IMPROVED DESIGN OF HIGH-PERFORMANCE PARALLEL DECIMAL MULTIPLIERS

By

MANEESHA.V.P

Reg. No. 1020106012

of

KUMARAGURU COLLEGE OF TECHNOLOGY

(An Autonomous Institution affiliated to Anna University, Coimbatore)

COIMBATORE - 641049

A PROJECT REPORT

Submitted to the

FACULTY OF ELECTRONICS AND COMMUNICATION ENGINEERING

In partial fulfillment of the requirements
for the award of the degree
of
MASTER OF ENGINEERING
IN
APPLIED ELECTRONICS
APRIL 2012

BONAFIDE CERTIFICATE

Certified that, this project report entitled "**DESIGN OF HIGH-PERFOMANCE PARALLEL DECIMAL MULTIPLIERS**" is the bonafide work of Mrs.Maneesha.V.P [Reg. no. 1020106012] who carried out the project under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Project Guide	Head of the Department
Prof.K.Ramprakash	Dr. Ms. Rajeswari Mariappan

The candidate with university Register no. 1020106012 is examined by us in the project viva-voce examination held on …………………….

Internal Examiner	External Examiner

ACKNOWLEDGEMENT

I express my profound gratitude to our director **J.Shanmugham.** for giving this opportunity to pursue this course

At this pleasing moment of having successfully completed the project work, I wish to acknowledge my sincere gratitude and heartfelt thanks to our beloved Principal **Prof.Ramachandran,** for having given me the adequate support and opportunity for completing this project work successfully.

I extend my heartfelt thanks to my internal guide **Prof.K.Ramprakash,** for his ideas and suggestion, which have been very helpful for the completion of this project work. His careful supervision has ensured me in the attaining perfection of work.

I express my sincere thanks to **Dr.Rajeswari Mariyappan Ph.D.,** the ever active, Head of the Department of Electronics and Communication Engineering, who rendering us all the time by helps throughout this project

In particular, I wish to thank and everlasting gratitude to the project coordinator **Asst.Prof.R.Hemlatha**, Department of Electronics and Communication Engineering for her expert counseling and guidance to make this project to a great deal of success.

Last, but not the least, I would like to express my gratitude to my family members, friends and to all my staff members of Electronics and Communication Engineering department for their encouragement and support throughout the course of this project.

ABSTRACT

The new generation of high-performance decimal floating-point units demands efficient implementations of parallel decimal multipliers. In this paper we discuss about the implementation of the decimal parallel multipliers used in the decimal floating-point units. Here we discuss two architectures using SD radix-5 and SD radix-10 encoding of the multiplier with which the partial products are generated and a multioperand carry save algorithm is used for the reduction of the partial products. The proposed method allows the reuse of the binary CSA for computing the sum of BCD operands. Corrections required for decimal operands are done in parallel, separately from the calculation of the binary sum such that the layout of the binary carry save adder is not rearranged. 16 digit adders while implemented using the proposed architectures gives excellent are area-delay values when compared with the conventional binary multipliers.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

DFP	-------	**Digital Floating Point**
SD	-------	**Signed Digit**
BCD	-------	**Binary Coded Decimal**
CSA	-------	**Carry Save Adder**
CLA	-------	**Carry Look-ahead Adder**
CPL	-------	**Complementary Pass-transistor Logic**
NMOS	-------	**N-type Metal Oxide Semiconductor**
PMOS	-------	**P-type Metal Oxide Semiconductor**
MCIT	-------	**Multiplexing Control Input Technique**
VLSI	-------	**Very Large Scale Integration**
FPGA	-------	**Field Programmable Gate Array**
ASIC	-------	**Application Specific Integrated Circuits**
CPLD	-------	**Complex Programmable Logic Device**
SoC	-------	**System-On-Chip**
VHDL	-------	**Very High Speed Integrated Circuit Hardware Description Language**

CHAPTER 1
INTRODUCTION

The microprocessor manufacturers include decimal floating–point units in their products, oriented to mainframe servers, to satisfy the high performance demands of current financial, commercial and user–oriented applications. Providing hardware support for decimal floating-point (DFP) arithmetic is becoming a topic of interest. Although software DFP implementations satisfy the precision requirements, they are about an order of magnitude slower than hardware implementations and could not satisfy the high-performance demands. Specifically, the revision of the IEEE 754 Standard for Floating-Point Arithmetic (IEEE 754-2008) incorporates specifications for DFP arithmetic that can be implemented in software, hardware, or in a combination of both. An important and frequent operation in decimal computations is multiplication. However, due to the inherent in-efficiency of decimal arithmetic implementations in binary logic, practically most of the proposed decimal multipliers are sequential units. Parallel multipliers are used extensively in most of the binary floating–point units and are of interest for decimal applications to scale performance.

1.1 PROJECT GOAL

An important and frequent operation in decimal computations is multiplication. However, decimal multiplication is more difficult to implement due to the complexity in the generation of multiplicand multiples and the inefficiency of representing decimal values in systems based on binary signals. These issues complicate the generation and reduction of partial products. Thus, while decimal adders are implemented in a parallel fashion and are almost as efficient as binary ones, commercial implementations of decimal multipliers are sequential. The goal of this project is to introduce two novel architectures which are fully combinational for fixed point parallel decimal multipliers. We also describe new techniques for partial product generation and reduction that can be implemented in combined binary/decimal floating point units so as to reduce the latency and the hardware complexity of the previous designs

1.2 OVERVIEW

In this project, we describe the architectures of two parallel decimal multipliers. The parallel generation of partial products is performed using signed-digit radix-10 or radix-5 recodings of the multiplier and a simplified set of multiplicand multiples. The reduction of partial products is implemented in a tree structure based on a combined decimal/binary multioperand carry-save addition algorithm that uses unconventional (non BCD) decimal-coded number systems. The synthesis results of the 16 bit operands of the proposed architecture (combined binary/decimal multiplier) will be compared with existing decimal multiplier architectures (SD radix-10 and SD radix-5) and the binary multipliers in terms of area, delay, power consumption.

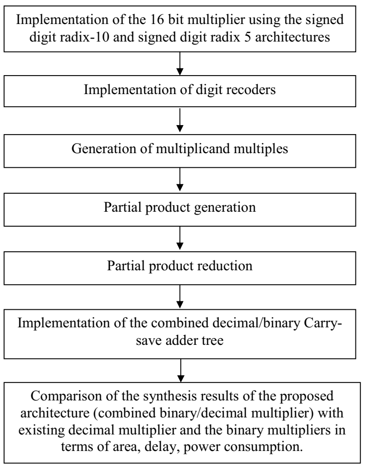

Implementation of the 16 bit multiplier using the signed digit radix-10 and signed digit radix 5 architectures
Implementation of digit recoders
Generation of multiplicand multiples
Partial product generation
Partial product reduction
Implementation of the combined decimal/binary Carry-save adder tree
Comparison of the synthesis results of the proposed architecture (combined binary/decimal multiplier) with existing decimal multiplier and the binary multipliers in terms of area, delay, power consumption.

Figure 1: Project Flow

1.3 SOFTWARE USED

- ModelSim XE 111 6.2g
- Xilinx ISE 9.2i

1.4 ORGANIZATION OF THE REPORT

- **Chapter 2** discusses about fixed point decimal multiplication.
- **Chapter 3** briefs about fixed point decimal architectures.
- **Chapter 4** explains about partial product generation.
- **Chapter 5** discusses about partial product reduction.
- **Chapter 6** describes in detail about the proposed architecture.
- **Chapter 7** mentions about the radix-4 binary multiplier
- **Chapter 8** shows the evaluation results.
- **Chapter 9** provides the conclusion and future scope.

CHAPTER 2
FIXED-POINT DECIMAL MULTIPLICATION

Multiplication consists of three stages: generation of partial products, fast reduction (addition) of partial products to a two operand and a final carry propagate addition. Decimal multiplication is more complex than binary multiplication mainly for two reasons: the higher range of decimal digits ([0, 9]), which increments the number of multiplicand multiples and the inefficiency of representing decimal values in systems based on binary logic using BCD–8421 (since only 9 out of the 16 possible 4–bit combinations represent a valid decimal digit). These issues complicate the generation and reduction of partial products

2.1 AN OVERVIEW OF DECIMAL MULTIPLICATION

A digit Z_i of a decimal integer operand $Z = \sum_{i=0}^{d-1} Z_i\ 10^i$ is coded as a positive weighted 4-bit vector as

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

Where, Zi ∈ [0,9] is the ith decimal digit and $z_{i,j}$ is the jth bit of the ith digit, and rj ≥ 1 is the weight of the jth bit. The previous expression represents a set of coded decimal number systems that includes BCD (with $r_j = 2^j$), shown in Table 1. The other decimal codes shown in Table 1 are also used for representing different decimal operands as required by the methods used in the project. These codes are represented by their weight bits as $(r_3r_2r_1r_0)$. The 4-bit vector that represents the decimal digit Z_i in a decimal code $(r_3r_2r_1r_0)$ is denoted by $Z_i(r_3r_2r_1r_0)$.

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-digit BCD words. Fixed-point multiplication (both binary and decimal) consists of three stages

- generation of partial products
- reduction (addition) of partial products to two operands
- final conversion (usually a carry propagate addition) to a non-redundant 2d-digit BCD representation $P = \sum_{i=0}^{2d-1} P_i\ 10^i$.

Table 1: Decimal Codings

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	0001	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 1101	1110 1101	1110 1101	1110
9	1001	1100	1111	1111	1111	1111

Extension to decimal floating-point multiplication involves exponent addition, rounding of P = X × Y to fit the required precision, sign calculations and exception detection and handling.

In Table 1 diverse BCD codings are represented. For BCD–8421, $r_j = 2^j$. BCD–4221 and BCD–5211 are the coding schemes characterized by the use of redundancy in decimal digit representation. As we have mentioned, the use of BCD–8421 to represent decimal digits means introducing costly decimal corrections in the partial product reduction binary CSA tree to obtain the correct decimal carry and sum. To avoid these corrections we use the BCD–4221 coding of Table 1 to represent partial product digits which will be later discussed in detail.

2.2 EXISITING METHODS

Proposed methods for the generation of decimal partial products follow mainly two approaches. The first alternative performs a digit by digit multiplication of the input operands, using digit by digit lookup table methods .In this magnitude range reduction of the operand digits by a signed-digit radix-10 recoding (from [0, 9] to [-5,5]) is suggested. This recoding of both operands speeds up and simplifies the generation of partial products. Then, signed-digit partial products are generated using simplified tables and combinational logic. This class of methods is only suited for serial implementations, since the high hardware demands make them impractical for parallel partial product generation. The second approach generates and stores all the required multiplicand multiples. Next, multiples are distributed to the reduction stage through multiplexers controlled by the BCD multiplier digits ([0; 9]).This approach requires several wide decimal carry-propagate additions to generate some complex BCD multiplicand multiples {3X,6X,7X,8X,9X}. Usually only even multiples {2X; 4X; 6X; 8X} are computed and stored. Odd multiples {3X; 5X; 7X; 9X} are obtained on demand. A reduced set of BCD multiples {X; 2X; 4X; 5X} is pre-computed without a carry propagation. All the multiples can be obtained from the sum of two elements of this set. The other alternative is that 2X and 5X multiples can be computed in few levels of combinational logic. Negative multiples require an additional 10's complement operation.

Decimal carry-save addition methods use two BCD words to represent sum and carry or a BCD sum word and a carry bit per digit. The first group implements decimal addition mixing binary CSAs with combinational logic for decimal correction. A scheme of two levels of 3:2 binary CSAs is used to add the partial products iteratively. Since it uses BCD to represent decimal digits, a digit addition of +6 or +12 (modulo 16) is required to obtain the decimal carries and to correct the sum digit. In order to reduce the contribution of the decimal corrections to the critical path, three different techniques for multioperand decimal carry-save addition were proposed. Two of them perform BCD corrections (+6 digit additions) using combinational logic and an array of binary carry-save adders (speculative adders), although a final correction is also required. A sequential decimal multiplier based on these techniques uses BCD invalid combinations (overloaded BCD representation) to simplify the sum digit logic. The other approach basically a non-speculative adder uses a binary CSA tree followed by a single decimal correction. Among these proposals, the non-speculative adders present the best area-delay figures and are suited for tree topologies. Another recent proposal uses a binary carry-free tree adder and a subsequent binary to BCD conversion to add up to N d-digit BCD operands.

The second group of methods uses different topologies of 4-bit radix-10 carry-propagate adders to implement decimal carry-save addition. A serial multiplier can be implemented using an array of radix-10 carry look-ahead adders (CLAs). A CSA tree using these radix-10 CLAs is implemented in the using combinational decimal parallel

multiplier and to optimize the partial product reduction, they also use an array of decimal digit counters.

The reduction of all decimal partial products in parallel requires the use of efficient multioperand decimal tree adders. Among the different schemes, the most promising ones for fast parallel addition seem to be those using binary CSA trees or some parallel network of full adders due to their faster and simpler logic cells (full adders against SD adder cells or radix-10 CLAs). These methods assume that decimal digits are coded in BCD. However, BCD is highly inefficient for implementing decimal carry save addition using binary arithmetic because of the need to correct the invalid 4-bit combinations (those not representing a decimal digit). The previous methods use different schemes to perform these BCD corrections. Moreover, the BCD carry digit must be multiplied by 2, which requires additional logic. We also implement multioperand decimal tree adders using a binary CSA tree, but with operands coded in decimal codings that are more efficient than BCD, namely (4221) or (5211). These multioperand decimal CSA trees are detailed in later chapters.

CHAPTER 3
DECIMAL FIXED- POINT ARCHITECTURES

In this chapter we present a general overview of the architectures for d-digit (4d-bit) BCD decimal fixed-point parallel multiplication. These designs are based on the techniques for partial product generation and reduction as mentioned earlier. The main feature of these architectures is the use of codes (4221) and (5211), instead of BCD, to represent the partial products. This improves the reduction of decimal partial products with respect to other proposals, in terms of both area and latency.

3.1 Signed Digit Radix-10 Architecture

The architecture of the d-digit SD radix-10 multiplier is shown in Fig. 2. The multiplier consists of the following stages: generation of decimal partial products coded in (4221) (generation of multiplicand multiples and SD radix-10 encoding of the multiplier), reduction of partial products, and a final BCD carry-propagate addition.

The generation of the *d + 1* partial products is performed by an encoding of the multiplier into d SD radix-10 digits and an additional leading bit. Each SD radix-10 digit controls a level of 5:1 muxes, which selects a positive multiplicand multiple {0;X; 2X; 3X; 4X; 5X} coded in (4221). The generation of these multiples is detailed in Section 4.3. To obtain each partial product, a level of XOR gates inverts the output bits of the 5:1 muxes when the sign of the corresponding SD radix-10 digit is negative.

Before being reduced, the *d+ 1* partial product, coded in (4221), are aligned according to their decimal weights. Each p-digit column of the partial product array is reduced to two (4221) decimal digits using one of the decimal digit *p:2* CSA trees described in chapter 5. The number of digits to be reduced for each column varies from *p = d+1* to *p = 2*. Thus, the *d+1* partial products are reduced to two *2d* digit operands S and H coded in (4221).

The final product is a *2d*-digit BCD word given by P = 2H + S. Before being added, S and H need to be processed. S is recoded from (4221) to BCD excess-6 (BCD value plus 6, which requires practically the same logical complexity as a recoding to

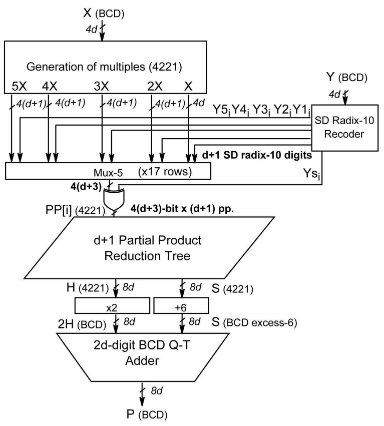


Figure 2: Combinational SD radix-10 architecture.

BCD). The H × 2 multiplication is performed in parallel with the recoding of S. This ×2 block uses a (4221) to (5421) digit recoder and a 1-bit wired left shift to obtain the operand 2H coded in BCD.

For the final BCD carry-propagate addition, we use a quaternary tree (Q-T) adder based on conditional speculative decimal addition. It has low latency (about 10 percent more than the fastest binary adders) and requires less hardware than other alternatives.

3.2 Signed Digit Radix-5 Architecture

The dataflow of the d-digit SD radix-5 architecture is shown in Fig. 3. The multiplier consists of the following stages: generation of decimal partial products (generation of multiplicand multiples and SD radix-5 encoding of the multiplier),

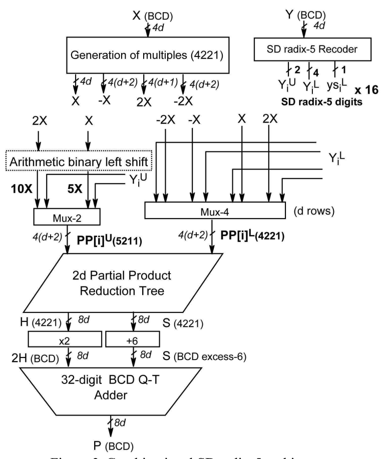


Figure 3: Combinational SD radix-5 architecture.

reduction of partial products, and a final BCD carry-propagate addition. SD radix-5 recoding, described in Section 4.2, generates 2d decimal partial products, half coded in (4221) and the other half in (5211). This improved scheme only requires the generation of simple multiplicand multiples {-2X;-X;X; 2X} coded in (4221), as shown in Section 4.2. The reduction of the aligned partial products is carried out using the mixed (4221/5211) decimal digit *p:2* CSA trees ($2 \le p \le 2d$) described in chapter 5. As in the SD radix-10 architecture, the 2d-digit operands S and H are processed before being assimilated in the 2d-digit BCD carry-propagate adder.

CHAPTER 4
DECIMAL PARTIAL PRODUCT GENERATION

For simplified multiplication we aim for parallel generation of a reduced number of partial products coded in (4221) or (5211). This is achieved with the recoding of the *d*- digit BCD multiplier and the generation of a reduced and simple set of multiplicand multiples. We present two different schemes with good trade-offs between fast generation of partial products and the number of partial products generated. A minimally redundant SD radix-10 recoding of the multiplier (with digits in {-5; . . . ; 0; . . . ; 5}) produces only *d+1* partial products but requires a carry-propagate addition to generate complex multiples 3X and -3X. A second scheme, named SD radix-5 recoding, encodes each BCD digit Y_i of the multiplier into two digits $Y_i^U \in \{0; 1; 2\}$; $Y_i^L \in \{-2;-1; 0; 1; 2\}$ such that $Y_i = Y_i^U \bullet 5 + Y_i^L$. It generates *2d* partial products (2 digits per radix-10 digit), but all multiplicand multiples are produced in a few levels of combinational logic. Furthermore, the (4221) and (5211) codes are self-complementing. Thus, an advantage with respect to previous schemes, which use BCD multiples, is that the 9's complement of each digit can be obtained by inverting its bits. This simplifies the generation of the negative multiplicand multiples from the positive ones. In addition, the previous methods based on the decomposition $Y_i = Y_i^U \bullet 5 + Y_i^L$ require combinational logic to generate the 5X multiple. We use mixed (4221/5211) decimal codings to remove this logic.

4.1 SD Radix-10 Recoding

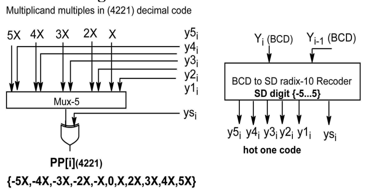


Figure 4: Partial product generation for SD radix-10.

Fig. 4 shows the block diagram of the generation of one partial product using the SD radix-10 recoding. This recoding transforms a BCD digit $Y_i \in \{0; . . . ; 9\}$ into an SD radix-10 $Yb_i \in \{-5; . . . ; 5\}$. The value of the recoded digit Yb_i depends on the decimal value of Y_i and on a signal ys_{i-1} (sign signal) that indicates if Y_{i-1} is greater than or equal to 5. Thus, the *d*-digit BCD multiplier Y is recoded into the *d+1*- digit SD radix-10 multiplier $Yb = \sum_{i=0}^{d} Yb_i\, 10^i$ with $Yb_d = ys_{d-1} \in \{0,1\}$.

Table 2: SD radix-10 selection signals.

Dec Value	BCD Y_i	$Y_{i-1} \ge 5$ ys_{i-1}	SD radix-10 digit Yb_i	Hot one code signals $ys_i\ y5_i\ y4_i\ y3_i\ y2_i\ y1_i$
0	0000	0	0	000000
		1	1	000001
1	0001	0	1	000001
		1	2	000010
2	0010	0	2	000010
		1	3	000100
3	0011	0	3	000100
		1	4	001000
4	0100	0	4	001000
		1	5	010000
5	0101	0	-5	110000
		1	-4	101000
6	0110	0	-4	101000
		1	-3	100100
7	0111	0	-3	100100
		1	-2	100010
8	1000	0	-2	100010
		1	-1	100001
9	1001	0	-1	100001
		1	0	100000

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: Y bi is represented as five "hot one code" signals $\{y1_i; y2_i; y3_i; y4_i; y5_i\}$ and a sign bit ys_i. "Hot one code" refers to a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0).These signals are obtained directly from the BCD multiplier digits Y_i using the following logical expressions:

$$ys_i = y_{i,3} \vee y_{i,2} \bullet (y_{i,1} \vee y_{i,0})$$
$$y5_i = y_{i,2} \bullet \overline{y_{i,1}} \bullet (y_{i,0} \vee ys_{i-1})$$
$$y4_i = ys_{i-1} \bullet y_{i,0} \bullet (y_{i,2} \oplus y_{i,1}) \vee \overline{ys_{i-1}} \bullet y_{i,2} \bullet \overline{y_{i,0}}$$
$$y3_i = y_{i,1} \bullet (y_{i,0} \vee ys_{i-1})$$
$$y2_i = \overline{ys_{i-1}} \bullet \overline{y_{i,0}} \bullet (y_{i,3} \vee \overline{y_{i,2}} \bullet y_{i,1}) \vee ys_{i-1} \bullet \overline{y_{i,3}} \bullet y_{i,0} \bullet \overline{y_{i,2} \oplus y_{i,1}}$$
$$y1_i = \overline{y_{i,2} \vee y_{i,1}} \bullet (y_{i,0} \vee ys_{i-1})$$

Symbols $\vee$, $\bullet$, and $\oplus$ indicate Boolean operators OR, AND, and XOR, respectively. The five "hot one code" signals are used as selection control signals for the 5:1 muxes to select the positive *d+1*- digit multiples {0;X; 2X; 3X; 4X; 5X}. The generation of the positive multiples {X; 2X; 3X; 4X; 5X} coded in (4221) from the BCD multiplicand is detailed in Section 4.3. To obtain the correct partial product, the selected positive multiple is 10's complemented if ys_i is one. This is performed simply by a bit inversion of the positive (4221) decimal-coded multiple using a row of XOR gates controlled by ys_i. The addition of one *ulp* (unit in the last place) is performed enclosing a tail-encoded bit ys_i (hot one) to the next significant partial product PP[*i*+1], since it is shifted a decimal position to the left from PP[*i*]. To avoid a sign extension, and thus, to reduce the complexity of the partial product reduction tree, the partial product sign bits ys_i are encoded at each leading position into two digits as

$$(PP\,[i]_{d+2}\,,\,\underline{PP}\,[i]_{d+2}) \;=\; \begin{cases} (ys_0.\ ys_0\, ys_0\, ys_0\, ys_0), & i = 0, \\ (0{,}111\ ys_i), & 0 < i < d\text{-}1, \\ (0,\ 0000), & i = d\text{-}1. \end{cases}$$

Therefore, each partial product PP[i] is at most of *(d+3)* - digit length.

4.2 SD Radix-5 Recoding

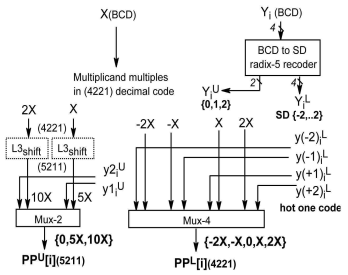


Figure 5: Partial product generation for SD radix-5.

Fig. 5 shows the diagram for partial product generation using the SD radix-5 recoding scheme. Each BCD digit of the multiplier is encoded into two digits $Y_i^U \in \{0; 1; 2\}$ and $Y_i^L \in \{-2,-1,0,1,2\}$ so that $Y_i = Y_i^U \bullet 5 + Y_i^L$. SD radix-5 "hot one code" selection signals are obtained from the BCD input digits using the following equations

$$Y_i^U \begin{cases} y2_i^U = y_{i,3}; \\ y1_i^U = y_{i,2} \vee y_{i,1} \bullet y_{i,0}; \end{cases}$$

$$Y_i^L \begin{cases} y(+2)_i^L = y_{i,1} \bullet (\overline{y_{i,2}} \bullet \overline{y_{i,0}} \vee y_{i,2} \bullet y_{i,0}) \\ y(+1)_i^L = \overline{y_{i,3}} \bullet \overline{y_{i,2}} \bullet y_{i,1} \bullet y_{i,0} \vee \overline{y_{i,2}} \bullet y_{i,1} \bullet y_{i,0} \\ y(-1)_i^L = y_{i,3} \bullet \overline{y_{i,0}} \vee \overline{y_{i,2}} \bullet y_{i,1} \bullet y_{i,0} \\ y(-2)_i^L = \overline{y_{i,3}} \bullet \overline{y_{i,0}} \vee y_{i,2} \bullet y_{i,1} \bullet y_{i,0} \end{cases}$$

Each multiplier digit Y_i generates two partial products $PP[i]^U$ and $PP[i]^L$. Therefore, this scheme generates *2d* partial products for a *d*-digit multiplier. The advantage of this recoding is that it uses a simple set of multiplicand multiples {-2X,-X,X, 2X} coded in (4221). This decimal partial product generation is comparable in latency to binary Booth radix-4, due to a faster generation of multiples.

Table 3: SD radix-5 selection signals.

Dec	BCD	Recoded Bits				Hot one signals				sign
Value	(Y_i)	Y_i^U	Y_i^L	$y2_i^U$	$y1_i^U$	$y(+2)_i^L$	$y(+1)_i^L$	$y(-1)_i^L$	$y(-2)_i^L$	ys_i^L
0	0000	0	0	0	0	0	0	0	0	0
1	0001	0	1	0	0	0	1	0	0	0
2	0010	0	2	0	0	1	0	0	0	0
3	0011	1	-2	0	1	0	0	0	1	1
4	0100	1	-1	0	1	0	0	1	0	1
5	0101	1	0	0	1	0	0	0	0	0
6	0110	1	1	0	1	0	1	0	0	0
7	0111	1	2	0	1	1	0	0	0	0
8	1000	2	-2	1	0	0	0	0	1	1
9	1001	2	-1	1	0	0	0	1	0	1

Moreover, the generation of $PP[i]^U$ (positive) only requires multiples {X, 2X}.. To obtain the correct value of $PP[i]^U$, the multiples selected by Y_i^U must be first multiplied by 5. This is performed by shifting 3 bits to the left the bit vector representation of the (4221) coded multiples {X, 2X}, producing, respectively, the multiples {5X; 10X} but coded in (5211). We denote by Lm_{shift} a left arithmetic binary shift of *m* bits, implemented with fixed wiring. The negative multiples {-X;-2X} are obtained by bit inverting the multiples {X; 2X}, coded in (4221), and adding an *ulp* as a hot one in the corresponding partial product. The sign bits ys_i^L, given by

$$ys_i^L \quad = \quad y_{i,3} \vee y_{i,2} \bullet \overline{y_{i,1}} \bullet \overline{y_{i,0}} \vee \overline{y_{i,2}} \bullet y_{i,1} \bullet y_{i,0};$$

are encoded to the left of $PP[i]^L$ and $PP[0]^U$ as

$$PP[i]_{d+1}^L = \begin{cases} (1,1,1,\overline{ys_i^L}) & \text{if } (0 \leq i < d-1) \\ (0,0,0,0) & \text{if } (i = d\text{-}1) \end{cases}$$

$$PP[0]_{d+1}^U = (0,0,0,ys_0^L)$$

The hot ones produced by the 10's complement of the partial products, $(0,0,0,ys_i^L)$are just placed in the least significant digit of $PP[i]^U$ and $PP[i]_0^U$which have a value of 0 or 5 coded

in (5211). The 2d partial products generated are at most of *d+2-* digit length, *d* of them coded in (5211) ($PP[i]^U$) and the other half in (4221) ($PP[i]^L$).

4.3 Generation of Multiplicand Multiples

All the required decimal multiplicand multiples, except the 3X multiple, are obtained in a few levels of combinational logic using different digit recoders and performing different fixed m-bit left shifts (Lm_{shift}) in the bit-vector representation of operands. The structure of these digit recoders is discussed in Section 4.4.

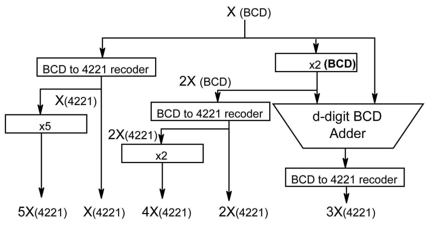


Figure 6: Generation of multiplicand multiples for SD radix-10.

Fig. 6 shows the block diagram for the generation of the positive multiplicand multiples {X, 2X, 3X, 4X, 5X} for the SD radix-10 recoding. All these multiples are coded in (4221). The X BCD multiplicand is easily recoded to (4221) using the logical expressions

$$(w_{i,3}, w_{i,2}, w_{i,1}, w_{i,0}) = (x_{i,3} \vee x_{i,2}; x_{i,3}; x_{i,3} \vee x_{i,1}; x_{i,0});$$

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

Multiple 2X: Each BCD digit is first recoded to the (5421) decimal coding shown in Table 1 (the mapping is unique). An $L1_{\text{shift}}$ is performed to the recoded multiplicand, obtaining the 2X multiple in BCD. Then, the 2X BCD multiple is recoded to (4221) using Expressions (4).

Multiple 4X: It is obtained as 2X × 2, where the 2X multiple is coded in (4221). The second ×2 operation is implemented as a digit recoding from (4221) to code (5211), followed by an $L1_{\text{shift}}$. The design of the (4221) to (5211) digit recoders is described in Section 4.4. The ×2 operation, with input operands coded in (4221) or (5211), is also implemented in the decimal CSA trees used for partial product reduction, and therefore, it is more detailed in Section 5.1.

Multiple 5X: It is obtained by a simple $L3_{\text{shift}}$ of the (4221) recoded multiplicand, with resultant digits coded in (5211). Then, a digit recoding from (5211) to (4221) is performed (see Section 4.4). Fig. 7 shows an example of this operation.

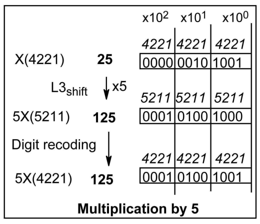


Figure 7: Calculation of ×5 for decimal operands coded in (4221).

Multiple 3X: It is evaluated by a carry-propagate addition of BCD multiples X and 2X in a *d*-digit BCD adder. The BCD sum digits are recoded to (4221) as indicated by previous expression. The latency of the partial product generation for the SD radix-10 scheme is constrained by the generation of 3X.The generation of (4221) decimal-coded multiples {-2X;-X; X; 2X} for the SD radix-5 recoding is shown in Fig. 8. The BCD multiplicand is first recoded to (4221) using Expressions (4). The 2X multiple is implemented as a digit recoding from (4221) to (5211) followed by an $L1_{\text{shift}}$. The negative multiples {-X;-2X}, coded in (4221), are obtained inverting the bits of the (4221) decimal-coded positive multiples and encoding the sign as described in Section 4.2.

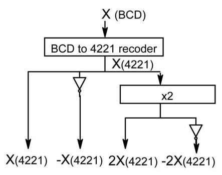


Figure 8: Generation of multiplicand multiples for SD radix-5.

4.4 Implementation of Digit Recoders

The design of efficient digit recoders is a critical issue, due to their high impact on the performance and area of the whole multiplier. Digit recoders are used to compute the decimal multiplicand multiples and in the reduction of partial products (Section 5) to compute $\times 2^n$ ($n > 0$) operations.

The logical implementation of digits recoders for BCD, BCD excess-6, and (5421) decimal codes is straightforward; since there is only a mapping of decimal digits to these codes (each decimal digit has a single 4-bit representation). However, due to the redundancy of (4221) and (5211) decimal codes, there are several choices for the digit recoding to (4221) or (5211). The sixteen 4-bit vectors of a coding can be mapped (recoded) into different subsets of 4-bit vectors of the other decimal coding representing the same decimal digit. These subsets of the (4221) and (5211) codes are also decimal codings.

Among all the subsets analyzed, the non-redundant decimal codes (4221s) and (5211s) (subsets of ten 4-bit vectors), shown in Table 2, present interesting properties. In particular, these codes verify

$$2Z(4221s) = L1_{shift}[Z(5211s)],$$

that is, after shifting 1 bit to the left an operand Z represented in (5211s), the resultant bit-vector represents the decimal value of 2Z coded in (4221s). This fact simplifies the implementation of $\times 2^n$ operations for $n > 1$. Specifically, for a decimal operand Z(4221), $Z \times 2^n$ is implemented by a first level of $Z_i(4221)$ to $Z_i(5211s)$ digit recoders followed by $n - 1$ levels of $Z_i(4221s)$ to $Z_i(5211s)$ digit recoders. The output of each level of digit

recoders is shifted 1 bit to the left such that the most significant bit of each (5211s) digit (weight 5) is shifted out to the next decimal position (weight 10).

Table 4: 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

Moreover, in some cases, the ×2 may be simplified. In particular, the recoding given by Expression (4) maps the BCD representation into the subset (4221s). Therefore, the subsequent ×2 operations in Figs. 5 and 7 are implemented using a level of simpler (4221s) to (5211s) digit recoders. A (4221) to (5211s) digit recoder has a hardware complexity of about 27 NAND2 gates, and its critical path has (roughly) the delay of a full adder. The (4221s) to (5211s) digit recoder has a simpler hardware complexity (about 19 NAND2 gates) with 25 percent less latency.

Additionally, the inverse digit recoding (from (5211) to (4221)) is easily implemented using a single full adder, since

$$Z_i(5211) = z_{i,3} \bullet 4 + z_{i,2} \bullet 2 + z^*_{i,1} \bullet 2 + z^*_{i,1} \quad ;$$

$$\text{with } z^*_{i,1} \bullet 2 + z^*_{i,0} = (z_{i,3} + z_{i,1} + z_{i,0}) \le 3.$$

This recoder is used to generate the ×5 multiple for the (4221) coding and in mixed (4221/5211) multioperand CSAs to convert a (5211) decimal-coded operand into the equivalent (4221) coded one.

CHAPTER 5
PARTIAL PRODUCT REDUCTION

First, the partial product arrays are generated by the SD radix-10 and SD radix-5 encodings. Each column of p digits is reduced to two digits by means of a decimal digit p:2 CSA tree. Also, decimal carries are passed between adjacent digit columns. Here, we present the set of preferred decimal codings and the method for decimal carry-save addition. We propose the use of the (4221) and (5211) decimal codings instead of BCD for an efficient implementation of decimal carry-save addition with binary CSAs or full adders. The use of these codes avoids the need for decimal corrections, so we only need to focus on the ×2 decimal multiplications. The implementation of decimal 3:2 CSAs for the proposed codings is also described in Section 5.2. To reduce the latency of the p:2 CSA trees, we make use of the decimal digit adders introduced in Section 5.3.These digit adders, implemented with bit counters, reduce up to 9 digits coded in (4221) or (5211) to 4 digits coded in (4221). Finally, we detail the design of the proposed p:2 decimal CSA trees implemented in the SD radix-10 (in Section 5.4) and SD radix-5 architecture (in Section 5.5). We present schemes optimized for area and for delay

5.1. Partial Product Arrays

The SD radix-10 architecture produces *d + 1* partial products coded in (4221) of *d + 3* digit length. Before being reduced, the *d + 1* partial products PP[i] are aligned according to their decimal weights by 4i-bit wired left shifts ($PP[i] \times 10^i$). The resultant partial product array for 16-digit input operands is shown in Fig. 9. In this case, the number of digits to be reduced varies from *p* = 17 to *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 5.4.

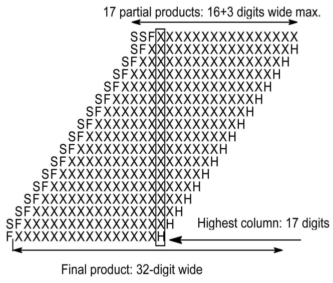

17 partial products: 16+3 digits wide max.
Highest column: 17 digits
Final product: 32-digit wide

Figure 9: Partial product arrays generated for 16-digit operands in the case of SD Radix-10 architecture

In this figure,

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

For the SD radix-5 architecture, the number of partial products generated is equal to *2d*, *d* of them coded in (5221) and the other *d* coded in (4221) (see Section 4.2). Both $PP[i]^U$ (5211) and $PP[i]^L$(4221) have the same weight 10^i.Thus, for 16-digit input operands, the alignment of the 32 partial products results in the digit array of Fig. 10. The p-digit columns of the SD radix-5 partial product array are reduced using the mixed (4221/5211) decimal *p:2* CSA trees presented in Section 5.5. The worst case for *d* = 16 corresponds to a column of *p* = 32 digits, reduced using a mixed (4221/5211) decimal 32:2 CSA.

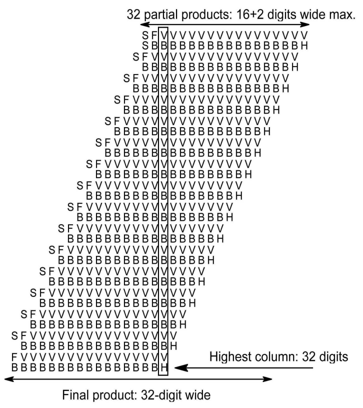

32 partial products: 16+2 digits wide max.
Highest column: 32 digits
Final product: 32-digit wide

Figure 10: Partial product arrays generated for 16-digit operands in the case of SD radix-5 architecture

In this figure,

S: Sign Encoding

H: Hot-One 10's complement encoding

V: Regular 4221 digit

B: Regular 5211 digit

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

5.2. Method for Decimal Carry-Save Addition

Decimal carry-save addition methods use a two BCD word to represent sum and carry or a BCD sum word and a carry bit per digit. The first group implements decimal addition mixing binary CSAs with combinational logic for decimal correction.

In another scheme two levels of 3:2 binary CSAs is used to add the partial products iteratively. Since it uses BCD to represent decimal digits, a digit addition of +6 or +12 (Modulo 16) is required to obtain the decimal carries and to correct the sum digit. In order to reduce the contribution of the decimal corrections to the critical path, three different techniques for multioperand decimal carry-save addition were proposed. Two of them perform BCD corrections (+6 digit additions) using combinational logic and an array of binary carry-save adders (speculative adders), although a final correction is also required. A sequential decimal multiplier using these techniques uses BCD invalid combinations (overloaded BCD representation) to simplify the sum digit logic. The other approach (non-speculative adder) uses a binary CSA tree followed by a single decimal correction. In the non-speculative adder, preliminary BCD sum digits are obtained using a level of 4-bit carry propagate adders after the binary CSA tree. Finally, decimal carry and sum digit corrections are determined from the preliminary sum digit and the carries passed to the next more significant digit position in the binary CSA tree. Decimal correction is performed using combinational logic (its complexity depends on the number of input operands added) and a 3-bit carry propagate adder per digit. Among these proposals, the non-speculative adders present the best area-delay figures and are suited for tree topologies.

The addition of all decimal operands in parallel requires the use of efficient multioperand decimal tree adders. Among the different schemes, the most promising ones for fast parallel addition seem to be those using binary CSA trees or some parallel network of full adders, due to their faster and simpler logic cells (full adders against SD adder cells or radix- 10 CLAs). These methods assume that decimal digits are coded in BCD. However, BCD is highly inefficient for implementing decimal carry-save addition by by means of binary arithmetic, because the need to correct the invalid 4-bit combinations (those not representing a decimal digit). Fig. 11 shows an example of the addition of 3 BCD digits using a 4-bit binary 3:2 CSA directly. In this case, the 4-bit representation (1100) of the decimal sum digit ('12') is an invalid BCD value and must be corrected to avoid overflows in subsequent BCD carry-save additions. The previous methods use different schemes to perform these BCD corrections. Moreover, the BCD carry digit must be multiplied by 2, which requires additional logic. We also implement

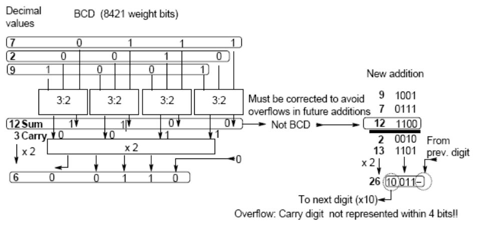


Figure 11: BCD carry-save addition using a 4-bit 3:2 CSA

multioperand decimal tree adders using a binary CSA tree, but with operands coded in decimal codings that are more efficient than BCD..

5.3. Alternative Decimal Digit Encodings

Among all the possible decimal codes defined by Expression (1) in Section 2 , there is a family of codes suitable for simple decimal carry-save addition. This family of decimal codings verifies that the sum of their weight bits is 9, that is,

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

which includes the (4221), (5211), (4311), and (3321) codes, shown in Table 1. Some of these decimal codings are already known , but we use them in a different context, to design components for decimal carry-save arithmetic. Moreover, they are redundant codes, since two or more different 4-bit vectors may represent the same decimal digit. These codes have the following two properties

- All the sixteen 4-bit vectors represent a decimal digit ($Z_i \in [0,9]$). Therefore, any Boolean function (AND, OR, XOR) operating over the 4-bit vector representation of two or more input digits produces a 4-bit vector that represents a valid decimal digit (input and output digits represented in the same code).

- The 9's complement of a digit Z_i can be obtained by inverting their bits (as a 1's complement) since

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

Negative operands can be obtained by inverting the bits of the positive bit vector representation and adding a 1 *ulp*, that is,

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

Next, we show how these codes can be used to improve multioperand decimal carry-save addition/subtraction using these two properties

5.4. Algorithm

Using the first property of these alternative decimal codings, we perform fast decimal carry-save addition using a conventional 4-bit binary 3:2 CSA as

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

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, and $H_i \in [0, 9]$ and $S_i \in [0, 9]$ are the decimal carry and sum digits at position *i*. No decimal correction is required because the 4-bit vector expressions of H_i and S_i represent valid decimal digits in the selected coding.

However, a decimal multiplication by 2 is required before using the carry digit H_i for later computations. Here, we restrict the analysis of decimal carry-save addition to only (5211) and (4221) decimal codes, since the generation of multiples of two for operands coded in (4311) and (3321) is more complex. Fig. 12 shows an example of ×2 multiplications for decimal operands represented in (4221) and (5211) decimal codes. To simplify the notation, we use H for the carry vector coded in (4221) and W for the carry vector coded in (5211). Thus, we have that

$$2H = 2\times H = L1_{shift}[W]$$

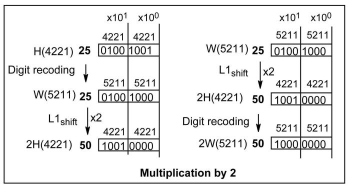


Figure 12: Calculation of ×2 for decimal operands coded in (4221) and (5211).

The resultant bit vector after shifting 1 bit to the left W represents 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 shift. The whole 2 × H multiplication is performed by a digit recoding of H_i into W_i followed by an $L1_{shift}[W]$. The bits of W_i are denoted by $w_{i,j}$. The bit shifted out ($w_{i,3}$) represents a decimal carry out (weight 10) to the next digit position, while the bit shifted in ($w_{i-1,3}$) is a decimal carry input (weight 1).

To subtract a decimal operand coded in (4221) or (5211) using a carry-save adder, we first invert the bits of the operand and add one *ulp* (unit in the last place). This ulp can be placed in the free room at the least significant bit position that results from the left shift of the carry operand *H*.

In the following Sections, we describe how to design decimal CSAs of any number of input operands coded in (4221) or (5211). We first detail the implementation of decimal 3:2 and 4:2 CSAs using the proposed method.

5.5. Decimal 3:2 and 4:2 CSAs

In this Section we detail the proposed implementations of a decimal 3:2 and 4:2 CSAs. We also describe the gate level implementation of the digit recoders required to perform conversions between different decimal codings. These recoders are the core logic components to compute $\times 2^n$ multiplications, which are also required for partial product generation in multiplication.

5.5.1. 4:2 Compressors

The 4:2 compressor structure actually compresses five partial products bits into three [1, 2, 3]. The architecture is connected in such a way that four of the inputs are coming from the same bit position of the weight j while one bit is fed from the neighboring position *j*-1(known as carry-in). The outputs of 4:2 compressor consists of one bit in the position *j* and two bits in the position *j*+1.This structure is called compressor since it compresses four partial products into two(while using one bit laterally connected between adjacent 4:2 compressors). Figure 13 shows the block diagram of 4-2 compressor. A 4-2 compressor can also be built using 3-2 compressors. It consists of two 3-2 compressors (full adders) in series and involves a critical path of 4 XOR delays as shown in Figure 14. An alternative implementation is shown in Figure 15. This implementation is better and involves a critical path delay of three XOR's , hence reducing the critical path delay by 1 XOR. The output C_{out}, being independent of the input C_{in} accelerates the carry save summation of the partial products

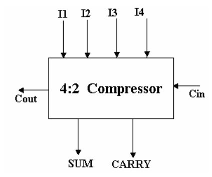


Figure13: Block diagram of a 4:2 compressor

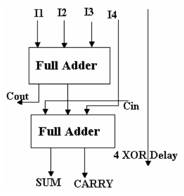


Figure14: Compressor design with full adder

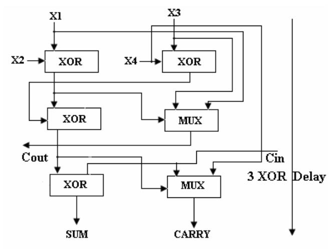
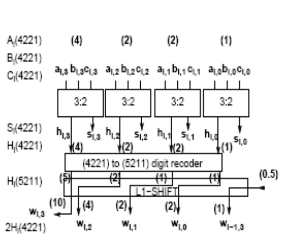


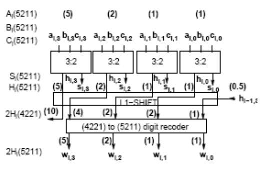
Figure15: Alternative Implementation of 4:2 Compressor with 3 XOR Delay

5.5.2. Gate level implementation

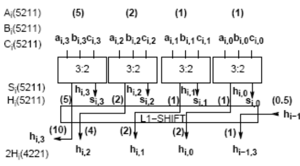
The proposed decimal 3:2 CSAs adds three decimal operands (A,B,C) coded in (4221) or (5211) and produce a decimal sum word (S) and a carry word (H) multiplied by 2 (2 × H) coded in (4221) or (5211), such that A + B + C = S + 2H. Depending on the decimal coding of the operands, we have three possible implementations of a decimal digit 3:2 CSA using a 4-bit binary 3:2 CSA, as shown in Fig. 16



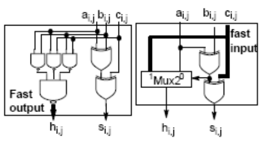
(a)Operands coded in (4221)



(b)Operands coded in (5211)



(c)Mixed (5211/4221) coded output operands.



(d)Full adder with fast carry output

(e) Full adder with fast input

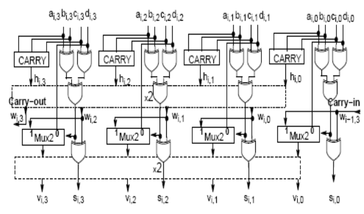
Figure 16(a)-(e): Proposed decimal digit (4-bit) 3:2 CSAs.

- Input operands and output operands (*S*, *H*, 2*H*) coded in (4221) (Fig. 16(a)). The weight bits in Fig. 16 are placed in brackets above each bit column. In this case, the decimal digit 3:2 CSA consists of a 4-bit binary 3:2 CSA and a digit recoder from (4221) to (5211).In this section we show two gate level implementations of a 1-bit 3:2 CSA: one with a fast carry output (Fig. 16(d)) and one with a fast input (Fig. 16(e)). The output of the digit recoder ($H_i(5211)$) is then left shifted by one bit position ($L1_{shift}$[H_i(5211)]). The recoder is placed in the carry path, so choosing an appropriate gate implementation of the binary 3:2 CSA, in this case the fast carry output configuration (Fig. 16(d)), part of the recoder delay can be hidden.

- Input and output operands coded in (5211) (Fig. 16(b)). The implementation of the (5211) decimal digit 3:2 CSA is similar to the (4221) case, except that here the 4-bit carry vector H_i(5211) is 1-bit left shifted before the digit recoding.
- Input operands coded in (5211), *S*, *H* coded in (5211) but 2*H* coded in (4221) (Fig. 16(c)).The decimal digit 3:2 CSA consists only of a level of 4-bit 3:2 CSA with the carry output shifted 1-bit to the left.

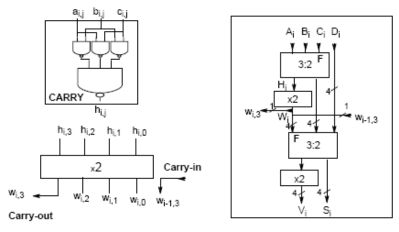
The gate level implementation of two decimal 4:2 CSAs for input and output operands coded in (4221) is shown in Fig. 17. The first decimal 4:2 CSA (Fig. 17(a)) uses a specialized gate configuration. The carry bit-vector *H* is computed as in binary from operands *A*, *B* and *C* coded in (4221).The intermediate decimal carry operand *W* is then obtained as 2 × *H*. The sum operand *S* (coded in (4221)) is obtained by XOR-ing the bits of *A*, *B*, *C*, *D* and *W* (approximately in 4 XOR gate delays). The decimal carry operand *V* is obtained (approximately in 6 XOR gate delays) by selecting the appropriate bits of *D* or *W*, depending on the xor of *A*, *B*, *C* and *D*, and multiplying the resulting bit vector (coded in (4221)) by 2.

The second decimal 4:2 CSA (Fig. 17(b)) is designed by interconnecting two decimal 3:2 CSAs (Fig. 17(a)). The blocks labeled as 3:2 represent a 4-bit binary 3:2 CSA. The intermediate decimal carry *W* is connected to a fast input of the second full adder (indicated by a letter F in Fig. 17(b)) to reduce the delay of the critical path. Thus, both implementations present a similar critical path delay (6 XOR gate delays in the carry path).



$a_{i,3}b_{i,3}c_{i,3}d_{i,3}$
$a_{i,2}b_{i,2}c_{i,2}d_{i,2}$
$a_{i,1}b_{i,1}c_{i,1}d_{i,1}$
$a_{i,0}b_{i,0}c_{i,0}d_{i,0}$
CARRY
$h_{i,3}$
$h_{i,2}$
$h_{i,1}$
$h_{i,0}$
Carry-out
Carry-in
x2
Mux2
$w_{i,3}$
$w_{i,2}$
$w_{i,1}$
$w_{i,0}$
$w_{i-1,3}$
$v_{i,3}$ $s_{i,3}$ $v_{i,2}$ $s_{i,2}$ $v_{i,1}$ $s_{i,1}$ $v_{i,0}$ $s_{i,0}$

(a) Using a specialized gate configuration



$a_{i,j}$ $b_{i,j}$ $c_{i,j}$
CARRY
$h_{i,j}$
$h_{i,3}$ $h_{i,2}$ $h_{i,1}$ $h_{i,0}$
x2
Carry-in
$w_{i,3}$ $w_{i,2}$ $w_{i,1}$ $w_{i,0}$ $w_{i-1,3}$
Carry-out
A_i B_i C_i D_i
3:2
F
H_i
x2
W_i
$w_{i,3}$
$w_{i-1,3}$
V_i S_i

(b) Using two decimal 3:2 trees

Figure 17: Proposed decimal (1-digit slice) 4:2 CSAs.

5.5.3. Implementation of digit recoders

The design of efficient digit recoders is a critical issue, due to their high impact on the performance and area of a decimal multiplier. Due to the redundancy of (4221) and (5211) decimal codes, there are many choices for the digit recoding between (4221) and (5211). The sixteen 4-bit vectors of a coding can be mapped (recoded) into different subsets of 4-bit vectors of the other decimal coding representing the same decimal digit. These subsets of the (4221) and (5211) codes are also decimal codings. Among all the subsets analyzed, we have selected the non-redundant decimal codes (subsets of ten 4-bit vectors) shown in Table 2 to represent the recoded digits. These codes lead to two

different configurations of digit recoders (*S*1 and *S*2) for the recoding from (4221) to (5211):

- The first group of codes, *S*1 = {(4221-S1), (5211-S1)} leads to a simpler implementation of a digit recoder when all the sixteen 4-bit input combinations are possible. Therefore, in general, a ×2 block is implemented by digit recoding *Z*(4221) into Z(5211-S1) and shifting the output one bit to the left. The gate level implementation of a S1 digit recoder is shown in Fig. 18. This operation can be seen as a two-step digit recoding of Z_i(4221) to Z_i(4221-*S*1) and Z_i(4221-*S*1) into Z_i(5211-*S*1). This operation can be seen as a two-step digit recoding of Z_i(4221) to Z_i(4221-*S*1) and Z_i(4221-*S*1) into Z_i(5211-*S*1). The digit recoding between Z_i(4221-*S*1) and Z_i(5211-*S*1) is very simple, since the 4-bit vectors representing each decimal digit value in both decimal codes are almost similar.

Z_i	Z_i(4221-S1)	Z_i(5211-S1)	Z_i(4221-S1)	Z_i(5211-S1)
0	0000	0000	0000	0000
1	0001	0001	0001	0001
2	0100	0100	0010	0100
3	0101	0101	0011	0101
4	0110	0111	1000	0111
5	1001	1000	1001	1000
6	1010	1010	1010	1001
7	1011	1011	1011	1100
8	1110	1110	1110	1101
9	1111	1111	1111	1111

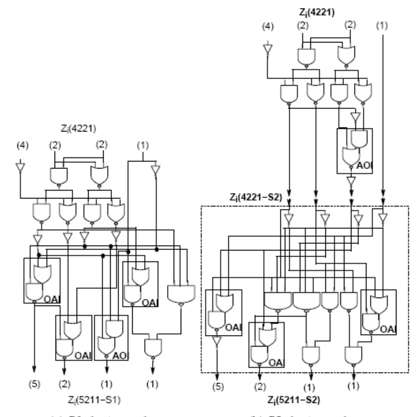
Table 4. Selected decimal codes for the recoded digits

- The second group of codes, S2 = {(4221-S2), (5211- S2)} verifies

$$2Z(4221\text{- S2}) = L1_{shift}[Z(5211\text{- S2})]$$

that is, after shifting one bit to the left an operand represented in (5211-S2), the resultant digits are represented in (4221- S2). This fact simplifies the implementation of $\times 2^n$ operations with |n| > 1. Specifically, $2^n \times Z$ can be

implemented recoding each digit Z_i(4221) to Z_i(4221-S2) followed by n stages of Z_i(4221-S2) to Z_i(5211-S2) digit recoders. The implementation of this S2 digit recoder is shown in Fig. 18(b) (the Z_i(4211- S2) to Z_i(5211- S2) recoder is shown inside the dashed line box). Moreover, when input digits into a 4-bit binary 3:2 CSA are coded in a S2 decimal coding then the resultant carry digit H_i is represented in the same S2 coding. In this case, 2×H is implemented as a row of the simpler H_i(4211-S2) to H_i(5211- S2) digit recoders with outputs or inputs 1-bit left shifted.



(a) S1 digit recoder. (b) S2 digit recoder.

Figure 18: Gate level implementation of the (4221) to (5211) digit recoders.

Additionally, the inverse digit recoding (from (5211) to (4221)) is easilyimplemented using a single full adder as shown in Fig. 19, since

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

with $z^*_{i,1} \bullet 2 + z^*_{i,0} = (z_{i,3}+z_{i,1}+z_{i,0}) \leq 3$. This recoder is used in mixed (4221/5211) multioperand CSAs to convert a (5211) decimal coded operand into the equivalent (4221) coded one.

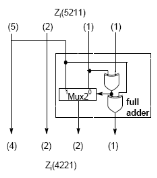


Figure 19: Implementation of a (5211) to (4221) digit recoder.

CHAPTER 6
PROPOSED ARCHITECTURE - COMBINED DECIMAL/BINARY CARRY SAVE ADDER

In this implementation the whole binary carry-save adder tree is shared for both binary and decimal operations. The latency of the binary operation is unaffected by the incorporation of hardware support for decimal and there is reduction in the power consumption also apart from the additional area required. This additional hardware computes a decimal correction amount which is added to the binary sum to produce the correct decimal result. Since the most part of this computation is overlapped with the binary carry-save addition, the maximum overhead delay of decimal multioperand addition is bounded approximately by 10 XOR gate delays. In this design decimal correction is completely separated from the binary carry-save adder, so that decimal hardware can be easily turned off to reduce power consumption in binary operation mode. Furthermore, it has a very regular and simple structure, which facilitates the integration of the proposed method into a CAD tool for automatic synthesis.

6.1 ALGORITHM

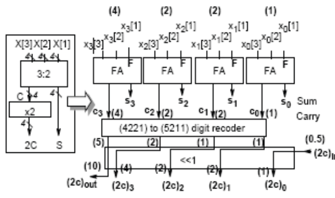


Figure20: Decimal 3:2 carry-save adder (1 digit)

The block diagram of a 1-digit (4-bit) decimal 3:2 carry-save adder is detailed in Fig. 20. The blocks labeled as FA, 3:2, and ×2 are respectively full adders with a fast

input (of 1 XOR delay, indicated with an F), 4-bit binary 3:2 carry save adders and decimal digit doubling units. A fixed left shift of one bit is denoted by <<1.

A binary/decimal 3:2 carry-save-adder is build in by a straightforward modification of the digit doubling unit of Fig. 20: a 4-bit 2:1 multiplexer is placed after each digit recoder and selects either the carry output of the 3:2 carry-save adder for binary mode or the output of the digit recoder for decimal mode. The output of the multiplexer is shifted one bit to the left. A combined multi-operand adder is implemented as a tree of these modified carry-save adders. Fig. 21 shows an example for 12 operands. The binary/decimal doubling units require an additional input signal dec to indicate the operation mode (dec=1 for the decimal mode).

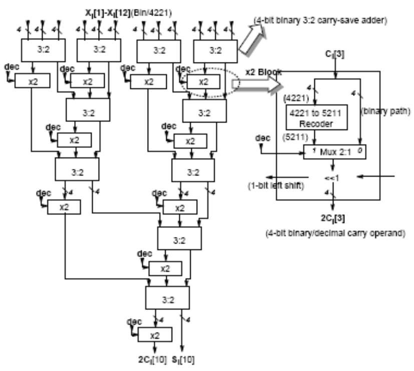


Figure 21: Combined multi-operand adder tree: (1 digit/4-bit column)

For further area savings a more interesting alternative would be to fully reuse the binary carry-save adder for both binary and decimal multi-operand additions. In this Section we present a method to implement decimal multi-operand addition using any binary carry-save adder (such as a binary 4:2 compressor tree) and separate hardware for decimal correction. The key idea is to perform the decimal doubling *C* × 2 of the carry

operand *C* coded in 4221 2 as a 1-bit left shift (*C* <<1), that is, as a binary doubling. This would allow us to compute both binary and decimal carry-save additions in the same fashion, just by left shifting the carry output of the binary 3:2 carry-save adder.

However, since a left shift of a 4221 decimal coded operand *C* does not produce exactly its double 2*C*, we have to estimate a correction amount to be added to the binary result in order to get the correct decimal sum. Thus, a left shift of a p-digit decimal operand *C* coded in 4221 produces that each digit C_i is modified as

$$(C_i << 1) = c_{i,3} \bullet 10 + c_{i,2} \bullet 4 + c_{i,1} \bullet 2 + c_{i,0} \bullet 2$$

On the other hand, the double of a 4221 decimal coded digit is given by

$$C_i \times 2 = c_{i,3}\,(10 - 2) + c_{i,2} \bullet 4 + c_{i,1}\,(2 + 2) + c_{i,0} \bullet 2$$

The operand 2*C* = *C* × 2 (represented in code 4221) and the 1-bit shifted operand (*C* <<1) are then related by:

$$2C = (C <<1) + 2 \times \sum_{i=0}^{p-1} (C_{i,1} - C_{i,3}) \times 10^i$$

Therefore, we have to increment the decimal correction amount *W* into +2 units at digit W_i if the carry bit $c_{i,1}$ is one or decrement it by -2 if the carry bit $c_{i,3}$ is one.
For multi-operand addition, each intermediate carry operand *C*[k] of the binary carry-save adder contributes to the decimal correction amount, but not for the final carry operand, which is multiplied by 2 for decimal. A functional scheme of the proposed method for 4p-bit binary/p-digit decimal 4221 coded operands is shown in Fig. 15. For simplicity, we consider in Fig. 15 that the m input operands X[k] are aligned to the decimal point and that the sum does not overflow. For m operands, the number of intermediate carry operands *C*[k] generated in a binary *m* : 2 carry-save adder is *m* − 3.

The decimal correction amount *W* is computed in parallel with the binary carry-save addition using an array of bit counters and a decimal carry-save addition. We separate the positive ($c[k]_{i,1}$) and the negative ($c[k]_{i,3}$) carry bits, as soon as they are generated in the binary carry-save addition, in groups of 9 bits at most for each decimal position *i*. These groups of bits are added as

$$W_i|2l\text{-}1| = \sum_{k=9l-8}^{9l+1} c[k]_{i,3} \qquad , \qquad W_i|2l| = \sum_{k=9l-8}^{9l+1} c[k]_{i,1}$$

using 2q rows of 9-bit counters (or simpler counters) with output coded in 4221, where q = [(m − 3)/9] and *l* goes from 1 to q. The 4-bit sum value of a 9-bit counter represents a decimal digit $W_i[l] \in [0, 9]$ coded in 4221, so that the output of each row of counters is a decimal operand W[2*l*-1] or W[2*l*] of p digits coded in 4221. The decimal correction amount W is given by

$$W = 2\times \sum_{i=1}^{q} (W[2l] - W[2l-1])$$

Since the representation 4221 is self-complementing the negative operands $-W[2l - 1]$ are obtained by a bit inversion of $W[2l - 1]$ and a subsequent addition of a unit in the least significant place, i.e. as if they are two's complement operands. To obtain the final sum

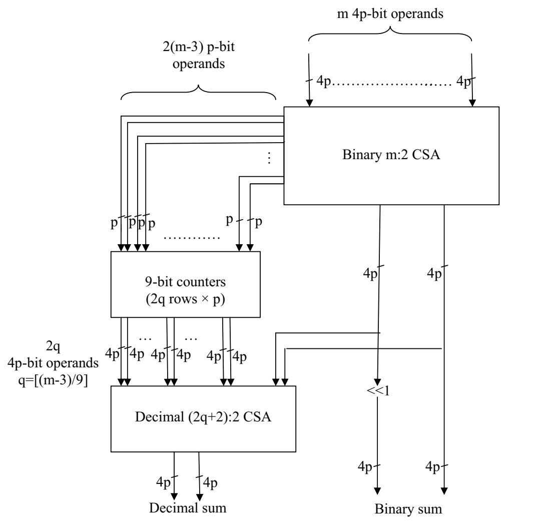


Figure22: Block diagram of the combined binary/decimal multi-operand addition method.

$S[2q+m-2]$ and carry $C[2q+m-2] \times 2$ operands, we add the 2*q* operands $W[2l] \times 2$ and $-W[2l - 1] \times 2$ to the result of the binary carry-save addition $S[m-2]$, $C[m-2] \times 2$. Since all operands are in 4221 code, we use binary 3:2 carry-save adders and decimal doubling units to perform a decimal (2*q* + 2) : 2 carry-save addition.

6.2 Implementation of combined decimal/binary carry save adder

Here we consider some pre-existing binary carry-save adder such a optimized tree of 3:2 or 4:2 compressors and reuse it to also support decimal multi-operand addition. In Fig. 23 we show a block diagram of 1-digit column (4-bit slice) of the proposed implementation for m = 12 input operands. The binary adder tree consists of two levels of 4:2 compressors and one level of 3:2 compressors (binary 3:2 carry-save adder). The 4:2 compressors are build of two levels of 3:2 compressors optimally interconnected so that the critical path only goes through 3 XOR gates. A total of 9 intermediate carry operands are generated by the compressor tree. The carry bits $c[k]_{i,3}$ and $c[k]_{i,1}$ are summed separately by two 9-bit counters, resulting 4221 coded digits $W_i[1]$ and $W_i[2]$. The internal structure of the 9-bit counter is detailed in the upper left corner of Fig. 19. It is build of 5 full adders arranged in two levels which calculates the sum of the input bits in code 4221. The fastest input goes through 2 XOR levels, while the slowest signal goes through 4 XOR levels. To obtain the negative sum in 4221 code, the counter outputs are inverted. Since carries arrive to the counters with different delays, those produced in the last levels of the binary adder tree are connected to faster inputs of the counter in order to balance the different path delays.

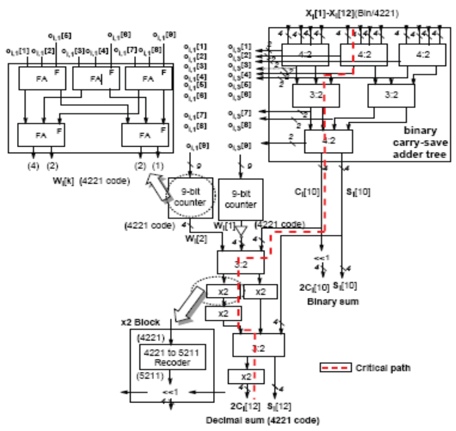


Figure 23: Multi-operand adder tree

The decimal digits $\overline{W_i[1]}$ and $W_i[2]$, and the sum Si[10] and carry Ci[10] ouputs of the compressor tree are dispatched to a decimal 4:2 carry-save adder. To reduce the number of doubling units needed, the calculation

$$(W_i[1] + W_i[2] + C_i[10]) \times 2 = S_i[11] \times 2 + C_i[11] \times 2 \times 2$$

is performed first using a binary 3:2 carry-save adder and three doubling units. The two cascaded doubling units can be merged into a ×4 unit to obtain a small reduction in area and delay. The critical path of the decimal operation is indicated in Fig. 4 by a thick dotted line. It goes through 15 levels of XOR gates, eight of them corresponding to the binary adder tree.

CHAPTER 7
HIGH-PERFORMANCE RADIX-4 MULTIPLIER USING THE PASS TRANSISTOR LOGIC

A high-performance adder has been designed with modified complementary pass transistor logic technique. The adder has been implemented on a radix-4 multiplier.

7.1 Adder Architecture Using The CPL Technique

The CPL technique eliminates the occurrence of P-type Metal Oxide Semiconductor (PMOS) latch, and techniques capable of overcoming the pass transistor logic threshold voltage loss problem do so by adding an inverter at the output. The logic style of CPL results in a smaller number of transistors and smaller input loads, especially when N-type Metal Oxide Semiconductor (NMOS) networks are used. However, the CPL circuit has some drawbacks due to body effects, source follower action, and high power leakage. When it is not cross-coupled, it will cause low performance at large stage counts and limited fan-out capability. According to Markovic *et al.*,the duality principle of the proposed CPL adder circuit topology, with inverted gate signals, gives the dual logic function. Dual logic functions include AND-OR, NAND-NOR and XOR-XNOR. Referring to the basic structure of pass transistor logic style, by simply modifying the input nodes, AND, OR, NAND and NOR logic gates can be constructed. By changing the input nodes at the source terminal, XOR and NXOR logic gates can be constructed.

The full adder cell is designed with the CPL technique and the multiplexing control input technique (MCIT) for both sum and carry operations. The sum and carry operation is designed based on the given equation, where two XOR logic gates are used, since pass-transistor logic is advantageous in constructing XOR logic gates. By combining the sum and carry circuits, the XOR gate in the carry operation can be omitted, and both circuits can share the common term, $A\oplus B$, in the sum operation.

$$\text{Sum} = A\oplus B\oplus C$$

$$C_{out} = (A\oplus B)\, C_{in} + AB$$

The inputs A, A's complement (A'), B, and B's complement (B') are fed as inputs to the pass transistors and form an XOR logic gate. These four inputs construct an XOR logic operation at the transistor level, which is designed using two transistors. In order to reduce the number of transistors, the output of the XOR gate (A⊕B) is fed through an NOT gate from the differential node to the pass transistors as a control input. On the other hand, C_{in} is treated as variable input, which is fed through the pass transistor source terminal. At this stage, the functionality of the circuit is equivalent to the sum operation, sum A⊕B⊕C, and six transistors have been used. As mentioned before, the number of transistors in the carry operation can be reduced by taking A⊕B as the input from the sum operation circuit AND with C_{in} in order to produce the operation equivalent to (A⊕B)C_{in} , which only uses another two transistors. Meanwhile, the inputs A, A', B, and B' are fed into pass transistors in order to produce an AND logic gate, which represents the AB operation. The outputs of both (A⊕B) C_{in} and AB are used as multiplexing inputs in order to sum both terms with the OR gate operation. The transistor count can be reduced by modifying the OR gate at the last stage of the carry equation. This is done by removing the inverter and the transistor fed by the inverter. Markovic's full adder circuit has 22 transistors. At an earlier point, 3 transistors were omitted in this design and the number of transistors of the full adder cell was reduced to 17 transistors, which is lower than the number of transistors in the circuit described by Markovic , which is 22.

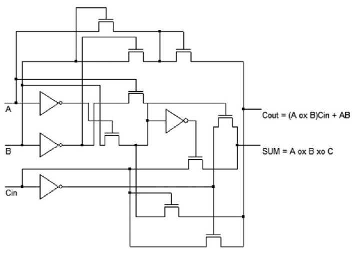


Figure 24: Full adder circuit

Fig.24 shows the proposed full adder circuit using 17 transistors after applying the redundant transistor reduction technique. The basic architectures of the 16 × 16 bit basic CSA multiplier were constructed based on the architectures given by Yeo *et al*. The full adder blocks presented were placed with our proposed full adder cell, and all the logic gates in both multiplier architectures were designed based on the CPL technique in order to compare their performance under identical conditions.

7.2 Architecture of Radix-4 Multiplier

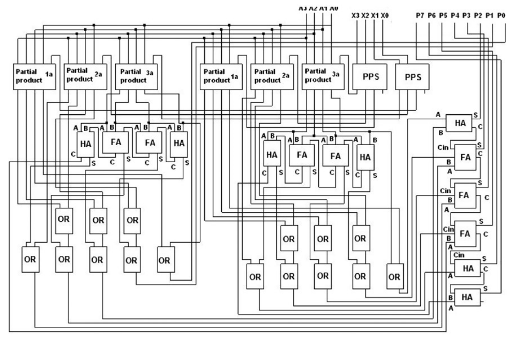


Figure 25: 4 × 4 bit radix-4 multiplier circuit

PPS : Partial Product Selector

HA : Half Adder

FA : Full Adder

OR : OR Logic gate

The architecture of our proposed radix-4 multiplier circuits comprises partial product selectors, partial product pre-computation blocks, and half adder and full adder block, which is shown in Fig.25. In the radix-4 circuits, 2 bits per cycle will be considered. Therefore, 4 multiples, 0*a*, 1*a*, 2*a* and 3*a*, are pre-computed, where "*a*" is the

multiplicand. This is done by the partial product pre-computed blocks, where 2*a* is simply the shifted version of "*a*", and 3*a* = 2*a* + 1*a* . The pre-computation circuit for 3*a* consists of half adder and full adder blocks configured using the ripple carry adder (RCA) architecture. The half adder circuit is designed based on the CPL technique, and the full adder blocks are used with our proposed full adder circuit. Partial product selectors are formed by OR and AND gates, which are used to determine the partial products. By connecting all the pre-computation blocks and partial product selectors, a 4-to-1 multiplexer can be realized, as shown in Fig.26. The multiplexer is functioned such that the first 2 bits of the multiplier, *x*, will be grabbed to determine the first partial product and shifted to the next 2 bits of the multiplier to determine the successive partial products by repeating the same process. For a 4-bit radix-4 multiplier, two partial products will be generated. As a result, half of the number of partial products has been reduced compared to the normal 1-bit shift-add algorithm.

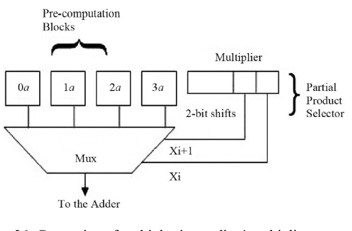


Figure 26: Generation of multiples in a radix-4 multiplier

Before adding the partial product, all pre-computed partial products are OR-ed with each other, since

Partial Product = 0a+1a+2a+3a

At the end, all partial products with proper shifts are connected to RCAs to compute the final output product of the radix-4 multiplier. The multiplicand, "*a*", and multiplier, "*x*", are two inputs that are calculated in parallel by the multiplier circuit. A 4-bit binary

number can be interpreted as a 2-digit radix-4 number, and radix-4 multiplication can be represented as

$$p^{(j+1)}=[p^{(j)}+(x_{j+1}x_j)_{two}ar^k]4^{-1}$$

where *p* = product, *a* = multiplicand and x = multiplier. Based on the multiplication recurrences above, a more practical example of radix multiplication is shown below. Without considering whether the 3*a* multiple will be needed during the multiplication, the 3*a* multiple is always computed at the outset and stored in a register for future use.

a	1110
3a	101010
x	1011
p(0)	000
+(*x*1*x*0) *two a*	101010
4*p*(1)	101010
p(1)	1010 10
+(*x*3*x*2) *two a*	011100
4*p*(2)	100110 10
p(2)	1001 1010

CHAPTER 8
EVALUATION RESULTS

The simulation of this project has been done using MODELSIM XE111 6.2g and XILINX ISE 9.1i.

Modelsim is a simulation tool for programming {VLSI} {ASIC}s, {FPGA}s, {CPLD}s, and {SoC}s. Modelsim provides a comprehensive simulation and debug environment for complex ASIC and FPGA designs. Support is provided for multiple languages including Verilog, SystemVerilog, VHDL and SystemC. The Modelsim conceptual overview is shown below.

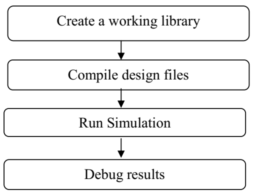


Figure 27: Conceptual Overview of Modelsim

In ModelSim, all designs, be they VHDL, Verilog, or some combination thereof, are compiled into a library. We can stat a new simulation in ModelSim by creating a working library called "work". "Work" is the library name used by the compiler as the default destination for compiled design units. After creating the working library, we compile our design units into it. The ModelSim library format is compatible across all supported platforms. We can simulate our design on any platform without having to recompile your design. With the design compiled, invoke the simulator on a top-level module (Verilog) or a configuration or entity/architecture pair (VHDL). Assuming the design loads successfully, the simulation time is set to zero, and enter a run command to begin simulation. If the results are not as expected, use ModelSim's robust debugging environment to track down the cause of the problem.

Xilinx, Inc. is an American technology company, which designs, develops and markets programmable logic products including integrated circuits (ICs), software design

tools, predefined system functions delivered as intellectual property (IP) cores, design services, customer training, field engineering and technical support. Xilinx sells both FPGAs and CPLDs programmable logic devices for electronic equipment manufacturers in end markets such as communications, industrial, consumer, automotive and data processing The Virtex-II Pro, Virtex-4, Virtex-5, and Virtex-6 FPGA families are particularly focused on system-on-chip (SOC) designers because they include up to two embedded IBM PowerPC cores. Xilinx has offered two main FPGA families: the high-performance Virtex series and the high-volume Spartan series, with a cheaper EasyPath option for ramping to volume production. With the introduction of its 28 nm FPGAs in June 2010, Xilinx replaced the high-volume Spartan family with a Kintex family and the low-cost Artix family. The Spartan series targets applications with a low-power footprint, extreme cost sensitivity and high-volume; e.g. displays, set-top boxes, wireless routers and other applications

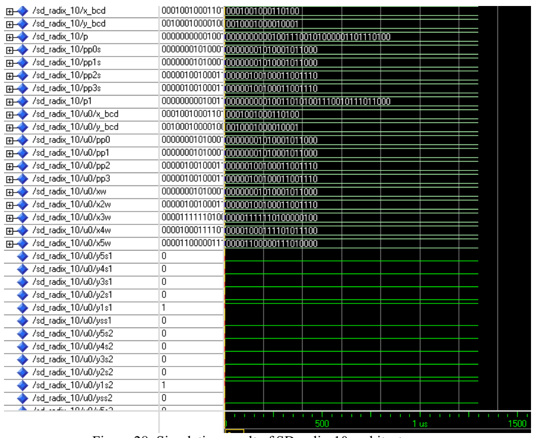
The ISE Design Suite is the central electronic design automation (EDA) product family sold by Xilinx. The ISE Design Suite features include design entry and synthesis supporting Verilog or VHDL, place-and-route (PAR), completed verification and debug using Chip Scope Pro tools, and creation of the bit files that are used to configure the chip.

Xilinx is a synthesis tool which converts Schematic/HDL design entry into functionally equivalent logic gates on Xilinx FPGA, with optimized speed & area. So, after specifying behavioral description for HDL, the designer merely has to select the library and specify optimization criteria; and Xilinx synthesis tool determines the net list to meet the specification; which is then converted into bit-file to be loaded onto FPGA-PROM. Also, Xilinx tool generates post-process simulation model after every implementation step, which is used to functionally verify generated net list after processes, like map, place & route

The synthesis and the simulation results of the proposed and the existing architecture are shown below.

8.1. SIMULATION RESULTS

8.1.1.SD Radix-10 Architecture



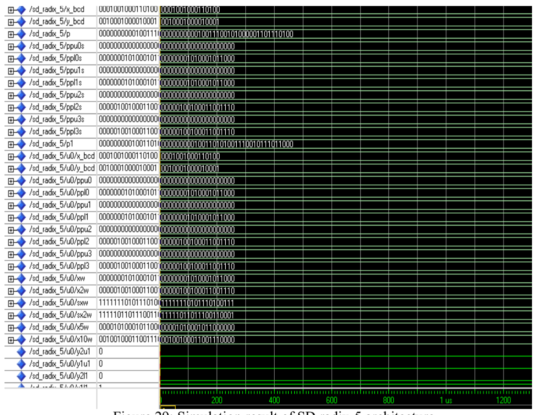
```
/sd_radix_10/x_bcd       0001001000110   0001001000110100
/sd_radix_10/y_bcd       0010001000010   0010001000010001
/sd_radix_10/p           000000000010    0000000000100111001010000011011110100
/sd_radix_10/pp0s        0000000101000   00000001010001011000
/sd_radix_10/pp1s        0000000101000   00000001010001011000
/sd_radix_10/pp2s        0000010010001   00000100100011001110
/sd_radix_10/pp3s        0000010010001   00000100100011001110
/sd_radix_10/p1          000000000100    0000000001001101010011100101110110000
/sd_radix_10/u0/x_bcd    0001001000110   0001001000110100
/sd_radix_10/u0/y_bcd    0010001000010   0010001000010001
/sd_radix_10/u0/pp0      0000000101000   00000001010001011000
/sd_radix_10/u0/pp1      0000000101000   00000001010001011000
/sd_radix_10/u0/pp2      0000010010001   00000100100011001110
/sd_radix_10/u0/pp3      0000010010001   00000100100011001110
/sd_radix_10/u0/xw       0000000101000   00000001010001011000
/sd_radix_10/u0/x2w      0000010010001   00000100100011001110
/sd_radix_10/u0/x3w      0000111111010   00001111110100000100
/sd_radix_10/u0/x4w      0000100011110   00001000111101011100
/sd_radix_10/u0/x5w      0000110000011   00001100000111010000
/sd_radix_10/u0/y5s1     0
/sd_radix_10/u0/y4s1     0
/sd_radix_10/u0/y3s1     0
/sd_radix_10/u0/y2s1     0
/sd_radix_10/u0/y1s1     1
/sd_radix_10/u0/yss1     0
/sd_radix_10/u0/y5s2     0
/sd_radix_10/u0/y4s2     0
/sd_radix_10/u0/y3s2     0
/sd_radix_10/u0/y2s2     0
/sd_radix_10/u0/y1s2     1
/sd_radix_10/u0/yss2     0
```

Figure 28: Simulation result of SD radix-10 architecture

In the above figure x and y are the multiplicand and the multiplier respectively (in BCD). The final product in BCD is p.pp0,pp1,pp2,pp3 shows the partial products. All the other variables are intermediate results.

Here x in decimal	=	1234
x in BCD	=	0001001000110100
y in decimal	=	2211
y in BCD	=	0010001000010001
p in decimal	=	2728374
p in BCD	=	0010011100101000001101110100

8.1.2. SD Radix-5 Architecture



```
/sd_radix_5/x_bcd        0001001000110100    0001001000110100
/sd_radix_5/y_bcd        0010001000010001    0010001000010001
/sd_radix_5/p            000000000100111     0000000000100111001010000011011110100
/sd_radix_5/ppu0s        000000000000000     00000000000000000000
/sd_radix_5/ppl0s        000000010100010     00000001010001011000
/sd_radix_5/ppu1s        000000000000000     00000000000000000000
/sd_radix_5/ppl1s        000000010100010     00000001010001011000
/sd_radix_5/ppu2s        000000000000000     00000000000000000000
/sd_radix_5/ppl2s        000001001000110     00000100100011001110
/sd_radix_5/ppu3s        000000000000000     00000000000000000000
/sd_radix_5/ppl3s        000001001000110     00000100100011001110
/sd_radix_5/p1           000000000100110     0000000001001101010011100101110110000
/sd_radix_5/u0/x_bcd     0001001000110100    0001001000110100
/sd_radix_5/u0/y_bcd     0010001000010001    0010001000010001
/sd_radix_5/u0/ppu0      000000000000000     00000000000000000000
/sd_radix_5/u0/ppl0      000000010100010     00000001010001011000
/sd_radix_5/u0/ppu1      000000000000000     00000000000000000000
/sd_radix_5/u0/ppl1      000000010100010     00000001010001011000
/sd_radix_5/u0/ppu2      000000000000000     00000000000000000000
/sd_radix_5/u0/ppl2      000001001000110     00000100100011001110
/sd_radix_5/u0/ppu3      000000000000000     00000000000000000000
/sd_radix_5/u0/ppl3      000001001000110     00000100100011001110
/sd_radix_5/u0/xw        000000010100010     00000001010001011000
/sd_radix_5/u0/x2w       000001001000110     00000100100011001110
/sd_radix_5/u0/sxw       111111101011101     11111110101110100111
/sd_radix_5/u0/sx2w      111110110111001     11111011011100110001
/sd_radix_5/u0/x5w       000010100010110     00001010001011000000
/sd_radix_5/u0/x10w      001001000110011     00100100011001110000
/sd_radix_5/u0/y2u1      0
/sd_radix_5/u0/y1u1      0
/sd_radix_5/u0/y2l1      0
```

Figure 29: Simulation result of SD radix-5 architecture

In the above figure x and y are the multiplicand and the multiplier respectively (in BCD). The final product in BCD is p. ppu and ppl shows the partial products corresponding to 4221 and 5211 recoding. All the other variables are intermediate results.

Here x in decimal	=	1234
x in BCD	=	0001001000110100
y in decimal	=	2211
y in BCD	=	0010001000010001
p in decimal	=	2728374
p in BCD	=	0010011100101000001101110100

.

8.1.3. SD Radix-10 Architecture using Combined Decimal/Binary CSA

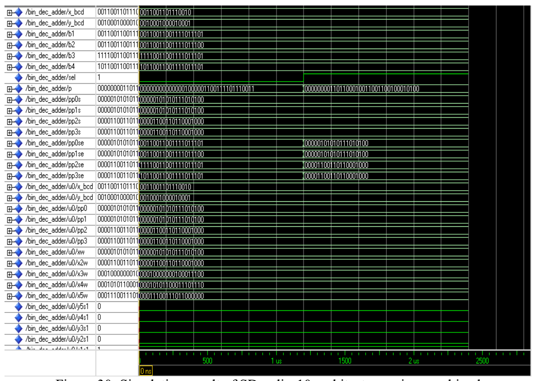


Figure 30: Simulation result of SD radix-10 architecture using combined decimal/binary CSA

In the above figure x and y are the multiplicand and the multiplier respectively (in BCD). The final product in BCD is p. pp0,pp1,pp2,pp3 shows the partial products as per the SD radix-10 recoding scheme and b1,b2,b3,b4 corresponds to binary partial products. All the other variables are intermediate results. 'sel' corresponds to selection signal for mux (decimal/binary)

Here x in decimal = 1234
x in BCD = 0001001000110100
y in decimal = 2211
y in BCD = 0010001000010001
p in decimal = 2728374
p in 4221 = 0100110101001110010111011000
p in binary = 1010011010000110110110

8.1.4. SD Radix-5 Architecture using Combined Binary-Decimal CSA

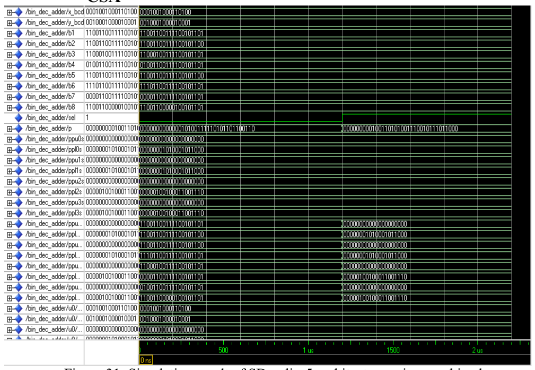


Figure 31: Simulation result of SD radix-5 architecture using combined decimal/binary CSA

In the above figure x and y are the multiplicand and the multiplier respectively (in BCD). The final product in BCD is p. ppu and ppl shows the partial products as per the SD radix-5 recoding scheme in 4221 and 5211 respectively and b1, b2, b3, b4, b5, b6, b7 corresponds to binary partial products. All the other variables are intermediate results. 'sel' corresponds to selection signal for mux (decimal/binary)

Here x in decimal = 1234
x in BCD = 0001001000110100
y in decimal = 2211
y in BCD = 0010001000010001
p in decimal = 2728374
p in 4221 = 0100110101001110010111011000
p in binary = 1010011010000110110110

8.1.5. Radix-4 Binary Multiplier

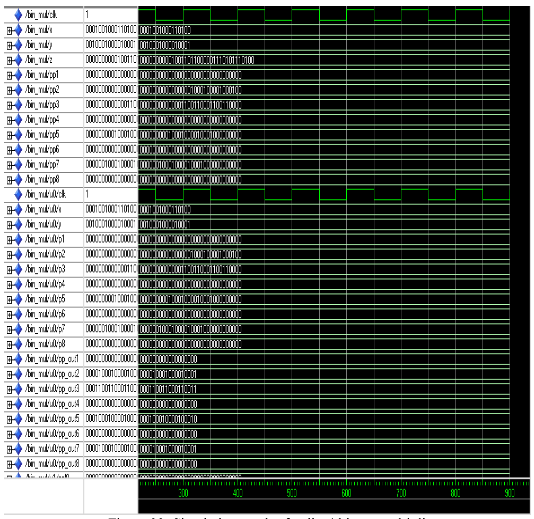


Figure 32: Simulation result of radix-4 binary multiplier

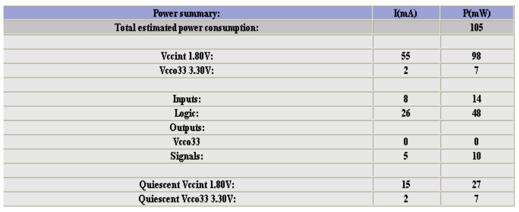
In the above figure x and y are the inputs and z is the final product.p1,p2,p3,p4 represents the partial products. In the first cycle,

x in decimal = 1234
x in binary = 101101
y in decimal = 2211
z in binary = 1010011010000110110110
z in decimal = 2728374
z in BCD = 0010011100101000001101110100

8.2. SYNTHESIS RESULTS

8.2.1. Power Report

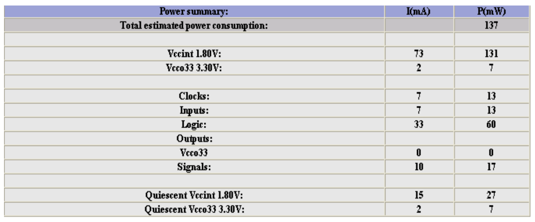
8.2.1.1. SD Radix-10 architecture



Power summary:	I(mA)	P(mW)
Total estimated power consumption:		105
Vccint 1.80V:	55	98
Vcco33 3.30V:	2	7
Inputs:	8	14
Logic:	26	48
Outputs:		
Vcco33	0	0
Signals:	5	10
Quiescent Vccint 1.80V:	15	27
Quiescent Vcco33 3.30V:	2	7

Figure 33: Power report of SD radix-10 architecture

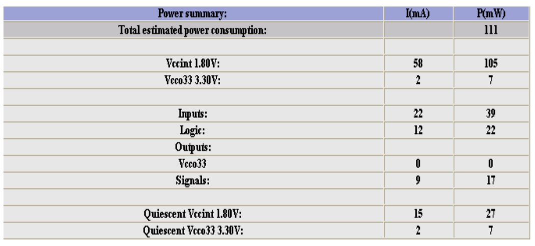
8.2.1.2. SD Radix-5 architecture



Power summary:	I(mA)	P(mW)
Total estimated power consumption:		137
Vccint 1.80V:	73	131
Vcco33 3.30V:	2	7
Clocks:	7	13
Inputs:	7	13
Logic:	33	60
Outputs:		
Vcco33	0	0
Signals:	10	17
Quiescent Vccint 1.80V:	15	27
Quiescent Vcco33 3.30V:	2	7

Figure 34: Power report of SD radix-5 architecture

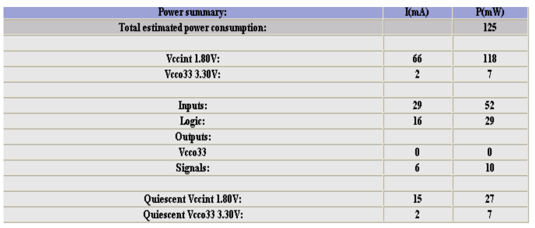
8.2.1.3. SD Radix-10 Architecture using Combined Decimal/Binary CSA



Power summary:	I(mA)	P(mW)
Total estimated power consumption:		111
Vccint 1.80V:	58	105
Vcco33 3.30V:	2	7
Inputs:	22	39
Logic:	12	22
Outputs:		
Vcco33	0	0
Signals:	9	17
Quiescent Vccint 1.80V:	15	27
Quiescent Vcco33 3.30V:	2	7

Figure 35: Power report of SD radix-10 architecture using combined decimal/binary CSA

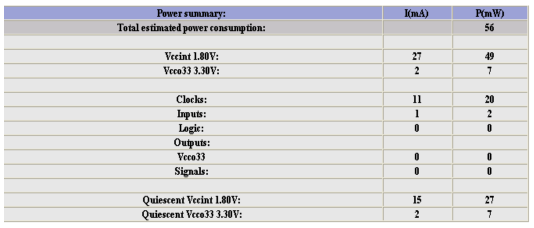
8.2.1.4. SD Radix-5 architecture using combined binary-decimal CSA



Power summary:	I(mA)	P(mW)
Total estimated power consumption:		125
Vccint 1.80V:	66	118
Vcco33 3.30V:	2	7
Inputs:	29	52
Logic:	16	29
Outputs:		
Vcco33	0	0
Signals:	6	10
Quiescent Vccint 1.80V:	15	27
Quiescent Vcco33 3.30V:	2	7

Figure 36: Power report of SD radix-5 architecture using combined decimal/binary CSA

8.2.1.5. Radix-4 Binary Multiplier



Power summary:	I(mA)	P(mW)
Total estimated power consumption:		56
Vccint 1.80V:	27	49
Vcco33 3.30V:	2	7
Clocks:	11	20
Inputs:	1	2
Logic:	0	0
Outputs:		
Vcco33	0	0
Signals:	0	0
Quiescent Vccint 1.80V:	15	27
Quiescent Vcco33 3.30V:	2	7

Figure 37: Power report of radix-4 binary multiplier

8.2.2. MAP REPORT

8.2.2.1. SD Radix-10 architecture

Area report:

Design Summary

Number of errors: 0

Number of warnings: 16

Logic Utilization:

Number of Slice Latches: 123 out of 13,824 1%

Number of 4 input LUTs: 1,016 out of 13,824 7%

Logic Distribution:

Number of occupied Slices: 530 out of 6,912 7%

Number of Slices containing only related logic: 530 out of 530 100%

Number of Slices containing unrelated logic: 0 out of 530 0%

Total Number 4 input LUTs: 1,032 out of 13,824 7%

Number used as logic: 1,016

Number used as a route-thru: 16

Number of bonded IOBs: 68 out of 510 13%

Number of GCLKs: 3 out of 4 75%

Total equivalent gate count for design: 7,056

Additional JTAG gate count for IOBs: 3,264

Delay Report:

Timing Summary:

Minimum input arrival time before clock: 29.986ns

Maximum output required time after clock: 66.863ns

Therefore the total delay = **36.877ns**

8.2.2.2. SD Radix-5 architecture

Area report:

Design Summary

Number of errors: 0

Number of warnings: 12

Logic Utilization:

Number of Slice Latches: 437 out of 13,824 3%

Number of 4 input LUTs: 492 out of 13,824 3%

Logic Distribution:

Number of occupied Slices: 244 out of 6,912 3%

Number of Slices containing only related logic: 244 out of 244 100%

Number of Slices containing unrelated logic: 0 out of 244 0%

Total Number of 4 input LUTs: 469 out of 13,824 3%

Number of bonded IOBs: 68 out of 510 13%

Total equivalent gate count for design: **8,670**

Additional JTAG gate count for IOBs: 8,356

Delay Report:

Timing Summary:

Minimum input arrival time before clock: 31.623ns

Maximum output required time after clock: 64.419ns

Therefore the total delay = **32.796ns**

8.2.2.3. SD Radix-10 architecture using combined decimal/binary CSA

Area report:

Design Summary

Number of errors: 0

Number of warnings: 8

Logic Utilization:

Number of Slice Latches: 88 out of 13,824 1%

Number of 4 input LUTs: 1,045 out of 13,824 7%

Logic Distribution:

Number of occupied Slices: 547 out of 6,912 7%

Number of Slices containing only related logic: 547 out of 547 100%

Number of Slices containing unrelated logic: 0 out of 547 0%

Total Number 4 input LUTs: 1,069 out of 13,824 7%

Number used as logic: 1,045

Number used as a route-thru: 24

Number of bonded IOBs: 229 out of 510 44%

Total equivalent gate count for design: 7,978
Additional JTAG gate count for IOBs: 11,115
Delay Report:
Timing Summary:

Speed Grade: -7
Minimum input arrival time before clock: 9.929ns
Maximum output required time after clock: 77.816ns

Therefore the total delay = **67.887 ns**

8.2.2.4. SD Radix-5 architecture using combined decimal/binary CSA

Area report:
Design Summary

Number of errors: 0
Number of warnings: 11
Logic Utilization:
Number of Slice Latches: 139 out of 13,824 1%
Number of 4 input LUTs: 1,489 out of 13,824 10%
Logic Distribution:
Number of occupied Slices: 791 out of 6,912 11%
Number of Slices containing only related logic: 791 out of 791 100%
Number of Slices containing unrelated logic: 0 out of 791 0%
Total Number 4 input LUTs: 1,527 out of 13,824 11%
Number used as logic: 1,489
Number used as a route-thru: 38
Number of bonded IOBs: 149 out of 510 29%

Number of GCLKs: 4 out of 4 100%

Total equivalent gate count for design: 10,831
Additional JTAG gate count for IOBs: 7,697

Delay Report:
Timing Summary:

Speed Grade: -7
Minimum input arrival time before clock: 28.279ns
Maximum output required time after clock: 79.862ns

Therefore the total delay = **51.583 ns**

8.2.2.5. Radix-4 Binary Multiplier

Design Summary

Number of errors: 0
Number of warnings: 0
Logic Utilization:
Number of Slice Flip Flops: 143 out of 13,824 1%
Number of 4 input LUTs: 387 out of 13,824 2%
Logic Distribution:
Number of occupied Slices: 290 out of 6,912 4%
Number of Slices containing only related logic: 290 out of 290 100%
Number of Slices containing unrelated logic: 0 out of 290 0%
Total Number 4 input LUTs: 573 out of 13,824 4%
Number used as logic: 387
Number used as a route-thru: 42
Number used as Shift registers: 144
Number of bonded IOBs: 68 out of 510 13%

IOB Flip Flops: 1
Number of GCLKs: 1 out of 4 25%
Number of GCLKIOBs: 1 out of 4 25%

Total equivalent gate count for design: 22,941
Additional JTAG gate count for IOBs: 3,312

Delay Report
Timing Summary:

Speed Grade: -6
Minimum period: 3.792ns (Maximum Frequency: 263.713MHz)
Minimum input arrival time before clock: 10.571ns
Maximum output required time after clock: 16.507ns

Therefore the total delay = **5.936 ns**

8.3. COMPARISON

To obtain the area, delay and power estimate's, the designs have been modeled in modelsim and synthesized in Xilinx. We have also compared and evaluated the area and delay figures obtained from synthesis of representative proposals of decimal with a binary radix-4 multiplier. Table 6 shows the comparison results of the various architectures. From the table given below it is clear that the SD radix-10 and the SD radix-5 multiplier with the combined decimal/binary CSA is an interesting option when compared to the representative proposals for decimal multiplication namely SD radix-10 and SD radix-5. SD radix-10 multiplier is an interesting option for high performance with moderate area but when comparing the power delay product the SD radix-5 architecture has about 3.5% improvement. The graphs have been plotted for the area and power for the different architectures.

Table 6: Comparison of various multiplier architectures

Architecture	Gate count	Delay(nS)	Power(mW)	Power-Delay Product(nW)
SD Radix-10	7,056	36.877	105	3.872
SD Radix-5	8,670	32.796	137	4.493
SD Radix-10 using combined decimal/binary CSA	7,978	67.887	111	7.535
SD Radix-5 using combined decimal/binary CSA	10,831	51.583	125	6.447
Radix-4 binary multiplier	22,941	5.936	56	3.3241
SD Radix-10 + Radix-4 binary multiplier	29997	42.813	161	6.892
SD Radix-5 + Radix-4 binary multiplier	31611	38.732	193	7.475

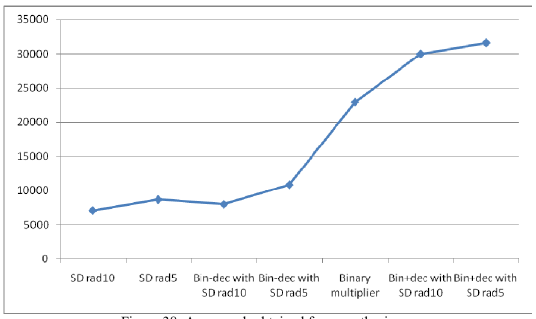


Figure 38: Area graph obtained from synthesis

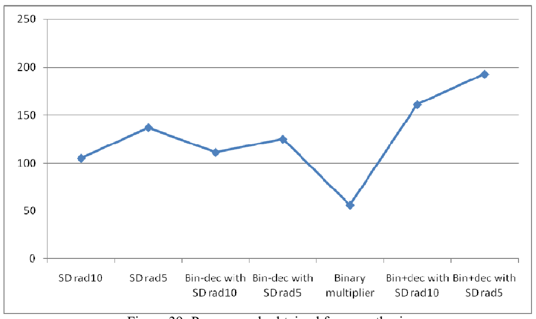


Figure 39: Power graph obtained from synthesis

CHAPTER 8
CONCLUSION AND FUTURE WORK

In this project, we have discussed the different techniques to implement decimal parallel multiplication in hardware. The two architectures for decimal multiplication employing two different Signed Digit encodings for the multiplier that lead to fast parallel and simple generation of partial products have been implemented Also a decimal carry-save algorithm based on unconventional (4221) and (5211) decimal encodings for partial product reduction has been discussed. It makes possible the construction of p:2 decimal CSA trees that outperform the area and delay figures of binary multipliers. The area and delay figures of these decimal multiplier architectures from a comparative study including conventional binary parallel multipliers show that our decimal SD radix-10 multiplier is an interesting option for high performance with moderate area. A new method for the combined computation of binary/decimal multi-operand additions is presented. It relies on a fully reuse of a binary carry-save adder to reduce area, power consumption and design time. There is drastic reduction in are and power consumption of the combined binary/decimal architecture when compared to using both binary and decimal multipliers. Decimal operands are represented in a 4221 coding different than BCD that allows to perform decimal addition via binary carry-save addition and small decimal corrections. As the decimal corrections are computed separately from the carry-save adder tree, there is no impact on the latency of the binary operation.

FUTURE WORK

Future scope of this project is to optimize the decimal fixed-point parallel multipliers to provide pipelined implementations that fit adequately in the dataflow and cycle time of current commercial decimal floating point units.

REFERENCES

[1] Alvaro Vazquez, Elisardo Antelo and PaoloMontuschi. "Improved Design of High Performance Parallel Decimal Multipliers," IEEE Transactions on Computers, vol. 59, May 2010.

[2] A. 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

[3] I.D. Castellanos and J.E. Stine, "Compressor Trees for Decimal Partial Product Reduction," Proc. 18th ACM Great Lakes Symp. VLSI, pp. 107-110, Mar. 2008.

[4] M. Cornea, C. Anderson, J. Harrison, P.T.P. Tang, E. Schneider, and C. Tsen, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," Proc. 18th IEEE Symp. Computer Arithmetic, pp. 29-37, June 2007.

[5] M.F. Cowlishaw, The decNumber ANSI C Library, IBM Corp., 2003.

[6] 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 J. Research and Development, vol. 51, nos. 1/2, pp. 217-227, Jan. 2007.

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

[8] 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.

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

[10] Himanshu Thapliyal, Pallavi Gopineedi and M.B Srinivas, "Novel and efficient 4:2 and 5:2 compressors with minimum number of transistors designed for low-power operations", SPIE Microelectronics, MEMS, and Nanotechnology Symposium, Brisbane, Australia, 11-14 December 2005.(Accepted)

[11] P. H. Abbott., Architecture and Software Support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point Arithmetic, IBM Journal of Research and Development, 43 (1999), pp. 723–760.

[12] G. M. Amdahl, G. A. Blaauw AND F. P. Brooks, Architecture of the IBM System/360, IBM Journal of Research and Development, 8 (1964), pp. 87–53.

[13] S. F. Anderson, J. G. Earle, R. E. Goldschmidt AND D. M. Powers, The IBM System/ 360 Model 91: Floating-point Execution Unit, IBM Journal of Research and Development, 11 (1967), pp. 34–53

[14] Mi Lu, Arithmetic and logic in computer systems, John Wiley and Sons, Edition 2004.

[15] Thomas C. Bartee, Digital Computer Fundamentals, Tata McGraw-Hill, Edition 2005

[16] C Senthilpari, A Low-power and High-performance Radix-4 Multiplier Design Using a Modified Pass-transistor Logic Technique, IETE Journal of Research, pp 149-55, vol.57, 2011.