175		0001	1100	1000
		↓			
Digit recoding			4221	4221	4221
$5X$ (4221)	175		0001	1011	1001

Figure3.4: calculation of $5X$ for decimal operands coded in (4221)

Table 3.2: Selected decimal codes for the recoded digits

Z_i	0	1	2	3	4
Z_i (4221s)	0000	0001	0010	0011	1000
Z_i (5211s)	0000	0001	0100	0101	0111
Z_i	5	6	7	8	9
Z_i (4221s)	1001	1010	1011	1110	1111
Z_i (5211s)	1000	1001	1100	1101	1111

3.3.1 Implementation of Digit Recoders

The outline of proficient digit recoders is a basic issue, due to their high effect on the area and performance of the entire multiplier. Digit recoders are utilized to process the decimal multiplicand multiples in section 3.3 and in the decrease of partial products (Section 3.4) to figure x2n (n > 0) operations.

For logical Execution of digits recoders for BCD, BCD excess 6, and (5421) decimal codes are straight forward, since there is just a mapping of decimal digits to these codes (every decimal digit has a solitary 4-bit representation). On the other hand, because of the redundancy of (4221) and (5211) decimal codes, there are a few decisions for the digit recoding to (4221) or (5211). The sixteen 4-bit vectors of a coding can be mapped (recoded) into distinctive subsets of 4-bit vectors of the other decimal coding depicting to the same decimal digit. These subsets of the (4221) and (5211) codes are additionally decimal codings.

Among all of the subsets considered, the non-redundant decimal codes (4221s) and (5211s) (subsets of ten 4-bit vectors), indicated in Table 2, present fascinating properties. Specifically, these codes verify

$$2Z(4221s) = L1_{shift}[Z(5211s)] \tag{3.5}$$

That is, subsequent to moving 1 bit to one side an operand Z depicted in (5211s), the resultant bit-vector represents to the decimal estimation of 2Z coded in (4221s). This rearranges the usage of x2n operations for n > 1. Specifically, for a decimal operand, $Z_i(4221)$, Z x 2n is implemented by a first level of $Z_i(4221s)$ to $Z_i(5211s)$ to digit

recoders followed by n - 1 levels of $Z_i(4221s)$ to $Z_i(5211s)$ digit recoders. The yield of every level of digit recoders is moved 1 bit to one side such that the most noteworthy bit of each (5211s) digit (weight 5) is moved out to the following decimal position (weight 10). Also, at times, the x2 may be rearranged. Specifically, the recoding given by Expression (4) maps the BCD representation into the subset (4221s). Consequently, the subsequent x2 operations in Figure 3.3 are implemented using a level of easier (4221s) to (5211s) digit recoders. A (4221) to (5211s) digit recoder has an equipment many-sided quality of around 27 NAND2 doors, and its basic way has (generally) the postponement of a full snake.The (4221s) to (5211s) digit recoder has a more straight forward equipment many-sided quality (around 19 NAND2 entryways) with 25 percent less inactivity. Moreover, the opposite digit recoding (from (5211) to (4221)) is effectively actualized utilizing a solitary full viper, since

$$Z_i(5211) = z_{i,3}4 + z_{i,2}2 + z^*_{i,1}2 + z^*_{i,0} \tag{3.6}$$

with $z^*_{i,1}2 + z^*_{i,0} = (z_{i,3} + z_{i,1} + z_{i,0}) \leq 3$.This recoder is utilized to produce the x5 different for the (4221) coding and in mixed (4221/5211) multioperand CSAs (Section 5.5) to change over a (5211) decimal-coded operand into the equivalent (4221) coded one.

3.4 Partial Product Reduction

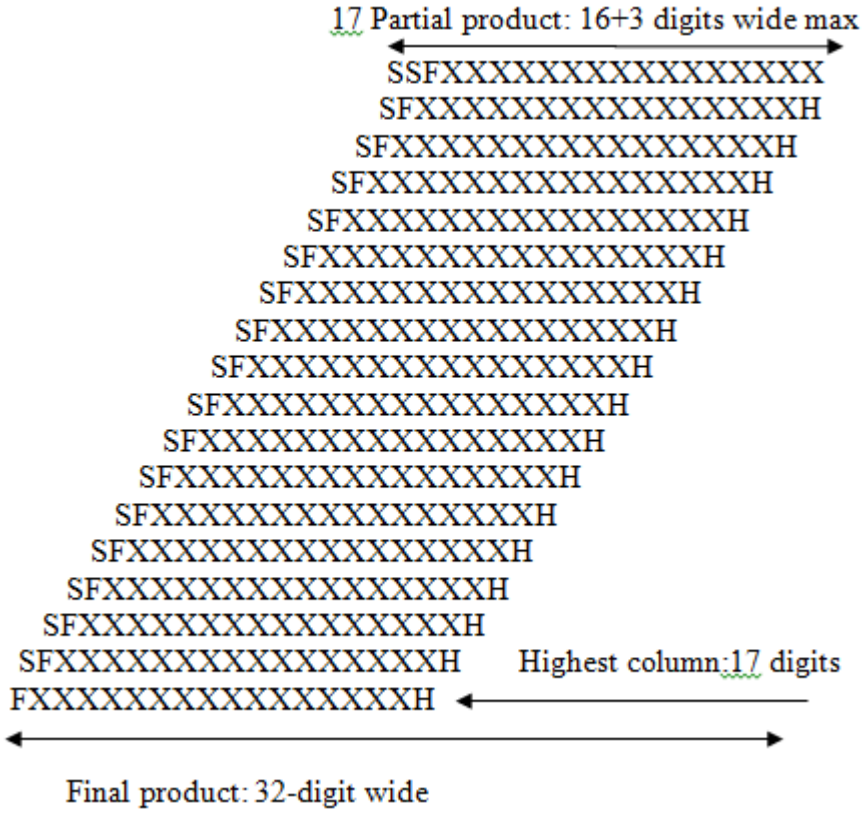
The partial product arrays are produced by the SD-radix 10 encodings. Every column of p digit is compressed to two digits by using of a decimal digit $p:2$ CSA tree. This represent the technique for the efficient reduction of decimal carry save addition CSAs or full adder, uses of (4221) and (5211) is performed instead of BCD. Thus, encoding needs decimal encoding. Decimal digit adders to also introduced in section3.4.2to reduce the delay. The digital adders also reduced the 9 digits coded in (4221) or (5221) to 4 digit coded (4221). Finally, the design of the proposed $p:2$ decimal CSA trees is implemented in the SD radix-10 in section 3.4.4.

3.4.1 Partial Product Arrays

In this section 3.2.1, the SD radix-10 architecture generates d+1 partial products coded in (4221) of d+3 digit length. These partial product $PP[i]$ are aligned according to their decimal weights by $4i$ -bit left shift $(PP[i]\,x\,10^i)$.The resultant partial product array for

16-digit input operands is shown in figure 3.7(a) .In this case, the number of digits to be reduced varies from $p = 17$ to $p = 2$. In number of digits to be reduced varies from $P = 2$. In particular, the highest columns can be reduced with the area- optimized or delay-optimized decimal $17:2$ CSA trees presented in section 3.4.4.

Every output of 3:2 CSA is multiplied by 2 before going to another 3:2 CSA input. The block x2 are connected to fast inputs of another 3:2 CSA to reduce the carry path complexity as compare to sum path. Summing the carry outputs of the last CSA before multiplied by 2 .therefore several x2 multiplication are performed in a series at the end. Both, figure 3.7, represents the same critical path delay irrespective of wiring issues.



S: Sign encoding

H: Hot one 10's complement encoding

X: Regular (4221) digit

F: Extra digit position to support the width of multiplicand multiples

Figure 3.5: Partial product arrays generated for 16-digit operands SD radix-10 architecture

3.4.2 Method for Carry Save Addition

The decimal codes obtained by expression 3.1, there is a family of codes appropriate for simple carry save addition. This family of codes proves that the 9 is the aggregate of their weighted bits, that is, which comprises of (52111), (4221), (3321) and (4321) codes as shown in Table 1.

$$\sum_{j=0}^{3} r_j = 9 \tag{3.7}$$

However, there are decimal coding which uses different context, to implement components for decimal carry save addition. These are known as redundant codes, as two or more different 4 bit vector may symbolize similar decimal digit. These codes exhibit following properties:

- A decimal digit $Z_i \in [0,9]$ is represented by sixteen 4-bit vectors.Hence, any Boolean function (AND, OR, XOR…) executing over the 4-bit vector representation of two or more input digits results a 4-bit vector that represents a valid decimal digit.

- 9's complement of a digit Z_j can be as 1's complement, that is, by inverting bits since

$$9 - Z_i = \sum_{j=0}^{3} r_j \sum_{j=0}^{3} z_{i,j}\, r_j = \sum_{j=0}^{3} (1 - z_{i,j})\, r_j$$

$$= \sum_{j=0}^{3} \overline{z_{i,j}} r_j \tag{3.8}$$

Negative operands can be calculated by inverting the bits of positive bit vector representation and summing with 1 ulp , such as

$$-Z(r_3 r_2 r_1 r_0) = \overline{Z(r_3 r_2 r_1 r_0)} + 1 \tag{3.9}$$

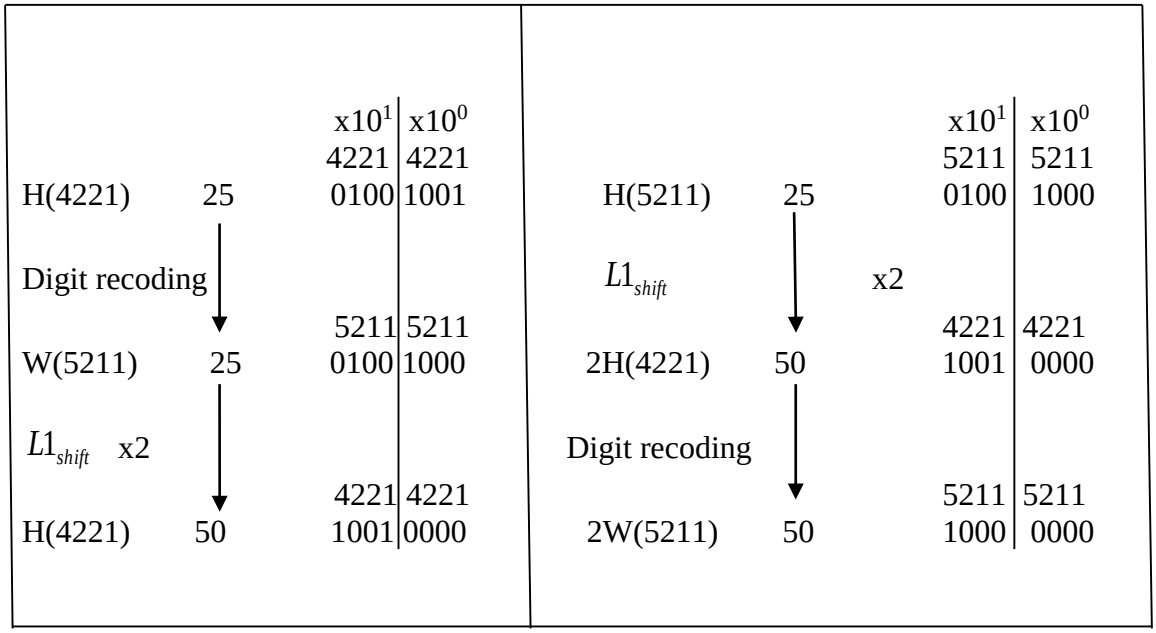
Hence fast decimal carry save addition can be performed by using conventional 4-bit binary 3:2 CSA as

$$A_i + B_i + C_i = \sum_{j=0}^{3} (a_{i,j} + b_{i,j} + c_{i,j})\, r_i$$
$$= \sum_{j=0}^{3} s_{i,j}\, r_i + 2 \times \sum_{j=0}^{3} h_{i,j}\, r_i = s_i + 2 \times H_i \qquad (3.10)$$

With $(r_3 r_2 r_1 r_0) \in \{(4221), (5211), (4311), (3321)\}$, $s_{i,j}$ and $h_{i,j}$ are the sum and carry bit of a full adder, $H_i \in [0,9]$ and $S_i \in [0,9]$ are the decimal carry and sum digits at position i. No decimal correction is needed because the 4-bit vector expression of H_i and S_i represent valid decimal digit in the selected coding.

Be that as it may, a decimal multiplication by 2 is needed before utilizing the convey digit *Hi* for later processing's. Here, the examination of decimal carry-save addition to just (5211) and (4221) decimal codes, subsequent to the generation of products of two for operands coded in (4311) and (3321) is more difficult. Figure. 8 demonstrates a sample of x2 multiplication for decimal operands written in (4221) and (5211) decimal codes. To streamline the documentation, utilize *H* and *W* for the carry vector coded in (4221) and (5211) respectively. Subsequently, the equation 3.11

$$2H = 2 \times H = L1_{shift}[W] \qquad (3.11)$$



Multiplication by 2

Figure: 3.6 Calculation of 2 for decimal operands coded in (4221) and(5211)

The resultant bit vector subsequent to moving 1 bit to the left side *W* represents to the double of *H*. The operand 2H is coded in (4221), since the weight bits of W are multiplied by 2 after the 1-bit left move. The entire $2\times$H augmentation is performed by a digit recoding of H_i into W_i took after by a $L1_{shift}[W]$. The bits of W_i are indicated by $w_{i,j}$. The bit moved out $(w_{i,3})$ indicate to a decimal carry out (weight 10) to the following digit position, while the bit moved in $(w_{i-1,3})$ is a decimal carry input (weight 1). In this manner, the digits of 2H are given by

$$(2H)_i = w_{i,2}\,4 + w_{i,1}\,2 + w_{i,0}\,2 + w_{i-1,3} \qquad (3.12)$$

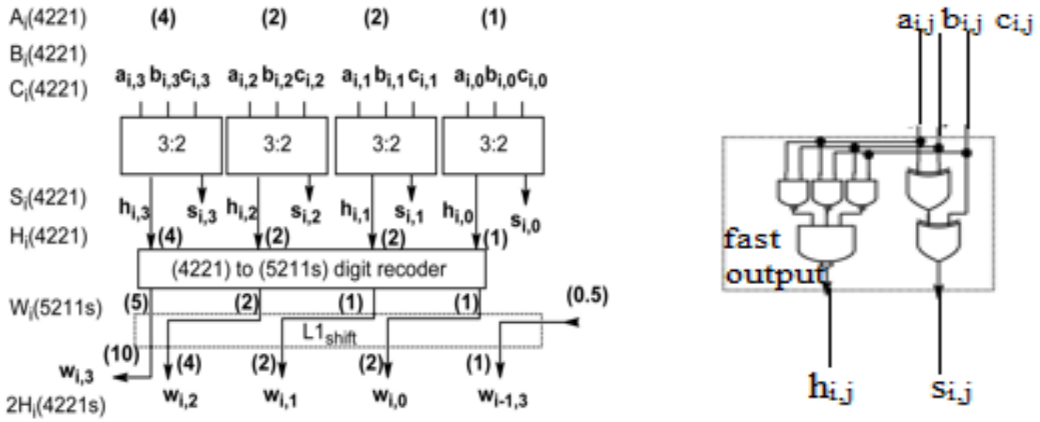
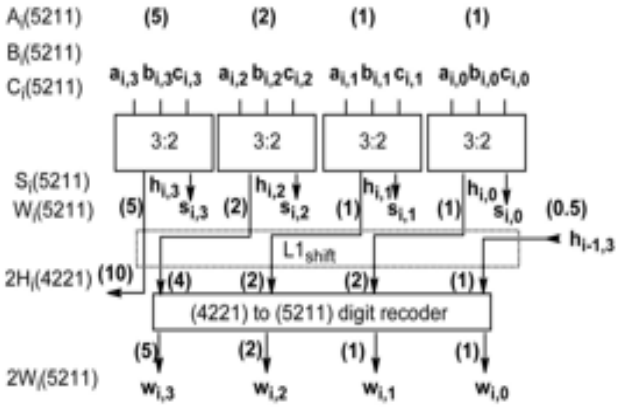
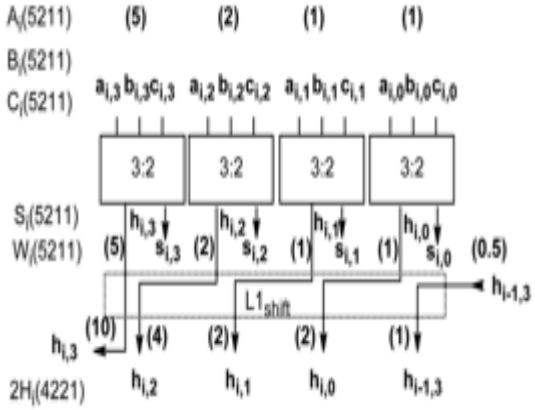


Fig 3.7: Proposed decimal 3:2 CSA for operands coded in (4221). (a) Decimal digit (4-bit) 3:2 CSA. (b) Full adder [2]

Figure 3.7 (a) demonstrates the usage of a decimal 3:2 CSA for digits coded in (4221) utilizing a 4-bit binary 3:2 CSA. The weight bits in Fig 3.7 (a) are set in parenthesis over every section. The 4-bitbinary 3:2 CSA includes three decimal digits (A_i, B_i, C_i), coded in (4221), and results in a decimal sum digit (S_i) and a carry digit H_i coded in (4221), such as $A_i + B_i + C_i = S_i + 2\times H_i$.



(a)



(b)

Figure 3.8: Proposed decimal 3:2 CSA for input operands coded in (5211). (a) Output operands coded in (5211). (b) Mixed (5211)/(4221) coded output operands.[2]

3.4.3 Decimal Digit Adders using Bit Counters

A group of fast decimal digit adders that diminishes 9, 8, or 7 digits coded in (4221) or (5211) into 4 or 3. These digit adders comprise of a row of bit counters. These bit counters total a segment of up to p=9 bits (same weight) and results a *q*-bit vector $(q = [\log_2 p] \leq 4)$ with weights (4221), which indicates to a decimal digit $Z_i \in [0,9]$. In Fig. 11a, demonstrate a usage of a bit counter that sums upto9 bits delivering a (4221) digit, which utilizes two levels of binary full adders. The binary weight of every yield is demonstrated in paranthesis. Contingent upon the data, the path delay differs generally from 2 to 4 XOR gate delays for yield (1), from 2 to 3 XORs for yield (2), and is around 2 XORs for yield (4). The 8-bit counter of Fig.3.11(b) just adds up to 8 bits however has a comparative critical path delay as a binary 4:2 CSA (3 XOR gate delays for yield (1)). Essentially, the initial two levels of half adders (HA) and the two OR gates perform the

process

$$Q0_i = q0_{i,2}4 + q0_{i,1}2 + q0_{i,0}1\sum_{k=0}^{3} x_k \tag{3.13}$$

$$Q1_i = q1_{i,2}4 + q1_{i,1}2 + q1_{i,0}1\sum_{k=0}^{3} x_k, \tag{3.14}$$

Since $Q0_i, Q1_i \in [0,4]$, the total sum $Z_i(4221) = Q1_i + Q0_i \in [0,8]$, is implemented in a simple way in the final logic level of figure.11 (b) as

$$Z_i(4221) = \begin{cases} z_{i,3} = q1_{i,2} \cdot q0_{i,2} \, V \, q1_{i,1} \cdot q0_{i,1}; \\ z_{i,2} = q1_{i,2} \cdot q0_{i,2} \, V \, q1_{i,0} \cdot q0_{i,0}; \\ z_{i,1} = q1_{i,2} \cdot q0_{i,2} \, V \, (q1_{i,1} \oplus q0_{i,1}); \\ z_{i,0} = q1_{i,0} \oplus q0_{i,0}; \end{cases} \tag{3.15}$$

In figure 3.9 (c) a conventional 7:3 binary counter is shown which diminishes 7 bits into 3-bit vector with vector (421). These counters can be utilized to diminish 9 or 8 decimal digits (coded in (4221) or (5211)) into 4, or 7 digits into 3.Anexample of this strategy is portrayed in Fig.3.10 for nine data operands coded in (5211). The methodology is comparative for (4221) input operands. A row of four (4221) decimal counters of Fig. 11a sums the estimations of every bit section creating a (4221) digit per bit section. The four bits of each (4221) digit are put in a section from the most significant (top) to the slightest significant (base) and adjusted in four rows as indicated by the weight bit of their section $(5\times10^i, 2\times10^i, 1\times10^i, 1\times10^i)$. This creates 4 decimal digits coded in (5211) that must be multiplied by an alternate element (given by the binary weights of (4221)) prior being included. This association causes all the bits of a yield operand to have the same latency. The x2 and x4 squares are executed as portrayed in Section 3.4.4. The 9:4 decimal digit adder is indicated by the marked box in Figure 3.12, where the multiplicative component of every yield is demonstrated in parenthesis. The 8:4 and 7:3 decimal digit adders are actualized utilizing a row of four counters of Figs. 11b and 11c, separately. On the other hand, the 4-bit decimal 3:2 CSA displayed in Section 3.4.1 is a 3:2 decimal digit adder executed utilizing a row of 3-bit counters (a level of full adders) with the carry yield multiplied by a factor x2.

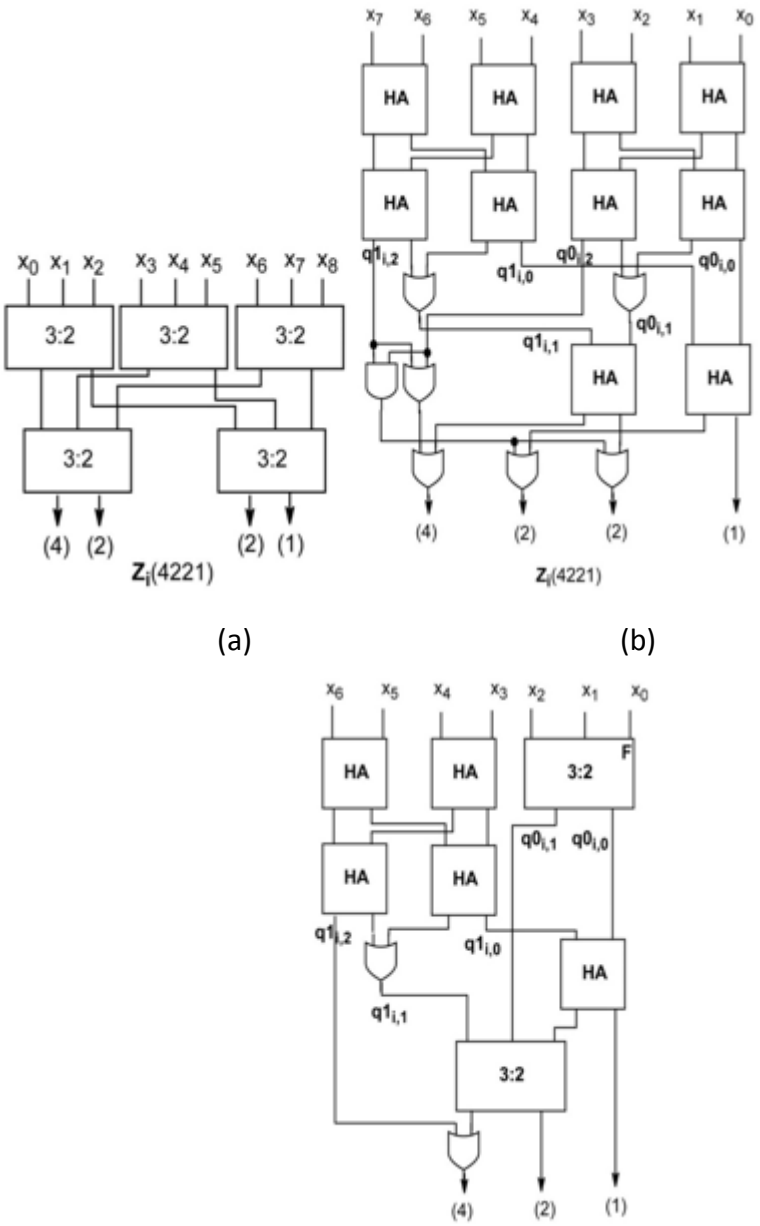


Figure3.9: Gate-level implementation of digit counters. (a) (4221) counter for 9 bits. (b) (4221) counter for 8 bits. (c) 7:3 binary counter.[2]

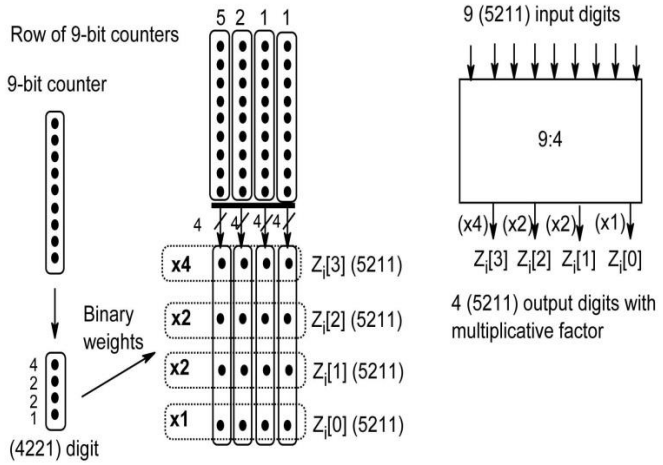
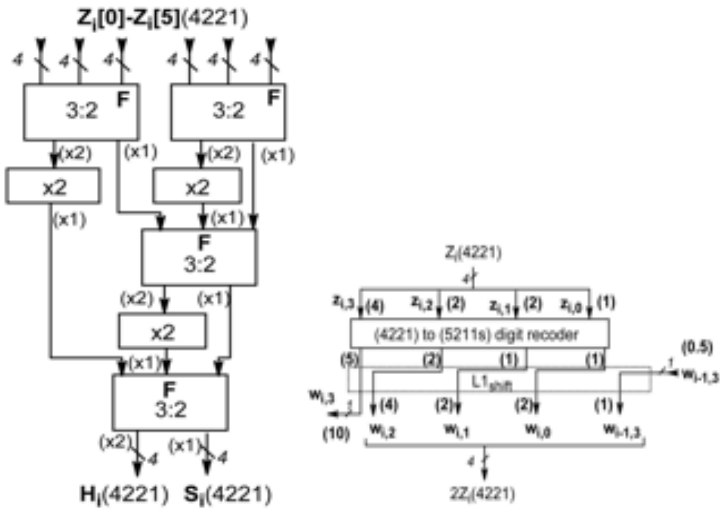


Figure 3.10: 9 : 4 reduction of (5211) decimal-coded operands.[2]

3.4.4 Decimal *P:2* CSA Trees for Digits Coded in (4221)

A decimal digit *p:2* CSA tree lessens $p\,(p \geq 9)$ input digits $Z_i \in [l]$ (with weight 10^i) coded in (4221) into two decimal digits S_i and H_i. Moreover, a few decimal carry yields are produced to the following significant decimal position (10^{i+1}) furthermore, a specific number of decimal carry inputs originate from the previous position (10^{i-1}). These decimal *p:2* CSA trees are composed as:



(a) (b)

Figure 3:11 Decimal 6:2 CSA for (4221) decimal-coded operands. (a) Digit column. (b) Implementation of a 2 block.[2]

- For p < 7, the input digits $Z_i\,[l]$ are diminished in a first level of binary 3:2 CSAs. Every carry yield digit is multiplied by 2 preceding being diminished in the following level of the binary 3:2 CSA tree. Each x2 operation produces a decimal

carry yield to the following significant digit column of the partial product array. The slowest yields are associated with fast inputs of the following binary 3:2 CSA level to adjust the aggregate delay of the distinctive path (an F demonstrates the fast input) Figure. 3.11(a) demonstrates an execution of a decimal 6:2 CSA (the multiplicative component connected with every signal is in parenthesis).The full adder setup of Figure.3.7 (b) to minimize the critical path delay of the CSA tree. The digit blocks marked x2 comprise of a (4221) to (5211s) digit recorder with the yields (for 4221 coded operands) 1-bit left moved, as indicated in Figure. 11(b).The most significant yield bit $(w_{i,3})$ indicates a decimal carry to the following digit column. To streamline the charts of the diverse decimal *p:2* CSA trees, the carries went between adjoining digit columns $(w_{i,3}, w_{i-1,3})$ are not indicated. The convey yield H_i must be multiplied by 2 preceding being acclimatized with the entirety yield S_i .

- For $p \geq 7,$ it take after diverse methodologies to acquire area-optimized or delay improved executions. For region upgraded usage, the data digits $Z_i[l]$ are decreased in a first level of binary 3:2 CSAs. Every middle operand is connected with a multiplicative variable power of 2. Operands with the same component are lessened in a binary 3:2 CSA some time recently being multiplied by this component, that is

$$2^n A + 2^n B + 2^n C = 2^n (A + B + C) = 2^n S + 2^{n+1} \times H \qquad (3.16)$$

This lessens the equipment complicated nature since the general number of x2 operations is lessened. An area-optimized decimal 17:2 CSA tree for operands coded in (4221) is demonstrated in Fig. 12(a). The x4 and x8 digit blocks create two and three decimal carry outs to the following critical digit section of the partial product array. This indicates two neighboring digit columns to show how decimal carries are passed (parallel associations).To improve the delay executions, the info digits $Z_i[l]$ are diminished in a first level of the decimal digit adders depicted in Section 3.4.3. These adders lessen 9 or 8 digits coded in (4221) or (5211) to 4 digits, and 7 digits to 3. The yield digits, coded in (4221), may have multiplicative elements of ($X4$), ($X2$), on the other hand ($X1$).

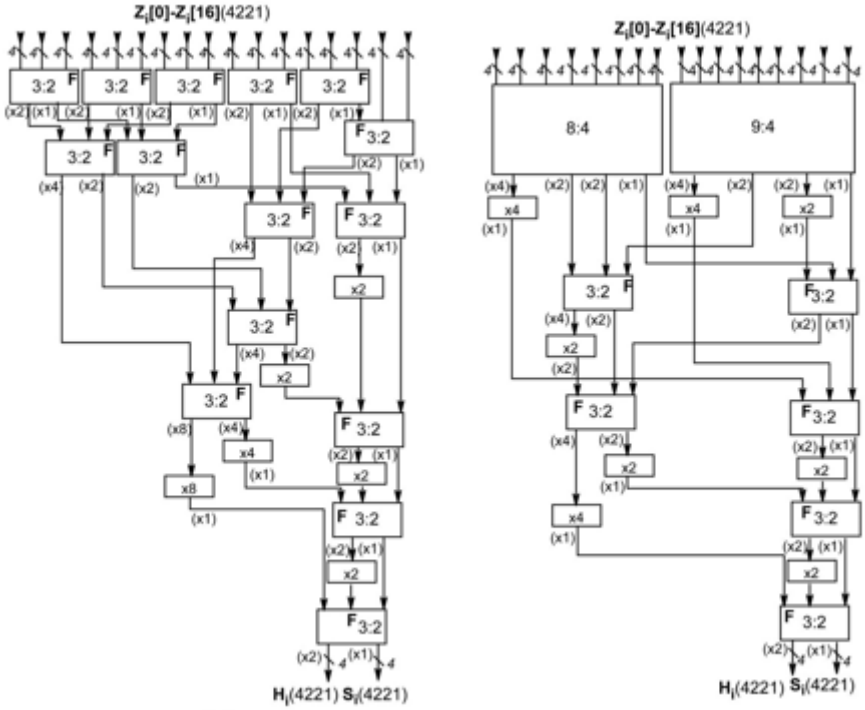


Figure 3.12: Proposed decimal 17:2 CSAs. (a) Area-optimized tree (b) Delay-optimized tree. [2]

The critical path delay is decreased by adjusting the delay of the distinctive paths. For this reason, the middle operands with higher multiplicative variables are multiplied in parallel with the diminishment of the other intermediate operands utilizing binary 3:2 CSAs. The delay improved 17:2 CSA tree in Figure. 3.12(b) has more equipment many-sided quality (identical to two x2 pieces all the more) yet the critical path is marginally speedier (around 1 XOR delay faster). Its delay is of around six levels of binary 3:2 CSAs and three levels of digit recorders. The blocks marked 9:4 and 8:4 indicates the decimal digit adders.

CHAPTER 4

FPGA IMPLEMENTATION USING XILLINX

This chapter presents about the FPGA ideas and FPGA Synthesis Flow. An FPGA is a device that comprises of thousands or even large number of transistors connected to implement logic functions. They implement functions from simple addition and subtraction to complex digital filtering and error detection and its correction.

4.1 Introduction to FPGA

A field programmable gate array (FPGA) is a semiconductor device that can be designed by the designer or the customer after manufacturing, hence it is known as "field programmable". Field Programmable gate arrays (FPGAs) are truly innovatory devices that combine the benefits of both hardware and software. FPGAs are programmed with the logic circuit diagram or the source code in Hardware Description Language (HDL) to determine how the chip will work. They may be used to perform any logical function that an Application Specific Integrated Circuit (ASIC) might perform but the capacity to update the functionality after shipping provides advantages for many applications. FPGAs contain programmable logic components also called "logic blocks", and a hierarchy of reconfigurable interconnects that permit the blocks to be "wired together" like a 1 chip programmable breadboard. Just like computer hardware, FPGAs perform calculations spatially and simultaneously computing a large number of operations in resources distributed across a silicon chip. These types of systems can be thousands of times faster than microprocessor-based designs. However, unlike in ASICs, these computations can be programmed into a chip, temporarily frozen by the manufacturing process. It means that an FPGA based design can be programmed and reprogrammed a large number of times.

4.2 FPGA Technology Trends

- Common trend is bigger and faster.

- ❖ This is achieved by increasing device density through even smaller fabrication process technology.
- ❖ New generations of FPGAs are geared towards performing entire systems on a single device.
- ❖ Features such as RAM, clock management, dedicated arithmetic hardware and transceivers are existed in addition to the main programmable logic.
- ❖ FPGAs are also available with the embedded processors.

4.3 XILLINX Specifics

All Xilinx FPGAs consists of the following basic resources

1) Configurable logic blocks (CLBs).

2) Input/output blocks (IOBs).

3) RAM blocks.

4) Programmable Interconnections (PIs).

5) Other resources like clock buffers, three-state buffers and boundary scan logic andso on.

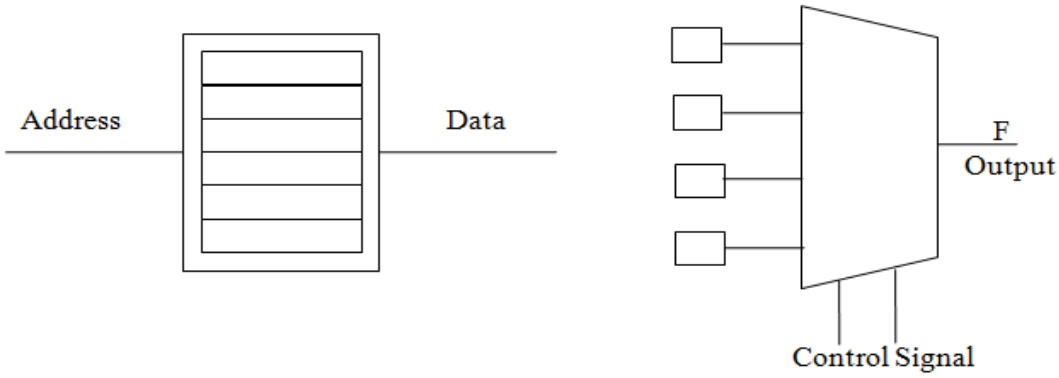


Figure 4.1: Look-up table implemented as (a) Memory (b) Multiplexers and Memory

4.3.1 Configurable Logic Blocks

The main building block of Xilinx CLBs is the slice. Spartan III holds four slices per CLB. Each slice contains two 4-input function generators (F/G), two storage elements and carry logic. Each function generator output drives both the CLB output and the D-input of a flip-flop. The look-up tables and storage elements of the CLB have the following characteristics:

1) Look-Up Tables

The way logic functions are implemented in a FPGA is another key feature. Logic blocks that carry out logical functions are look-up tables, implemented as memory, or multiplexer and memory. Figure 4.1 shows these alternatives, together with an example of memory contents for some basic operations. A 2n*1 ROM can implement any n-bit function. Typical sizes for n are 2, 3, 4, or 5. In figure 4.1(a), an n-bit LUT is implemented as a 2n*1 memory; the input address selects one of 2n memory locations. The memory locations are normally loaded with values from the user's configuration bit stream. In figure 4.1(b), the multiplexer control inputs are the LUT inputs. The result is a general-purpose "logic gate." An n-LUT can implement any n-bit function.

2) Storage Elements

The storage elements in a slice can be configured either a edge-triggered D-type flip-flops' or as level-sensitive latches. The D-input can be driven either by the function generators within the slice or directly from the slice inputs, bypassing the function generators. As well as clock and clock enable signals, each slice has also synchronous set and reset signals.

4.3.2 Input/output Blocks

The Xilinx IOB includes inputs and outputs that support a wide variety of I/O signaling standards. The IOB storage elements act either as D-type flip-flops or as latches. For each flip-flop, the set/reset (SR) signals can be independently configured as synchronous set, synchronous reset, asynchronous preset, or asynchronous clear. Pull-up and pull-down resistors and an optional weak-keeper circuit can be attached to each pad. IOBs are programmable and can be categorized as follows:

1) Input Path

A buffer in the IOB input path is routing the input signals either directly to internal logic or through an optional input flip-flop.

2) Output Path

The output path includes a 3-state output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer directly from the internal logic or through an optional IOB output flip-flop. The 3-state control of the output can also be routed directly from the internal logic or through a flip-flop that provides synchronous enable and disable signals.

3) Bidirectional Block

This can be any combination of input and output configurations.

4.3.3 Ram Blocks

Xilinx FPGA incorporates several large RAM memories (block selects RAM). These memory blocks are organized in columns along the chip. The number of blocks ranging from 8 up to more than 100, depending on the device size and family.

4.3.4 Programmable Routing

Adjacent to each CLB stands a general routing matrix (GRM). The GRM is a switch matrix through which resources are connected; the GRM is also the means by which the CLB gains access to the general-purpose routing. Horizontal and vertical routing resources for each row or column include:

- Long Lines: Bidirectional wires that distribute signals across the device. Vertical and horizontal long lines span the full height and width of the device.
- Hex Lines: Route signals to every third or sixth block away in all four directions.
- Double Lines: Route signals to every first or second block away in all four directions.
- Direct Lines: Route signals to neighboring blocks vertically, horizontally & diagonally.
- Fast Lines: Internal CLB local interconnections from LUT outputs to LUT inputs.

4.4 FPGA Implementation Using XILINX

The FPGA that is used for the implementation of the circuit is the Xilinx Spartan 6E(Family), XC3S5000 (Device). The working environment/tool for the design is the Xilinx ISE 14.2i is used for FPGA Design flow of VHDL code.

4.4.1 Overview of FPGA Design Flow

As the FPGA architecture evolves and its complexity increases. Today, most FPGA vendors provide a fairly complete set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down to a bit stream to program FPGA chips. Atypical FPGA design flow includes the steps and components shown in Figure 4.2.

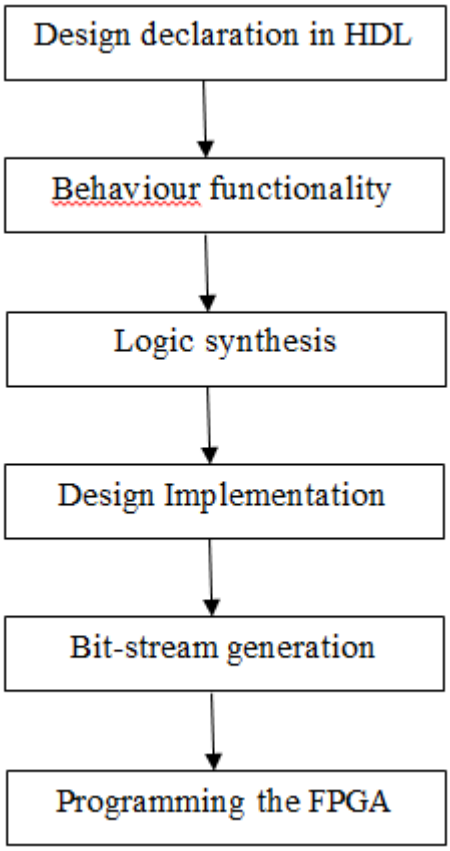


Figure 4.2: FPGA Design Flow

In some cases, delays between some specific pairs of registers may be constrained. The second design input component is the choice of FPGA device. Each FPGA vendor typically provides a wide range of FPGA devices, with different performance, cost and power tradeoffs. The selection of target device may be an iterative process. The designer may start with a small (low capacity) device with a nominal speed-grade. But, if synthesis effort fails to map the design into the target device, the designer has to upgrade to a high-capacity device. Similarly, if the synthesis result fails to meet the operating frequency, he has to upgrade to a device with higher speed-grade. In both the cases, the cost of the FPGA device will increase in some cases by 50% or even by 100%. This clearly underscores the need to have better synthesis tools since their quality directly impacts the performance and cost of FPGA [25]. The FPGA implementation of designs is done to check their functionality on actual hardware. The cost of implementation and design cycle time of ASICs are large therefore the bigger designs are first checked on FPGA and if they give satisfactory results then there ASIC implementation is done.

Figure 4.2 shows the different steps involved in the FPGA implementation. It includes design entry through HDL, behavior simulation, logic synthesis, design implementation,

bit stream generation and finally programming the FPGA. Xilinx provides all the functions necessary for implementing a design on FPGA. The Xilinx ISE suite includes ISIM simulator for functional and timing simulation, XST for synthesis applications, Xpower analyzer for estimating power and chip scope pro analyzer for debug and verification purposes. The different steps involved in FPGA implementation using Xilinx are described below:

4.4.1.1 Design Entry

The basic architecture of the system is designed in this step which is coded in a Hardware description Language like Verilog or VHDL. A design module is split into two parts, each of which is called a design unit in Verilog. The module declaration represents the external interface to the design module. The module internals represents the internal description of the design module-its behavior, its structure, or a mixture of both.

4.4.1.2 Behavioral Simulation

After the design phase, create a test bench waveform containing input stimulus to verify the functionality of the Verilog code module using simulation software i.e. Model sim ISE for different inputs to generate outputs and if it verifies then precede further, otherwise modifications and necessary corrections will be done in the HDL code. This is called as the behavioral simulation.

4.4.1.3 Design Synthesis

After the correct simulations results, the design is then synthesized. During synthesis, the Xilinx ISE tool does the following operations: HDL Compilation: The tool compiles all the sub-modules in the main module if any and then checks the syntax of the code written for the design.

1) Design Hierarchy Analysis: analysis of the hierarchy of the design.

2) HDL Synthesis: The process which translates VHDL or Verilog code into a device net list format, i.e. a complete circuit with logical elements such as Multiplexer, Adder/subtractions, counters, registers, flip flops Latches, Comparators, XORs, decoders, etc. for the design. If the design contains more than one sub designs, ex. to implement a processor, a CPU as one design element and RAM as another and so on are needed, and then the synthesis process generates net list for each design element.

Synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture, the designer has selected.

The resulting net list is saved to an NGC (Native Generic Circuit) file (for Xilinx Synthesis Technology (XST)). Figure 4.3 shows the complete process of synthesis from HDL code to NGC file generation.

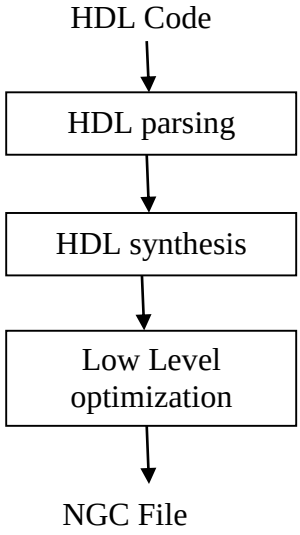


Figure 4.3: Steps in synthesis process

3) Advanced HDL Synthesis (Low Level synthesis): The blocks synthesized in the HDL synthesis are further defined in terms of the low level blocks such as buffers, lookup tables. The tool then generates a 'net list' file (NGC file) and then optimizes it. The final net list output file has an extension of NGC. This NGC file contains both the design data and the constraints.

4.4.1.4 Design Implementation

Design implementation process consists of the following sub processes

1) Translation:In this process all the input net lists and design constraints are combined and saved in a file called as native generic database file. The ports available in design are assigned to physical elements of the target device. Timing requirements of the design are also specified in translate process. Translate properties can also be changed by modifying them.

2) Mapping: After translate process mapping is done. In mapping the circuit is divided into sub-blocks. The sub-blocks are made so that they can fit into FPGA sub-blocks. A file is generated called as native circuit description file. This file contains our design mapped into components of FPGA.

3) Place and Route: Sub-blocks of map process are converted into logic blocks and connected in place and route step. This process takes NCD file as input and outputs the routed NCD file. Here placement and routing of blocks is done.

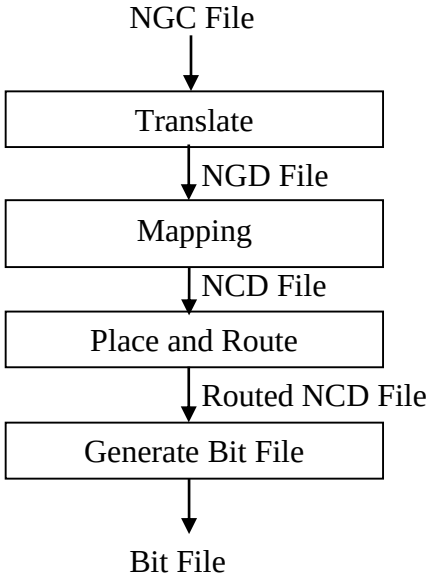


Figure 4.4: Different files generated in implementation process

4) Bit stream Generation:In this process bit file is generated for particular Xilinx device from the routed NCD file. The output bit file contains binary bits necessary to program the device. Sometimes this process is also called as bit-stream generation. The generated bit file is used to program the FPGA device.

5) Functional Simulation: Post-Translate (functional) simulation can be performed prior to mapping of the design. This simulation process allows the user to verify that the design has been synthesized correctly and any differences due to the lower level of abstraction can be identified.

4.5 Analyzing Design Using Chip Scope Pro

The FPGA designs are becoming more complex, due to need of faster designs and shorter design times. Debugging and verification is important factor in determining the complete design time. It takes almost 50% of the design time. But the Xilinx chip scope pro software performs faster debugging and verification. It shrinks overall design by 25%. It is a powerful tool that is easy to use. It is used for debug, verification and inserting short signal sequences. Chip scope pro uses FPGA resources like block RAM for trigger and data storage, slice logic for trigger comparison. It uses three types of flows i.e. core generator, core inserter and plan-ahead flow. The core inserter flow is similar to plan-

ahead flow and provided in plan-ahead software. In core generator flow the core is instantiated in source HDL, while in core inserter flow the core is inserted into generated file after synthesis. Different types of cores are used by chip scope pro software like ICON core, ILA core, VIO core, IBA core. Some of them are explained below:

- ICON core: It is used to control up to 15 capture cores. It acts as interface between JTAG interface and capture cores. The main function of this core is to control the other cores. It can be used in both the core generator and core inserter flows.
- VIO core: It defines and generates virtual i/os. It is used to apply stimulus and read outputs transition on the node that wants to be selected. This core is used to generate virtual input and outputs. It provides options for synchronous and asynchronous inputs and outputs, where each can have width of 256 bits. There is also option of clock. It can be system clock or JTAG clock The outputs of the design which are wanted to be implement are connected to the input of the VIO core and inputs of the design are connected to the output of the VIO core therefore the inputs are virtual LEDs and outputs are virtual DIP Switches. This core is controlled by the ICON core. So a control port is also provided. VIO core uses FPGA logic not RAM. It is only used in core generator flow.
- ILA core: It is a capture core. It can be used to create custom triggers when activated causes data to be stored during circuit operations. Signals can be stored depending on the condition specified by used. A design can contain up to 15 ILA cores.
- Agilent trace core: It used to store large amount of data off chip or when customer uses Agilent analyzer. ILA with Agilent trace is similar to ILA except data is captured off chip or by Agilent trace port analyzer.

CHAPTER 5

SIMULATION RESULTS AND SYNTHESIS

This chapter discusses about the implementation of parallel decimal multiplier, and their synthesis and simulation results. Synthesis report describes actual hardware utilization and timing constraints of the design. Simulation results describe the behavioural functionality of the design. Implementation of the design is done to transfer the design on to actual hardware. The software used for the simulation, synthesis and implementation is Xilinx ISE 14.5 targeting Spartan -6E FPGA. The working environment for all the designs is

- Tool version : ISE 14.5
- Optimization Goal : Speed
- Design Strategy : Balanced
- Family : Spartan 6E
- Device : XC6SLX45
- Speed : 3
- Package: CSG324
- Simulator : ISIM
- Total slices: 27288
- Total LUTs: 54576

5.1 Radix-10 parallel decimal multiplier

In radix-10 decimal multiplier design, the partial product are generated by using the SD radix-10 recoding. On calculating the generation of decimal partial products coded in (4221) generation of multiplicand multiples and SD radix-10 encoding of the multiplier,

partial product reduction and finally BCD carry propagate addition Using these approach, the Radix-10 decimal multiplier is designed in VHDL. Then the multiplier is simulated and synthesized using Xilinx ISE 14.5 targeting Spartan 6 FPGA device.

5.1.1 Simulation Results for Radix-10 Parallel Decimal multiplier

The design is simulated using ISIM simulator. Figure 5.1 shows the simulation results for Radix-10 Parallel decimal multiplier. Considering the following are the inputs in BCD format.

Clk = changing 1 or 0 (for 10ns with period 1ns) is the clock

A=0001000100010001000100010001000100010001000100010001000100010001000100010 (for 10ns) is the (Multiplicand)

B=0001000100010001000100010001000100010001000100010001000100010001000100010 (for 10ns) is the (Multiplier)

The output obtained after calculating the generation of decimal partial products coded in (4221) generation of multiplicand multiples and SD radix-10 encoding of the multiplier ,partial product reduction and finally BCD carry propagate addition.

Result=0000000100100011010001010110011110001011111100100101100010111111001000100010001000100010001000100010000111011001010100001100100001000

The obtained results on the ISIM simulator are similar to the desired results as shown in figure below figure 5.1.

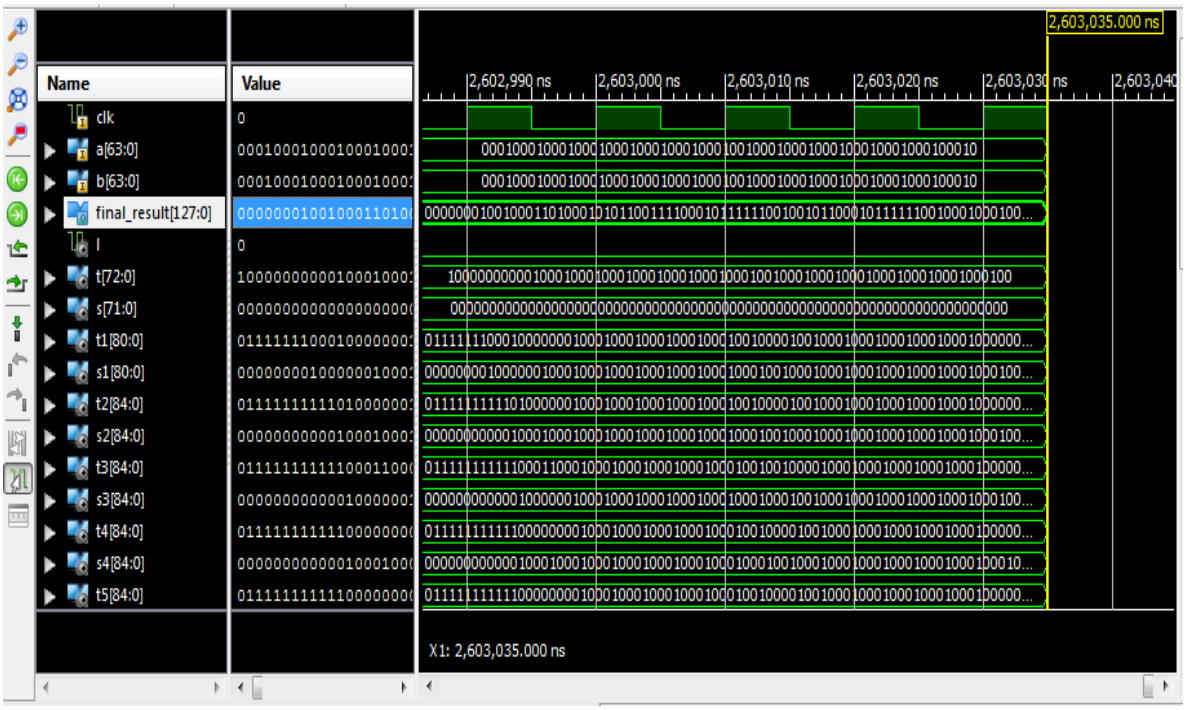


Figure 5.1: Simulation Results for Radix-10 Parallel Decimal Multiplier

5.1.2 Synthesis Results for Radix-10 Parallel Decimal multiplier

Table 5.1 shows the total delay and area utilization summary in terms of number of slices The 16-digit SD radix-10 (Figure 3.1) combinational multiplier have been synthesized using Model sim SE6.5c. For partial product reduction, the SD radix- 10 multiplier implements area-optimized decimal p:2 CSA trees, similar to the decimal 17:2 CSA tree.

Table 5.1: Synthesis report of radix-10 parallel decimal multiplier

Architecture	**Delay (min. period)**	**Area (no. of slice LUTS)/ 27288**
DEC. SD radix-10	32.028	5347
Ref. 2	48	12500
Ref. 20	55.8	16000
Ref. 27	58.9	17638

5.1.3 Comparison Results

We have analyzed several decimal carry-free tree adders for sixteen 64-bit operands based on different method. We have evaluated and compared these implementations with our proposals using the area and delay evaluation model previously described. We have also compared the area and delay figures obtained from synthesis of representative proposals of Decimal multipliers (sequential and parallel) and a binary radix-4 parallel multiplier. From the synthesis report, it is clear that the area utilization and total delay of the radix-10 decimal multiplier with carry save adder is less than the ripple carry adder. So if carry save adder is used at each stage of the multiplier then it gives better performance in terms of delay rather than using ripple carry adder. The area and delay values measured for the SD radix-10 multiplier 2.30 ns (32.028) no. of slice LUTs and 5347 (area) to 2.88 ns (58.8) no. of slice LUTs and 17638 (area) . These figures 5.2 and 5.3 agree with the table 5.1 obtained from our area and delay evaluation model.

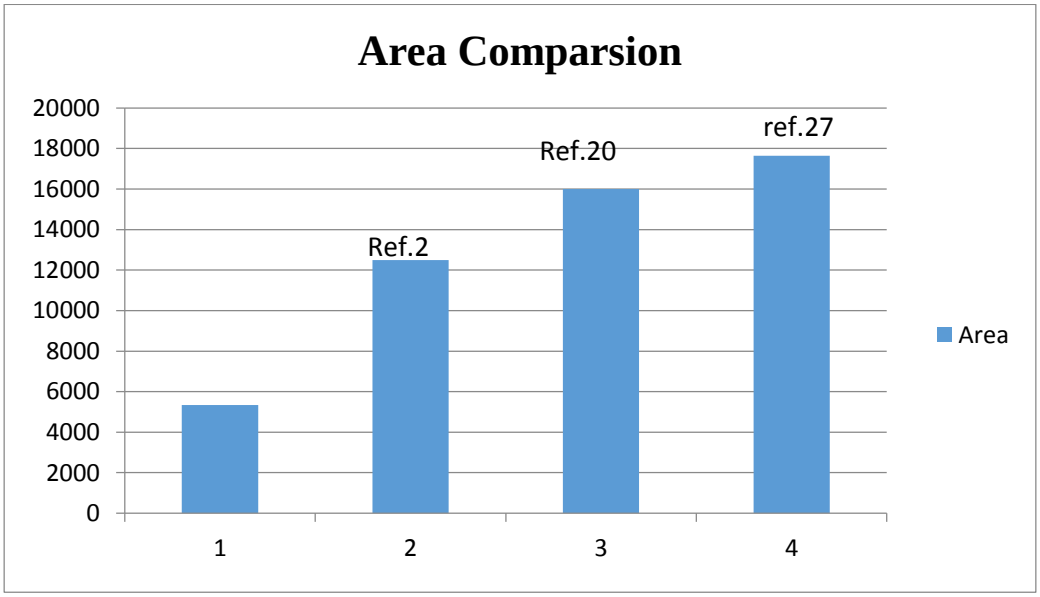


Figure 5.2: Area comparison of radix-10 decimal multiplier

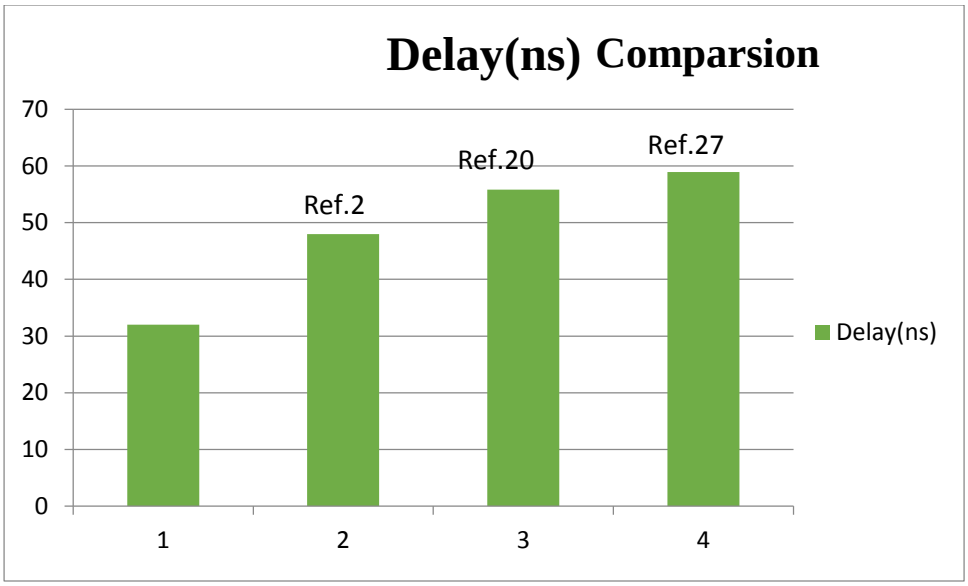


Figure 5.3: Delay comparison of radix-10 decimal multiplier

6 CONCLUSION AND FUTURE SCOPE

In this dissertation, the SD radix-10 decimal multiplier is implemented. With the usage of the SD encoding implied for developing the multiplier, it was possible to generate fast , parallel and simple generation of partial products. This work proposes a decimal carry save algorithm based on unconventional (4221) and (5211) decimal encoding for partial product reduction. It makes the construction of p:2 decimal CSA trees possible, that further helps to outperform the area and delay figures of existing proposals. Architecture for decimal SD radix-10 parallel decimal multiplication Evaluation results for 16-digit operands show that the proposed architectures has interesting area-delay figures compared to conventional Booth radix-4 and radix-8 parallel binary multipliers and outperform the figures of previous decimal multiplication. From comparative study including conventional binary parallel multiplier and other representative decimal proposals it is observed that our decimal SD radix-10 multiplier is an interesting option for high performance with moderate area. The total delay has been reduced from (32.028) number of slice LUTs, while the area is 5347 in this scheme and 1763 in the previous proposed work [2,20,27].

6.2 Future Scope

The present work on Radix-10 parallel decimal multiplier can be extending in various directions. Some of the suggestions as given below:

1. Different encoding method can be analyzed to optimize the speed and area.
2. In place of carry save adder, other adders such as carry select adder and carry look ahead adder can be used to increase the performance.

3. Different compressors can be used for accumulation of partial products so that delay can be further reduced.
4. In order to enhance the performance higher order compressors like 17:2 can be used to accumulate the partial products.

REFERENCES

[1] M.F. Cowlishaw, "Decimal Floating-Point: Algorism for Computers," *Proc. 16th IEEE Symp. Computer Arithmetic*, pp. 104-111, July 2003

[2] Vazquez, A., Antelo, E., and Montuschi, P: 'Improved design of high performance parallel decimal multipliers', *IEEE Trans. Comput.*, 2010,59, (5), pp. 679–693

[3] L. Eisen et al., "IBM POWER6 Accelerators: VMX and DFU,"*IBM J. Research and Development*, vol. 51, no. 6, pp. 663-684, Nov. 2007.

[4] N. Ohkubo et al., "A 4.4 ns CMOS 54x54-Bit Multiplier Using Pass-Transistor Multiplexer,"*IEEE J. Solid State Circuits*, vol. 30, no. 3, pp. 251-256, Mar. 1995.

[5] F.Y. Busaba, C.A. Krygowski, W.H. Li, E.M. Schwarz, and S.R. Carlough, "The IBM z900 Decimal Arithmetic Unit,"*Proc. Conf. Record of the Asilomar Conf. Signals, Systems and Computers, vol. 2*, pp. 1335-1339, Nov. 2001.

[6] F.Y. Busaba, T. Slegel, S. Carlough, C. Krygowski, and J.G. Rell, "The Design of the Fixed Point Unit for the z990 Microprocessor*," Proc. 14th ACM Great Lakes Symp*. VLSI 2004, pp. 364-367, Apr. 2004.

[7] T. Ohtsuki et al., "Apparatus for Decimal Multiplication," *US Patent 4,677,583*, June 1987

[8] R.H. Larson, "High-Speed Multiply Using Four Input Carry-Save Adder," *IBM Technical Disclosure Bull.*, vol. 16, no. 7, pp. 2053-2054, Dec. 1973.

[9] M.A. Erle and M.J. Schulte, "Decimal Multiplication via Carry- Save Addition," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors*, pp. 348-358, June 2003.

[10] M.A. Erle, M.J. Schulte, and B.J. Hickman, "Decimal Floating- Point Multiplication via Carry-Save Addition*," Proc. 18th IEEE Symp. Computer Arithmetic*, pp. 46-55, June 2007.

[11] R.D. Kenney, M.J. Schulte, and M.A. Erle, "High-Frequency Decimal Multiplier," *Proc. IEEE Int'l Conf. Computer Design:* VLSI in Computers and Processors, pp. 26-29, Oct. 2004.

[12] Va´zquez, E. Antelo, and P. Montuschi, "A New Family of High-Performance Parallel Decimal Multipliers,"*Proc. 18th IEEE Symp. Computer Arithmetic*, pp. 195-204, June 2007.

[13] Kaivani A, Liu Han and Seok -Bum Ko,*"*Improved design of high- frequency sequential decimal multiplier," *Electronics Letters*, Vol. 50 No. 7, pp. 558–560, March 2014.

[14] Vazquez, A., and Antelo, E.: 'Conditional speculative decimal addition*'. Proc. 7th Conf. Real Numbers and Computers (RNC7),* France, July 2006, pp. 47–57.

[15] Cowlishaw .M.F, "Decimal Floating-Point: Algorism for Computers," *Proceedings of the 16thIEEE Symposium on Computer Arithmetic*, pp. 104-111, June 2003.

[16] Erle, M.A., Schwarz, E.M., and Schulte, M.J.: 'Decimal multiplication with efficient partial product generation*'*. Proc.*17th IEEE Symp. Computer Arithmetic,* USA, June 2005, pp. 21–28.

[17] A Kaivani , Chen, L., and Ko, S.: 'High-frequency sequential decimal Multipliers'. *Proc. IEEE Int. Symp. Circuits and Systems,* South Korea , May 2012, pp. 3045–3048.

[18] Jaberipur. G and A. Kaivani, "Binary-Coded Decimal Digit Multipliers," *IET Computer and Digital Techniques*, Vol. 1, No. 4, pp. 377-381, july 2007.

[19] Schmookler .M and Weinberger. A, "High Speed Decimal Addition,"*IEEE Trans. Computers, vol.* 20, no. 8, pp. 862-866, Aug. 1971.

[20] T. Lang and A. Nannarelli, *"*A Radix-10 Combinational Multiplier*,"Proc. 40th Asilomar Conf. Signals, Systems, and Computers*, pp. 313-317, Oct. 2006.

[21] T. Ueda, "Decimal Multiplying Assembly and Multiply Module,"*US Patent 5379245*, Jan. 1995.

[22] R.D. Kenney and M.J. Schulte, "High-Speed Multioperand Decimal Adders," *IEEE Trans. Computers*, vol. 54, no. 8, pp. 953- 963, Aug. 2005.

[23] L. Dadda, "Multioperand Parallel Decimal Adder: A Mixed Binary and BCD Approach*,"IEEE Trans. Computers,* vol. 56, no. 10, pp. 1320-1328, Oct. 2007.

[24] B. Shirazi, D.Y.Y. Yun, and C.N. Zhang, "RBCD: Redundant Binary Coded

Decimal Adder," *Proc. IEE Conf. Computers and Digital Techniques*, vol. 136, pp. 156-160, Mar. 1989.

[25] M.A. Erle, J.M. Linebarger, and M.J. Schulte, "Potential Speed up Using Decimal Floating-Point Hardware," *Proc. 36th Asilomar Conf. Signals, Systems and Computers*, pp. 1073-1077, Nov. 2002.

[26] M.A. Erle, M.J. Schulte, and B.J. Hickman, "Decimal Floating-Point Multiplication via Carry-Save Addition," *Proc. 18th IEEE Symp. Computer Arithmetic*, pp. 46-55, June 2007.

[27] L. Dadda and A. Nannarelli, "A Variant of a Radix-10 Combinational Multiplier," *Proc. IEEE Int'l Symp. Circuits and Systems (ISCAS '08*), pp. 3370-3373, May 2008.

[28] IEEE Std 754(TM)-2008, IEEE Standard for Floating-Point Arithmetic, *IEEE CS*, Aug. 2008.

[29] A. Svoboda, "Decimal Adder with Signed-Digit Arithmetic," *IEEE Trans. Computers*, vol. 18, no. 3, pp. 212-215, Mar. 1969.

[30] G.S. White, "Coded Decimal Number Systems for Digital Computers*," Proc. Institute of Radio Engineers*, vol. 41, no. 10, pp. 1450-1452, Oct. 1953

[31] M. Morris Mano, "Computer System Architecture," Pearson Education India, 3rd edition