A PARALLEL
BINARY/DECIMAL FIXED-POINT MULTIPLIER
WITH BINARY PARTIAL PRODUCTS ACCUMULATION

By

Mervat Mohammed Adel Mahmoud

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2011

A PARALLEL BINARY/DECIMAL FIXED-POINT MULTIPLIER WITH BINARY PARTIAL PRODUCTS ACCUMULATION

By

Mervat Mohammed Adel Mahmoud

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Amin Nassar

Professor
Electronics and Communications Engineering
Cairo University

Hossam Fahmy

Associate Professor
Electronics and Communications Engineering
Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2011

Abstract

Combined binary/decimal arithmetic has become an important topic to support decimal and binary applications with high speed and low area. This thesis presents a combined binary/decimal fixed-point multiplier design. Since the partial products accumulation stage has the largest area and delay of the multiplier, it is the most significant stage. A novel binary column tree is shared for binary and decimal reduction tree. A comparison between the proposed design and the previously published designs shows a significant decrease in area with almost the same delay as the fastest known design.

The structure of this thesis is as follows. Chapter 1 presents an overview of decimal and binary computer arithmetic. Chapter 2 summarize the multiplication algorithms. Chapter 3 show a background of decimal multiplication techniques. Chapter 4 focused on previous published combined binary/decimal multipliers. In chapter 5, the proposed parallel combined binary/decimal multiplier is explained. Chapter 6 goes over testing procedure and comparison between the proposed and previous designs in area and delay. Finally we present the conclusions and future work.

Acknowledgment

This research would not have been possible without the support of many people. I would like to sincerely thank my supervisors: Dr. Amin Nassar for his supervision, and Dr. Hossam Fahmy for his step-by-step support, guidance throughout the whole process of research and writing, and patience with me to help this thesis to achieve success.

I would like also to thank Ghada and SilMinds group, Rodaina, Amira, Tarek, and Ramy for their cooperation. And special thanks to A. ElShafiey for his time and expertise in ASIC simulation.

Finally, I would like to thank my parents, sisters, and numerous friends who endured this long process with me, always offering support and love.

Contents

Abstract ii

Acknowledgment iii

Contents iv

List of Figures vii

List of Tables xi

Chapter 1 Introduction 1

Chapter 2 Multiplication Techniques 5

2.1 Signed Integer's Representation 6

2.1.1 Sign-and-magnitude representation 6

2.1.2 Two’s complement representation 8

2.2 Add-and-shift multiplication algorithm 10

2.2.1 Sequential multiplication 15

2.2.2 Parallel (Combinational) multiplication 21

2.2.3 Adders connection approaches 28

2.3 "Composition of smaller multipliers" multiplication algorithm 32

2.4 Bit/Digit serial multiplication algorithm 35
2.5 Booth multiplication algorithm 37
2.5.1 Original Booth algorithm 37
2.5.2 Modified Booth algorithm 40
2.6 Conclusion 43
Chapter 3 Decimal Multipliers 45
3.1 Multiplicand Multiples Generation 47
3.2 Multiplier recoding for multiples selection 55
3.3 Partial Products accumulation 56
3.3.1 Sequential accumulation approach 57
3.3.2 Parallel accumulation approach 59
3.3.3 Decimal adder block implementation 60
3.3.3 Decimal trees 63
3.4 Conclusion 68
Chapter 4 Combined Binary/Decimal Multipliers 69
4.1 Vázquez combined binary/decimal multiplier 69
4.2 Hickmann combined binary/decimal multiplier 72
4.3 Conclusion 73
Chapter 5 Proposed Combined Binary/Decimal Fixed-Point Multiplier 74
5.1 First Proposed Design 75
5.1.1 Multiplicand Multiples Generation Stage 75
5.1.2 Partial Products Selection Stage 80
5.1.3 Partial Products Accumulation Stage 84
5.2 Second proposed Design 88

5.2.1 Multiplicand Multiples Generation Stage 89
5.2.2 Partial Products Selection Stage 91
5.2.3 Partial Products Accumulation Stage 94
5.3 Third Proposed Design 94
5.3.1 Multiplicand Multiples Generation Stage 96
5.3.2 Partial Products Selection Stage 96
5.3.3 Partial Products Accumulation Stage 101
5.4 Final Proposed Design 111
5.5 Conclusion 114
Chapter 6 Verification and Results 115
6.1 Testing 115
6.2 Results 117
6.3 Conclusion 119
6.4 Future Work 119
References 121

List of Figures

Figure 2.1 Multiplication example for (a) sign-and-magnitude representation (b) two's complement representation 9

Figure 2.2 Multiplication of two 4-bit unsigned binary numbers in dot notation 11

Figure 2.3 Partial product selection logic for 8-bit add-and-shift 12

Figure 2.4 Sequential multiplication accumulation schemes 15

Figure 2.5 Sequential multiplication 16

Figure 2.6 High radix sequential multiplication 17

Figure 2.7 High radix sequential multiplier design 19

Figure 2.8 High radix sequential multiplier design using two recoding values, radix4 and/or radix-2, (for radices higher than 4) 20

Figure 2.9 General structure of a combinational full-tree multiplier 22

Figure 2.10 Radix-4 recoding in parallel multiplication 23

Figure 2.11 Partial products for 8-digit multiplication 25

Figure 2.12 Carry save Adder (counter) 26

Figure 2.13 CSA Compressor 27

Figure 2.14 Addition of 8 partial products in an array topology using CSAs and CPA at the end 29

Figure 2.15 Regular tree (a) Using CPAs (b) Using [4:2] compressors....................... 30

Figure 2.16 Irregular tree topology using [3:2] CSAs and CPA for last level 31

Figure 2.17 Addition of four 4-bit partial products (a) using Wallace tree (b) using Dadda tree 32

Figure 2.18 Implementation of 8 × 8 multiplier using four 4 x 4 multipliers 33

Figure 2.19 Using 4 × 4 multiplier with 8-bit product for various multiplier arrays up to 64 × 64 34

Figure 2.20 4×4 Bit Serial Multiplier 36

Figure 2.21 Digit Serial Multiplier. 37

Figure 2.22 16 bit Booth 2 multiply 42

Figure 3.1 4-bit Decimal multiplication example... 46

Figure 3.2 BCD multiplication by two 48

Figure 3.3 Multiplicand multiples generation (generate all multiplicand multiples)... 49

Figure 3.4 Decimal multiplicand multiples generation sets 50

Figure 3.5 Signed digit recoding by Tomás Lang and Alberto Nannarelli 50

Figure 3.6 signed digit-by-digit multiplier block 52

Figure 3.7 Sequential Decimal Multiplication Design 58

Figure 3.8 Sequential Decimal Multiplication Design 59

Figure 3.9 Parallel Decimal Multiplier Design 60

Figure 3.10 Generic design for the 3:2 decimal CSA 62

Figure 3.11 Decimal carry-save addition example (a) in BCD-4221 format (b) in BCD-5211 format 63

Figure 3.12 (a) n-digit radix-10 CSA (b) m-digit radix-10 counter. 64

Figure 3.13 Array for partial products. Solid circles indicate BCD digits, hollow circles indicate carry bits. 64

Figure 3.14 A Radix-10 Combinational Multiplier Adder tree....................................65
Figure 3.15 (a) 4-bit 3:2 decimal CSA (b) decimal multiplication by 2 for BCD-4221 66
Figure 3.16 16:2 decimal CSA tree 67
Figure 3.17 basic decimal column adder scheme for N=33 addends 68
Figure 4.1 Vázquez binary/decimal multiplier. 70
Figure 4.2 Vázquez binary/decimal CSA Tree. 71
Figure 4.3 Binary/Decimal multiplication by two block. 71
Figure 4.4 Hickmann binary/decimal multiplier. 72
Figure 4.5 Hickmann split binary/decimal CSA Tree. 73
Figure 5.1 First combined binary/decimal multiplier block diagram. 76
Figure 5.2 Binary multiples generation 77
Figure 5.3 Decimal multiples generation 77
Figure 5.4 Multiplexers design for each multiplier digit. 83
Figure 5.5 Binary column tree scheme. 85
Figure 5.6 CSA binary tree (for 32 digits, 4-bit). 86
Figure 5.7 64-bit binary CSA tree for the 16 partial products out of MUXs3. 87
Figure 5.8 CSA binary tree (for 16 partial products, 64-bit). 87
Figure 5.9 Second combined binary/decimal multiplier block diagram. 89
Figure 5.10 Three input, 64-bit, Carry Save Adder 90
Figure 5.11 1-bit Carry Save Adder 90
Figure 5.12 binary multiples generation 91
Figure 5.13 Partial products selection. 92

Figure 5.14 Proposed combined binary/decimal multiplier (a) shared design, (b) split design. 95

Figure 5.15 Used binary multiples generation. 96

Figure 5.16 Partial products selection. 98

Figure 5.17 Multiplexers design for each multiplier digit. 99

Figure 5.18 Proposed binary column tree scheme (S and C maximally 8 bits). 101

Figure 5.19 33 digits CSA binary tree. 102

Figure 5.20 The four binary bit-vectors after rearranging. 103

Figure 5.21 The six decimal bit-vectors after rearranging. 103

Figure 5.22 8-bit binary to decimal converter example. 105

Figure 5.23 8-bit binary to decimal converter block diagram. 106

Figure 5.24 9-bit, max. value 319, binary to decimal converter block diagram. 107

Figure 5.25 (a) Tiny split binary tree. (b) Tiny split decimal tree. 108

Figure 5.26 Tiny shared binary/decimal tree 108

Figure 5.27 Split Binary/Decimal Kogge-Stone based carry propagate adder 109

Figure 5.28 Shared Binary/Decimal Kogge-Stone based carry propagate adder 110

Figure 5.29 Final proposed Binary/Decimal Multiplier design (split scheme). 112

Figure 5.30 The three decimal bit-vectors after rearranging. 112

Figure 5.31 Tiny split binary and decimal trees. 113

Figure 5.32 Tiny shared binary/decimal tree. 113

List of Tables

Table 2.1 radix-4 multiplier recoding 13

Table 2.2 Radix-4 recoding in parallel multiplication 23

Table 2.3 Summary of number of partial products for various multipliers using small multiplier where n is the operand size. 35

Table 2.4 Original booth recoding scheme 38

Table 2.5 Booth radix 4 recoding scheme 41

Table 3.1 Complexity of digit-by-digit products for different ranges of decimal inputs 51

Table 3.2 Signed digit-by-digit products 52

Table 3.3 BCD coding formats 53

Table 3.4 Example of multiplier recoding 56

Table 5.1 BCD-8421 to BCD-5421 conversion 78

Table 5.2 BCD-5421 to BCD-8421 conversion 79

Table 5.3 9’s complement of BCD-8421 digits 80

Table 5.4 Binary multiplicand multiples selection. 81

Table 5.5 Decimal multiplicand multiples selection. 82

Table 5.6 Binary multiplicand multiples selection. 92

Table 5.7 Binary partial products selection according to Booth4 recoding. 98

Table 6.1 Test cases..116

Table 7.1 Area/Delay figure for different Binary/decimal multipliers using FPGA virtex5. 118

Table 7.2 Area/Delay figure for different Binary/decimal multipliers using FPGA virtex5. 118

Table 7.3 Area/Delay figure for different binary/decimal multipliers using ASIC low power CMOS 130nm technology. 119

Chapter 1

Introduction

Decimal arithmetic is the norm in human calculations. Early mechanical computers were almost all decimal machines; they mirrored human manual calculations of commerce and science. Also the first general-purpose electronic computer, ENIAC, in 1946, holds a ten-digit decimal number in memory. However, in 1961, most computers turned to binary representation of numbers as shown by a survey in computer systems in USA. It reported that “131 utilize a straight binary system internally, whereas 53 utilize the decimal system (primarily binary coded decimal)...” [2]. The use of binary arithmetic reduces the number of components and is simpler. The difference between data and hardware representation is controlled using software programs. Today, few computing systems include decimal hardware. However, the growing importance of commercial and financial which deals with decimal data and the quick advancement of technology speed, support of decimal arithmetic is regaining popularity in the computing community. Also Initial benchmarks indicate that some applications spend 50% to 90% of their time in decimal processing. In 2002, ‘telco’ benchmark by Cowlishaw shows that the decimal processing overhead could reach over 90% in a telephone company’s daily billing application. [2]

The need for decimal in hardware is urgent. So some companies added a hardware decimal arithmetic unit to its processor (i.e. The IBM z9 Decimal floating point Arithmetic Unit in 2007) [4].

Decimal arithmetic units are inherently more complex than binary arithmetic units, since they need to handle a wider range of digits, 10 digits versus 2 digits for binary arithmetic. Also the six invalid BCD-8421 digits need a correction blocks. Therefore most computers today support binary in hardware where it is simpler, faster and less in area and cost compared to decimal.

However, Binary arithmetic gives an inexact solution when decimal fractions are involved. It implies inexact conversions between binary and decimal representations. For example, using the Java or C double binary floating point for multiplying 0.1 × 8 gives the result 0.8000000000000000444089209850062616169452667236328125 but adding 0.1 to itself 8 times give a different answer 0.79999999999999993338661852249060757458209991455078125. The two results would not compare equal, and further, if these values are multiplied by ten and rounded to the nearest integer below (‘floor’ function), the result will be 8 in one case and 7 in the other.

Another example, consider a calculation involving a 5% sales tax on an item such as a $0.70 telephone call, rounded to the nearest cent, Using double binary

floating-point, the result of 0.70 x 1.05 is 0.73499999999999998667732370449812151491641998291015625; the result should have been 0.735, which would be rounded up to $0.74, but instead the rounded result would be $0.73. [23]

Now, decimal arithmetic is supported through software on most machines. And while using decimal floating-point arithmetic software gives the right answer, sometimes the software conversions between decimal and binary are

time consuming. For example, in some applications like databases the conversion time between binary and decimal using software programs takes large time. Initial benchmarks indicate that some applications spend 50% to 90% of their time in decimal processing, because software decimal arithmetic takes a 100× to 1000× over hardware time. [2]

Moreover, in other applications like simulation programs, if the simulation takes long time and conversions only needed at the start and end of the simulation, binary hardware arithmetic will be faster. So conversion problem between hardware and application data representation depends on how frequent conversions are needed.

Binary arithmetic hardware is better than decimal arithmetic hardware in some applications which do not need high accuracy or have a long run time or do not deal with decimal numbers, such as numerical analysis, scientific computing, simulations, and addressing. Decimal arithmetic provide higher accuracy in financial and commercial applications like banking, tax calculations, currency conversion, insurance, accounting which need high precision. Decimal data in these applications can not be represented exactly using binary arithmetic, also it is better to use decimal arithmetic in databases applications where most databases data types is decimal or integer ≈ 98.7% [2].

Optimally two hardware arithmetic units, binary and decimal, are needed in processors. [2]

This thesis proposes a combined binary/decimal multiplier with binary partial products reduction tree. Chapter 2 describes the multiplication algorithms. Chapter 3 and 4 show a background of decimal multiplication techniques and previous proposed combined binary/decimal multipliers respectively. In chapter 5 the proposed parallel combined binary/decimal multiplier is explained. Chapter 6 goes over testing procedure and comparison

between the proposed and previous designs in area and delay. Finally present the future work.

Chapter 2

Multiplication Techniques

In this chapter we consider the multiplication algorithms for signed integers. The multiplication operation is

$$P = A \times B \tag{2.1}$$

where A is the multiplicand, B is the multiplier, and P is the product. The multiplication operands, A and B, are represented by a sign bit and an n-bit magnitude

$$A = S_a \, a_{n\text{-}1} \, . \, . \, . \, . \, . \, . \, . \, . \, . \, . \, a_2 \, a_1 \, a_0 \tag{2.2}$$

$$B = S_b \, b_{n\text{-}1} \, . \, . \, . \, . \, . \, . \, . \, . \, . \, . \, b_2 \, b_1 \, b_0 \tag{2.3}$$

and the result P is represented by a sign and a 2n-bit magnitude

$$P = S_P \, P_{2n\text{-}1} \, . \, . \, . \, . \, . \, . \, . \, . \; P_2 \, P_1 \, P_0 \tag{2.4}$$

where S_a, S_b, and S_P are the sign bit of A, B, and P respectively. The digits a_i, b_i, and P_i are number digits (i.e. binary digit, 0 and 1, or decimal digit, from 0 to 9, etc.).

In this chapter the signed integer's representation and the multiplication techniques are discussed focusing on binary and decimal multiplications.

2.1 **Signed Integer's Representation**

The signed integers can be represented in two ways: sign-and-magnitude representation and two's complement representation. [5]

2.1.1 Sign-and-magnitude representation

In sign-and-magnitude representation, the operands are represented by a sign bit and an n-bit magnitude, and the product is represented by a sign and a 2n-bit magnitude where

$$v(sign(P)) = v(sign(A)) \cdot v(sign(B)) \quad (2.5)$$

$$|P| = |A| . |B| \quad (2.6)$$

The sign bit takes the values '0' and '1' for positive and negative signs, respectively. The $v()$ notation in equation 2.5 represents the value of the sign bit. The implementation of product sign can be implemented separately from magnitude using XOR gate where operands with similar signs give product with the same sign, and operands with different signs give negative product sign.

$$\ell(sign(P)) = \ell(sign(A)) \ XOR \ \ell(sign(B)) \quad (2.7)$$

where $\ell(\ )$ notation in equation 2.7 represents the logic of the sign bit.

For any radix r, the operands magnitude values are

$$|A| = \sum_{i=0}^{n-1} a_i.r^i \qquad (0 \le a_i \le r-1\ ,\ 0 \le |A| \le r^n - 1) \qquad (2.8)$$

$$|B| = \sum_{i=0}^{n-1} b_i r^i \qquad (0 \le b_i \le r-1\ ,\ 0 \le |B| \le r^n - 1) \qquad (2.9)$$

And the product can be represented as

$$|P| = |A|\ .\ |B| = \sum_{i=0}^{2n-1} p_i r^i \qquad (0 \le p_i \le r-1\ ,\ 0 \le |P| \le (r^n - 1)^2) \quad (2.10)$$

The basic method to implement the value of the product is to multiply the multiplicand by each digit of the multiplier regarding its weight then adding these values.

$$|P| = |A| \sum_{i=0}^{n-1} b_i r^i \qquad (2.11)$$

Different methods are used to implement the magnitude of the product like add-and-shift, composition of smaller multiplications, digit serial

multiplication, and booth multiplication. These methods are discussed in next sections of this chapter.

2.1.2 Two’s complement representation

Two’s-complement is a representation in which negative numbers are represented by the two's complement of the absolute value. An n-bit two's complement number can represent every integer in the range -2^{n-1} to $+2^{n-1}-1$. For multiplication, by representing each operand by n-bit vector, the product is $2n$-bit vector and has values in the range $(-2^{n-1})(2^{n-1}-1)$ to $(-2^{n-1})(-2^{n-1}) = 2^{2n-2}$. [12]

Let A_R , B_R , and P_R are the corresponding positive integer representations of A, B, and P, respectively. When the two operands are positive, they are represented as A_R , B_R so the product will be $A_R \times B_R$. And when the two operands are negative, they are represented by their two's complement value $(2^n - A_R)$, $(2^n - B_R)$ so the product will be $(2^n - A_R) \times (2^n - B_R)$ and it’s a positive value. However, when one of the operands is positive (e.g. A) and the other operand is negative (i.e. B), they are represented as A_R , $(2^n - B_R)$ so the product will be the two’s complement of $A_R \times (2^n - B_R)$ because it has a negative value. The multiplication algorithm can be described as in [12].

$$P_R = \begin{cases} A_R B_R & \text{if } A \geq 0, B \geq 0 \\ 2^{2n} - (2^n - A_R)\, B_R & \text{if } A < 0, B \geq 0 \\ 2^{2n} - A_R\, (2^n - B_R) & \text{if } A \geq 0, B < 0 \\ (2^n - A_R)\, (2^n - B_R) & \text{if } A < 0, B < 0 \end{cases} \quad (2.12)$$

Figure 2.1(a) shows the multiplication of $-9_{10} \times 3_{10}$ in base 2 in sign-and-magnitude representation. The 4-bit multiplicand magnitude $9_{10} = 1011_2$ is stored in an 8-bit word as 00001011_2. Then multiply it to each bit of the multiplier magnitude $3_{10} = 0011_2$ regarding its weight. Then add these multiplication values. The sign value is the XOR of multiplicand sign and multiplier sign which is 1_2.

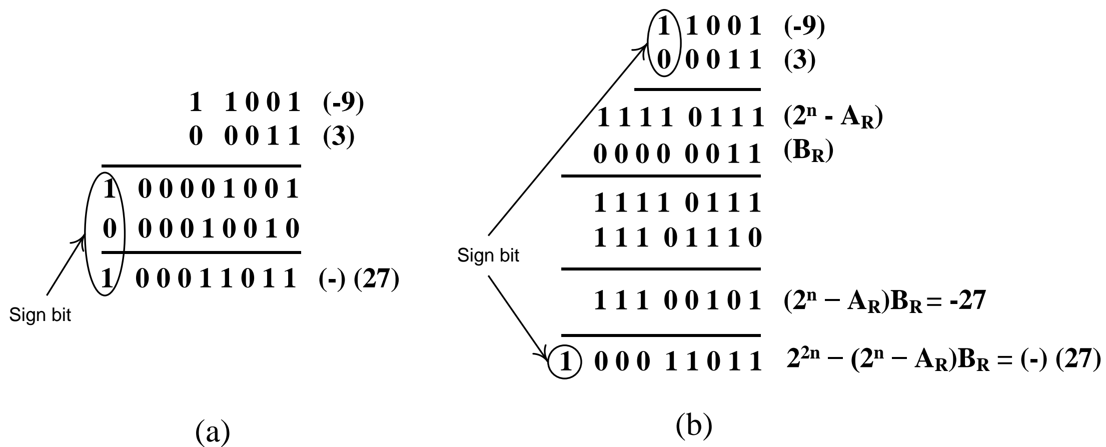


Figure 2.1 Multiplication example for (a) sign-and-magnitude representation (b) two's complement representation

However, Figure 2.1(b) shows the multiplication of $-9_{10} \times 3_{10}$ in base 2 in two's complement representation. The multiplicand and multiplier is stored in 8-bit word in its two's complement form if it has negative sign. Then multiply them as in sign-and-magnitude representation. If the product is negative according to eqn.2.8, the two's complement of it is determined.

Sign-and-magnitude representation is preferred in multipliers implementation where it does not require the two's complement conversion steps. The two's complement representation is used in addition and subtraction circuitry where it

does not need to examine the signs of the operands to determine whether to add or subtract.

In the next sections of this chapter, the multiplication algorithms: Add-and-shift, Composition of smaller multipliers, Bit/Digit serial, and Booth multiplication are considered. Also the sequential and parallel approaches are discussed.

2.2 Add-and-shift multiplication algorithm

The common and simplest method of multiplication is the add-and-shift multiplication algorithm. Let the two multiplication operands A and B called multiplicand and multiplier respectively, and each operand has n bits. This algorithm conditionally adds together copies of the multiplicand according to multiplier bits to produce the final product based on the following equation [12].

$$A \times B = \sum_{i=0}^{n-1} A \cdot b_i r^i \tag{2.13}$$

Figure 2.2 shows the multiplication of two 4-bit unsigned numbers. The two operands A and B are shown at the top. Each of the following four rows corresponds to the product of the multiplicand A and a single digit of the multiplier B generating four partial products (PPs), with each row shifted one bit to the left. Then all partial products are added to generate the final product (P). [19]

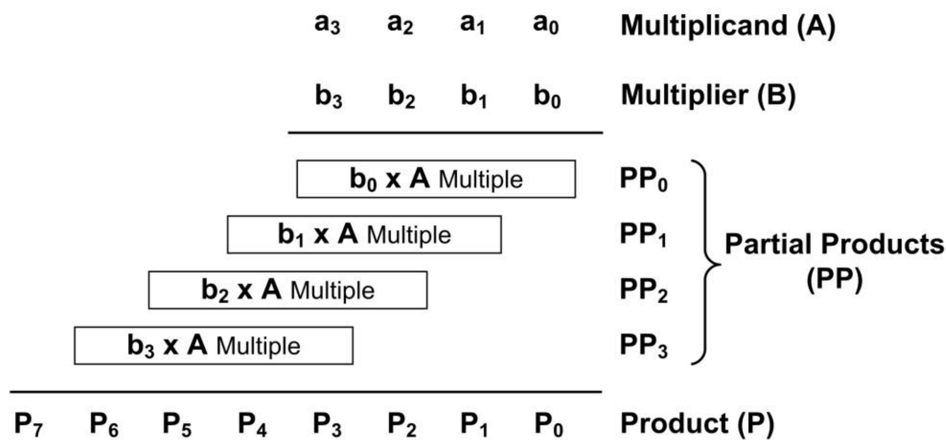


Figure 2.2 Multiplication of two 4-bit unsigned binary numbers in dot notation

High Radix Multiplication

The multiplier digits b_i can represent one bit in radix-2 design, binary system, or a set of bits (2, 3, 4 ….etc.) in higher radix designs. The simplest implementation is obtained by using radix-2 since the multiplier digits are either 1 or 0 so the multiples of the multiplicand are either A or zero and the number of partial products generated are n, where n is the number of bits in B. This number of partial products is reduced by using higher radices. For k-bit multiplier digit, b_i , the number of partial products are n/k, where $k = log_2 r$, and r is the radix. But the number of multiplicand multiples are $r - 1$. [5]

For example, radix-4 has A to $3A$ multiples to be generated, $n/2$ partial products. Radix-8 has A to $7A$ multiples to be generated, $n/3$ partial products. Radix-16 has A to $15A$ multiples to be generated, $n/4$ partial products. For decimal multiplication, the multiples from A to $9A$ are generated, where Binary Coded Decimal, BCD, format is used.

In Binary Multiplication, b_i is in $\{0, 1\}$ set, so each term $b_i . A$ is either 0 or A. Figure 2.3 shows the partial product selection logical AND for 8-bit multiplicand [8]. Thus the problem of add-and-shift binary multiplication

reduces to adding n partial products, each of which is 0 or a shifted version of the multiplicand A.

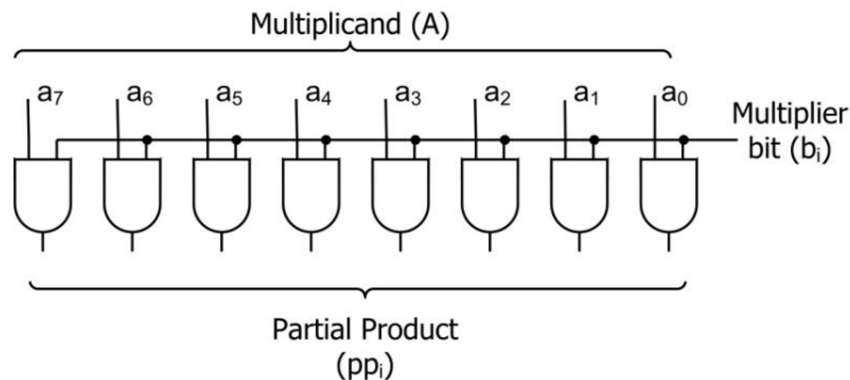


Figure 2.3 Partial product selection logic for 8-bit add-and-shift

In Decimal Multiplication, b_i is in $\{0, 9\}$ set. Each term $b_i . A$ is one of the multiplicand multiples $0, A, 2A, 3A, 4A, 5A, 6A, 7A, 8A, 9A$. Decimal multiplication needs to generate the multiplicand multiples then select the suitable multiple due to multiplier digits to generate the partial products. Decimal adders are used to accumulate the partial products.

High Radix Multiplier digit, b_i, Recoding

The main problem with high radix multiplication is the digit multiplication, since now the digit of the multiplier has r values.

For radix-4, the multiplier digit b_i, corresponding to two bits, has the values 0, 1, 2, and 3. When multiplying these digit values by the multiplicand, the generation of the multiples A and $2A$ by shifting the multiplicand A, are simple. But the multiple $3A$ requires an addition to A and $2A$. To avoid this multiple, the multiplier is recoded into a signed-digit set as {–1, 0, 1, 2} since

the multiplication by these values is simple needing only the complementation and shifting of the multiplicand. [5]

The recoding algorithm in [5] recodes the digits of the multiplier from Least Significant Digit (LSD). Using z_i as radix-4 recoded multiplier signed digit, and c_i as the carry bit. The recoding produces z_i such that

$$z_i = b_i + c_i - 4c_{i+1} \quad (2.14)$$

Where $b_i \in \{0, 1, 2, 3\}, c_i \in \{0, 1\}$ and $z_i \in \{-1, 0, 1, 2\}$. The carry c_{i+1} is selected so that the value $b_i = 3$ is avoided. Consequently, when $b_i + c_i \geq 3$, $c_{i+1} = 1$ and $z_i = b_i + c_i - 4$. This recoding is described by the following table.

$b_i + c_i$	z_i	c_{i+1}
0	0	0
1	1	0
2	2	0
3	–1	1
4	0	1

Table 2.1 radix-4 multiplier recoding

For further reduction of partial products, a radix higher than 4 is used. The algorithm is a direct extension of the radix-4 case, for example, radix-8 and radix 16.

For radix-8, the multiplier recoded into the digit set {–3, –2, –1, 0, 1, 2, 3, 4}, where

$$z_i = b_i + c_i - 8c_{i+1} \tag{2.15}$$

where $b_i \in \{0, 1, 2, 3\}, c_i \in \{0, 1\}$. The main problem with the implementation of this multiplication is the generation of $3A$, where it needs an extra addition step of $2A$ plus A. [5]

The extension to even higher radices requires the generation of more multiplicand multiples. An alternative is to use several radix-4 and/or radix-2 stages in one iteration [5].

For radix-16, the multiplier can be recoded into the digit set {–7, –6, ……, 0, 1, ……, 7, 8} where

$$z_i = b_i + c_i - 16c_{i+1} \tag{2.16}$$

where $b_i \in \{0, 1, \dots\dots, 15\}, c_i \in \{0, 1\}$, and $c_{i+1} = 1$ when $b_i + c_i \geq 9$. This requires a generation of many multiplicand multiples. So this recoding can be performed by recoding the multiplier b_i into two redundant radix-4 digits u_i and w_i [5] such that

$$z_i = 4u_i + w_i \quad \textit{where} \quad u_i, w_i \in \{-2, -1, 0, 1, 2\} \tag{2.17}$$

So only the multiple $2A$ and $4u_i$ are generated then an adder is used.

In the next two sections we will consider the sequential and combinational implementation techniques for add-and-shift algorithm for binary and high radix multipliers.

2.2.1 Sequential multiplication

Sequential multiplication can be done using a cumulative partial product register (initialized by 0) and successively adding to it the properly shifted terms $b_i.A$. Since each term to be added to the cumulative partial product register is shifted by one digit with respect to the preceding one, the cumulative partial product register is shifted one digit in order to align its digits with those of the next partial product.

Two schemes of this algorithm can be derived, depending on whether the partial product term $b_i.A$ are processed from top to bottom or from bottom to top (see Figure 2.4) depending on starting from the least significant digit or most significant digit of the multiplier, and right shift or left shift the cumulative partial product register, respectively. [14]

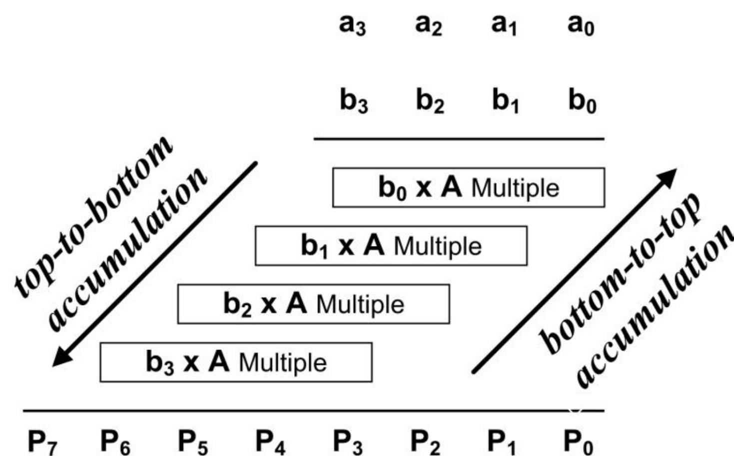


Figure 2.4 Sequential multiplication accumulation schemes

The hardware implementation of top to bottom accumulation multiplication algorithm is more logical and has less area so that it is the preferred method.

Sequential multiplication algorithm with top-to-bottom accumulation

In multiplication with top to bottom accumulation, a right shift cumulative partial product register is used. Figure 2.5 shows a right shift sequential multiplier using radix-2, binary. [14]

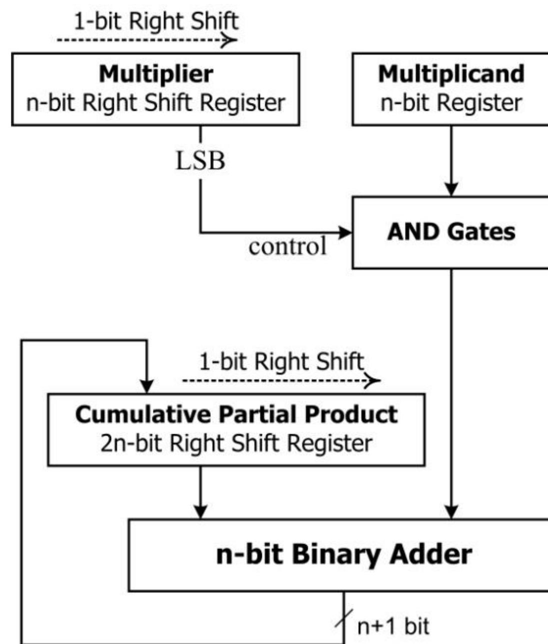


Figure 2.5 Sequential multiplication

For high radix sequential multiplication, the 1-bit right shift is replaced by a 1-digit right shift. Also the AND gate block which chooses between 0 and multiplicand A due to multiplier bits b_i is replaced by a multiplicand multiples generator block to generate the multiplicand multiples for the radix used then a selector block is added to select the suitable multiplicand multiple due to multiplier digits b_i. Figure 2.6 shows a right shift sequential multiplier for high radices.

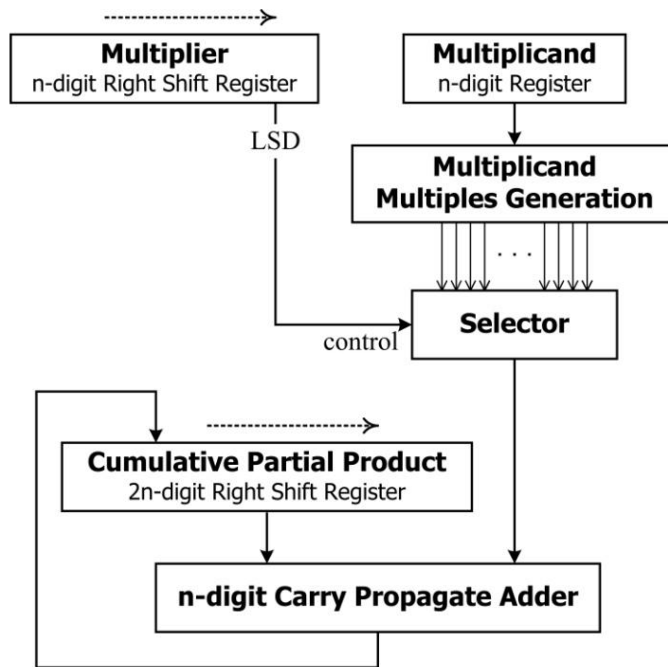


Figure 2.6 High radix sequential multiplication

Generally for right shift sequential multiplication the following steps are performed [14]

1. Store multiplicand A in an n-digit register, multiplier B in an n-digit right shift register, and initialize the cumulative partial product register with zero.

2. Add 0 or one of the multiplicand multiples to the left $n + 1$ digit of the cumulative partial product register according to multiplier least significant digit b_0.

3. Shift the cumulative partial product register and multiplier register one digit to the right.

4. Repeat step 2 and 3 till the end of the n iterations.

5. After n iteration, the final product is stored in the cumulative partial product register.

The accumulation of partial products can be described as

$$PP_{i+1} = (\ PP_i\ +\ b_i.A.r^n\ )\ r^{-1} \quad \text{with} \quad PP_0 = 0\ , PP_n = P \qquad (2.18)$$

|—— add ——|

|—— shift right ——|

because the right shifts will cause the first partial product to multiplied by r^{-n}, multiplicand A is pre-multiplied by r^n to offset the effect of the right shifts. This pre-multiplication is done simply by aligning A with the upper half of the $2n$-cumulative partial product register in the addition steps. [14]

The control portion of the multiplier, which is not shown in the figures, consists of a counter to keep track of the number of iterations and a simple circuit to effect initialization and detect termination. [5]

The delay of the sequential multiplier shown in Figure 2.6 is equal to $n\ \times t_{CPA}$ where t_{CPA} is the delay of Carry Propagate Adder (CPA). It has a large delay where the delay of n-bit ripple carry adder is of O(n), and the carry lookahead and other prefix adders are of O($\log n$). To decrease this delay a carry save adder is used for the iterations and a CPA is used at the end of iterations as shown in Figure 2.7. [5]

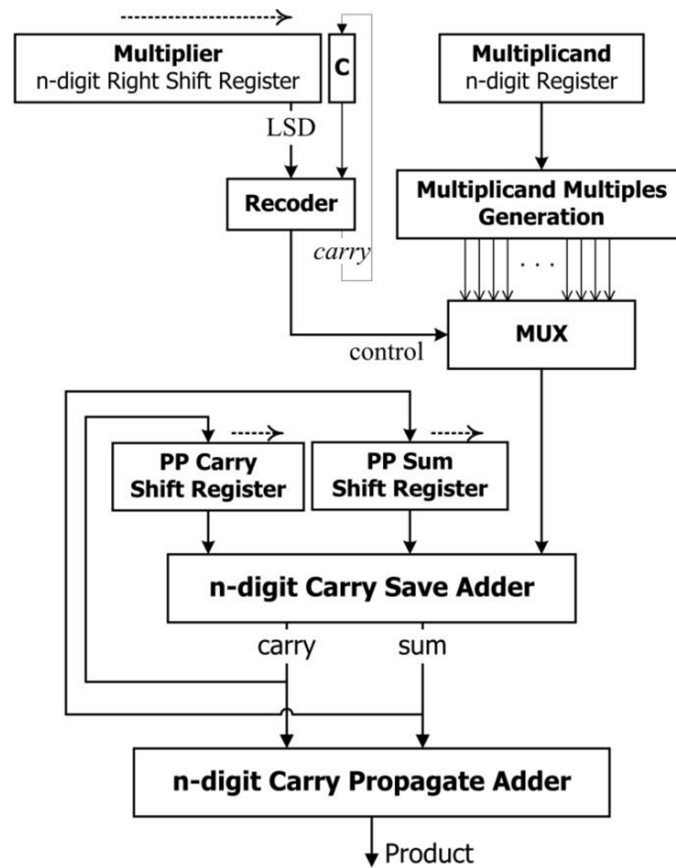


Figure 2.7 High radix sequential multiplier design

To avoid the generation of large number of multiplicand multiples in high radices, the multiplier digits is recoded into two values u_i and w_i each one follow radix-4 and/or radix-2 recoding. In the iterations, two CSAs are used for each recoding digit as shown in Figure 2.8. [14]

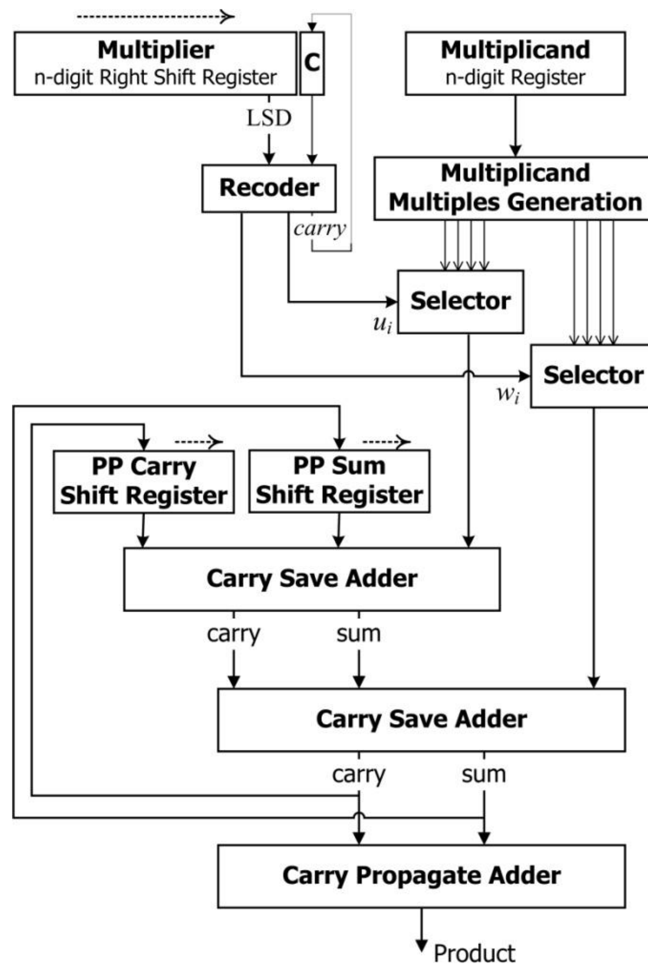


Figure 2.8 High radix sequential multiplier design using two recoding values, radix4 and/or radix-2, (for radices higher than 4)

The sequential multiplication can be divided into three stages as follows

Stage1: Multiplier recoding.

Stage2: Multiplicand multiples generation.

Stage3: Sequential addition and shift.

2.2.2 Parallel (Combinational) multiplication

Instead of performing the multiplication in several cycles (iterations) in sequential multiplication, parallel multiplication reuses the hardware to perform the operation in a single cycle. [5]

In parallel multiplication, all the n/k partial products, PPs, of the multiplicand are produced at once. For each digit, k-bit, generate the suitable partial product according to multiplier digit b_i. Then an n/k-input CSA tree is used to reduce the partial products to two operands for the final addition. Finally, a Carry Propagate Adder CPA is used to generate the final product. [19]

$$P = \sum_{i=0}^{n-1} A \cdot b_i r^i \qquad (2.19)$$

In this case all the multiples are obtained simultaneously and applied as operands in the first level of the tree. Therefore, the recoding has to be done in a parallel fashion.

Figure 2.9 shows the general structure of a full tree multiplication. Various multiples of the multiplicand are generated corresponding to multiplier radix formed at the top. These multiples are added in a combinational partial products CSA reduction tree, which produces their sum in redundant form (carry save form). Finally, a CPA is used to generate the final product result. [14]

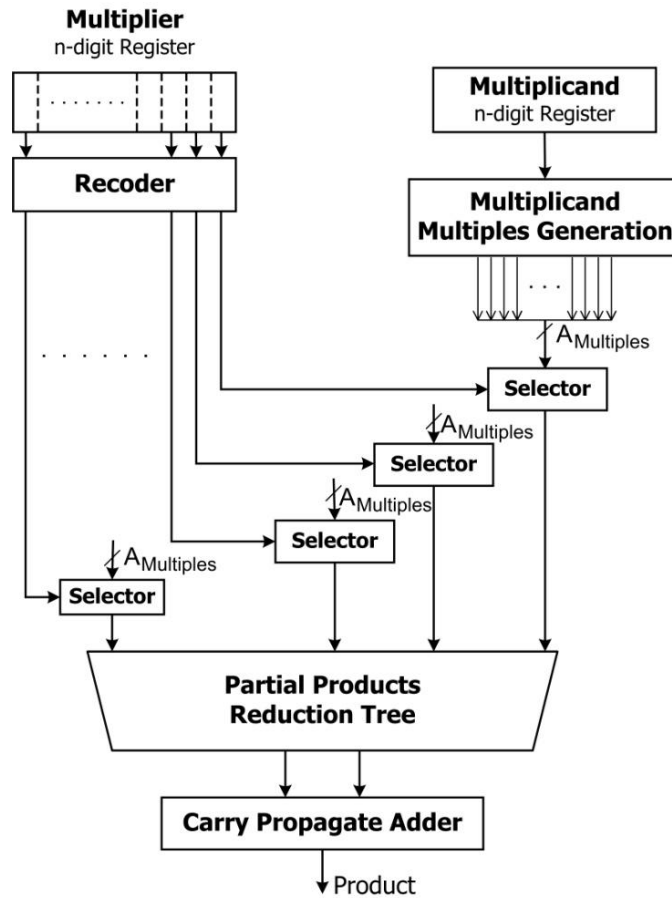


Figure 2.9 General structure of a combinational full-tree multiplier

The parallel multiplication can be divided into three stages

Stage1: Multiplier recoding.

Stage2: Shifted multiplicand multiples generation $(A.\,b_i)r^i$.

Stage3: Partial products accumulation.

Stage1: Multiplier recoding

As discussed for the sequential case, radix-4 multiplier digits have the values 0, 1, 2, and 3. The generation of the multiples A and $2A$ are simple, but the multiple $3A$ requires an addition. To avoid this multiple, the multiplier is

recoded into a signed-digit set {−1, 0, 1, 2} since the multiplication by these values is simple, the parallel multiplication recoding produce z_i such that

$$b_i = w_i + 4t_i \quad , \text{and} \quad z_i = w_i + t_{i-1} \tag{2.20}$$

where $w_i \in \{-2, -1, 0, 1\}$, $t_i \in \{0, 1\}$ so $z_i \in \{-2, -1, 0, 1, 2\}$. [12] Table 2.2 and Figure 2.10 show the radix-4 multiplier recoding [5]

b_i	w_i	t_i
0	0	0
1	1	0
2	−2	1
3	−1	1

Table 2.2 Radix-4 recoding in parallel multiplication

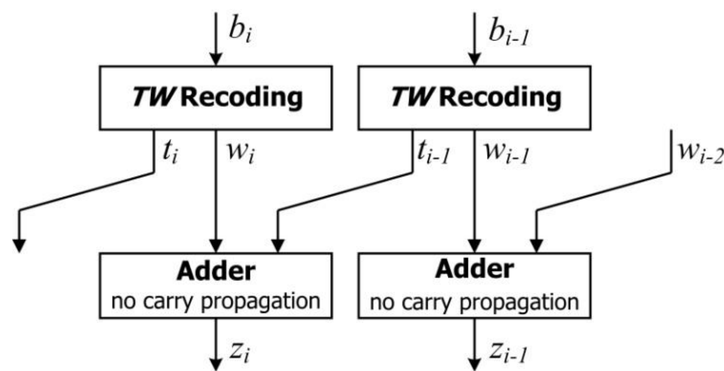


Figure 2.10 Radix-4 recoding in parallel multiplication

For parallel multiplication the addition step which generates z_i should be performed without carry propagation. This is achieved if

$$-2 \leq w_i \leq 1 \quad \text{, and} \quad 0 \leq t_{i-1} \leq 1 \tag{2.21}$$

consequently, the algorithm is as equation 2.22. [12]

$$\{t_i,\ w_i\} = \begin{cases} (0,\ b_i) & b_i \leq 1 \\ (1,\ b_i - 4) & b_i \geq 2 \end{cases} \tag{2.22}$$

The extension to higher radices has the same idea of radix-4 parallel multiplication recoding; trying to reduce the number of multiples which need an extra addition step.

Stage2: Multiplicand multiples generation

For a certain multiplier B, the multiplicand multiples due to the multiplier digits b_i are defined as $m[i]$ where

$$m[i] = A.b_i r^i \quad \text{, where } 0 \leq i \leq n/k - 1 \tag{2.23}$$

This corresponds to a multiplication of the multiplicand by each digit, i, of the multiplier and an arithmetic shift left by i digits. Figure 2.11 shows the resulting partial products in dot notation form where each dot represents one digit. [12]

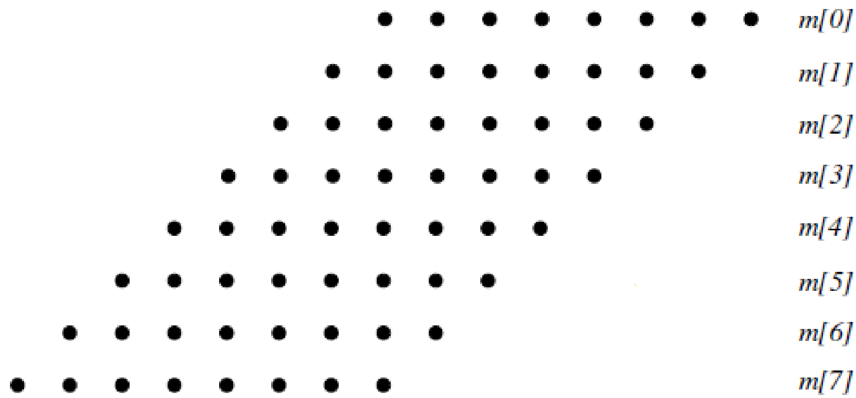


Figure 2.11 Partial products for 8-digit multiplication

In parallel multiplication, all possible multiplicand multiples due to the range of multiplier digit, $0 \leq b_i < r - 1$, are generated firstly. Then for each multiplier digit b_i select the suitable multiple. For radix-2, the result of this digit multiplication is n partial products and the shift is one bit. In general, for any radix r, the number of partial products are n/k, where n is the number of multiplier bits, and k is the number of bits in each digit. So i is in the range $0 \leq i \leq n/k - 1$ and the shift is one digit, k bits. The high radix multiplication is used to reduce the number of multiples and, therefore, the complexity of the partial products addition, but the number of multiplicand multiples needed to be generated increase.

An AND-OR network for each bit is used in the implementation of the multiples generation circuit to select among the different possible multiples.

Multiples like $2A$, $4A$, $8A$, and $16A$ are generated by only shifting. It is fast, easy, and has no additional area cost. Some other multiples like 3A, $5A$, and $7A$ need an addition steps which take large delay and area. They have different techniques to be generated. Some of them will be discussed in the next chapter.

The trade-offs for high radix multiplication are: higher radix gives more multiplicand multiples and more complex multiples circuit which has extra delay in some radices, but it leads to less partial products and more simple reduction tree having less delay.

Stage3: Partial products accumulation

After the n/k partial products are generated, they must be accumulated to obtain the final product. Using carry propagate adders, the time consuming carry propagate addition is repeated $n/k - 1$ times. The most commonly used method is carry save addition. In carry save addition, the carry propagation is done in the last step while in all other intermediate steps a sum and carry are generated for each bit position. [8]

The basic element used in reducing partial products is the Carry Save Adder (CSA). This is a binary full adder that takes 3 bits of the same weight as inputs and produces a sum bit and a carry bit (of one bit higher weight). Sometimes the [3:2] CSA is called a counter. Figure 2.12 shows a 1-bit CSA implementation and the addition of three n-bit partial products using CSAs.

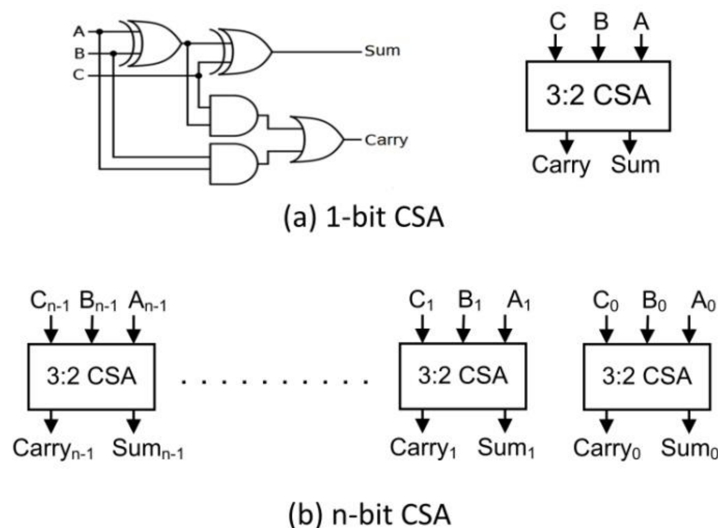


Figure 2.12 Carry save Adder (counter)

The compressor is a special form of [3:2] CSA or counter. It is designed to support regular tree implementation. The most common compressor is the [4:2]. The advantage of compressors is in their regularity. Figure 2.13 shows a [4:2] and [7:2] compressors implementation using counters ([3:2] CSAs). [7]

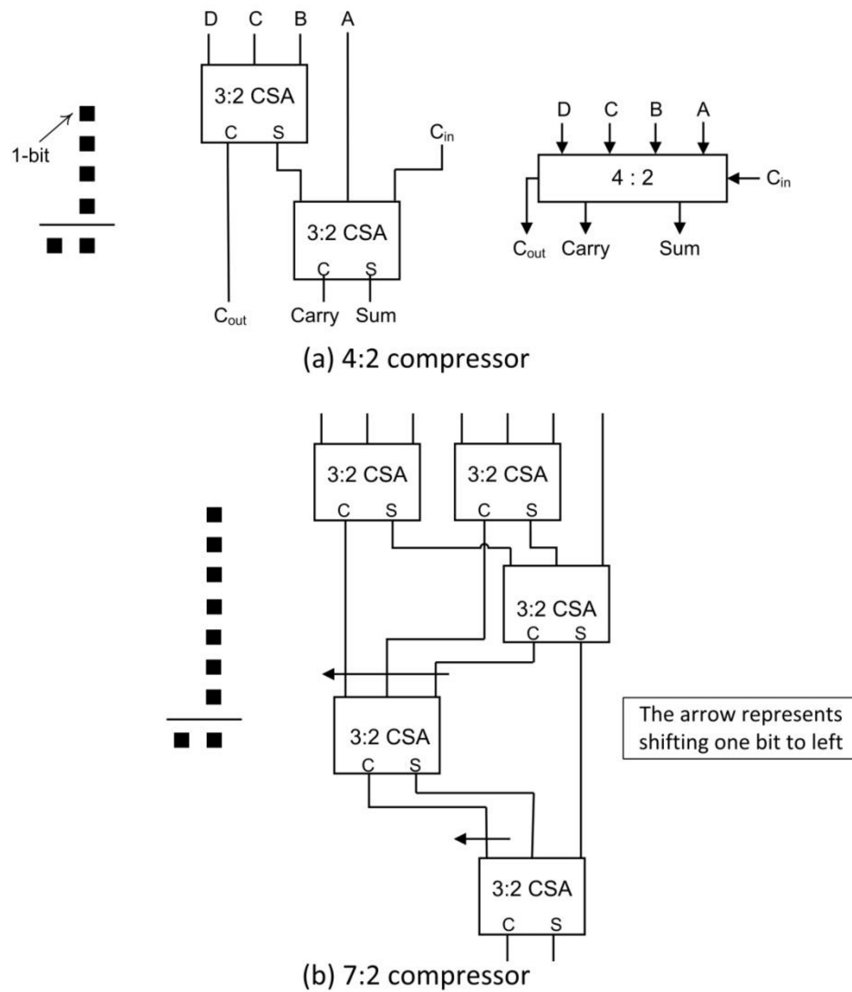


Figure 2.13 CSA Compressor

The adders used in the partial products accumulation can be connected in several approaches. In the next section these approaches are discussed.

2.2.3 Adders connection approaches

The implementation of the partial products accumulation is done using some variation of a carry save adders. These CSAs can be connected by different methods called topology. The topologies are classified into **regular** and **irregular** according to the way the counters are interconnected, and the wires required to connect the counters. In a regular topology, the CSAs are connected in a regular pattern that is replicated. The regular connections make the design of the partial product array a hierarchical design. In contrast, in an irregular topology, the CSAs are connected in order to minimize the delay, disregarding the ease of laying out the multiplier [8].

Regular topology

The regular topology is most commonly used, since it provides a compromise between optimization and design effort. The regularity allows designers to build a small group of building blocks that contain connected counters and compressors and then connect these blocks to form the topology. The delay of this topology is defined as the maximum number of counters and compressors connected in series. Regular topologies can be classified as either **array** or **tree topology**. [8]

Regular array topology

In an array, the counters and compressors are connected serially in an identical manner [8]. Figure 2.14 shows the addition of 8 partial products in an array topology. [8]

It is the slowest topology but it is very regular in its structure and uses short wires. Thus, it has a very simple and efficient layout in VLSI. Furthermore, it can be easily and efficiently pipelined by inserting latches after every CSA or after every few rows. [19]

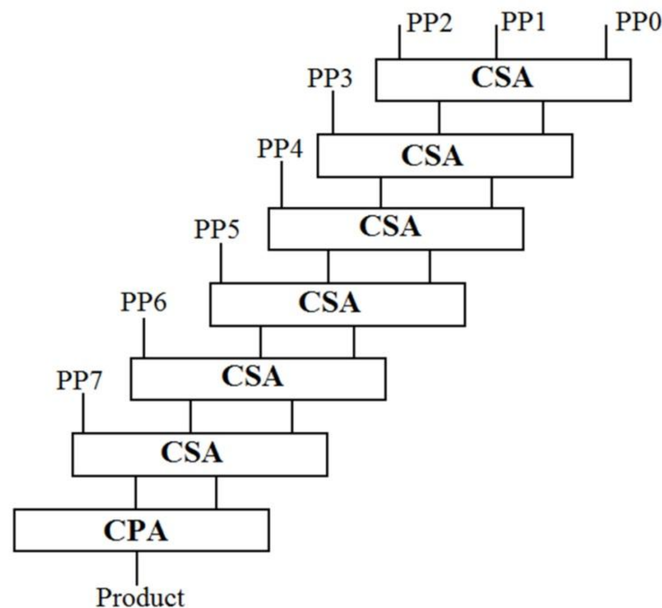


Figure 2.14 Addition of 8 partial products in an array topology using CSAs and CPA at the end

Regular tree topology

To reduce the number of adder's level, a tree is used. In a tree, counters and compressors are connected in parallel. Although trees are faster than arrays, they both use approximately the same number of counters and compressors, same area, to accumulate the partial products. The difference is in the interconnections between the adders. [8]

Trees are either regular or irregular. Regular trees have an easy structure for summing partial products and their delay is a known function of number of partial products. While irregular trees connected in order to minimize the total delay and their delay is determined by design layout.

Regular topologies allow a multiplier to be structured from building blocks where the interconnections between the adders are in a consistent pattern as shown in Figure 2.15(a) [8] and Figure 2.15(b) [5]. A CPA or a [4:2]

compressors are used which has a fast, symmetric and regular design. If the number of partial products is m, the number of CPA or [4:2] compressor levels is $log_2(m/2)$ plus one level CPA. [5]

It must be noted that the [4:2] compressor delay is approximately equivalent to two CSA levels and the CPA delay equivalent to n CSA levels for ripple carry or $\log n$ for carry lookahead, where n is the number of partial product bits.

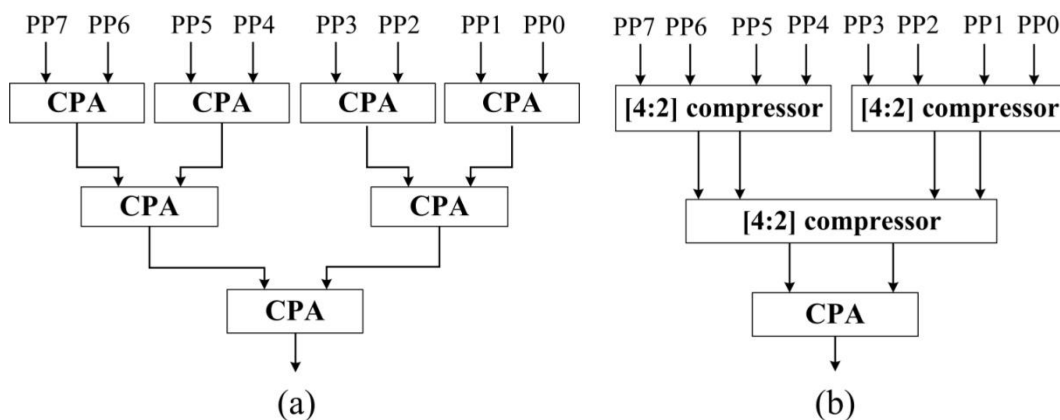


Figure 2.15 Regular tree (a) Using CPAs (b) Using [4:2] compressors

Irregular topology

Irregular topologies connect the counters and compressors in order to minimize the total delay but the design and layout is more difficult because they do not have a regular pattern for connection. Wallace tree and Dadda tree are examples for irregular trees. Wallace tree reduces the partial products by rows as array and regular tree while Dadda tree reduces the partial products by columns. [18]

Wallace tree

The Wallace tree combines partial product bits at the earliest opportunity which leads to the fastest possible design. If the number of partial products is m, the number of [3:2] CSA levels is approximately $log_{3/2}(m/2)$ plus one level CPA. Figure 2.16 shows [3:2] adder tree. [5]

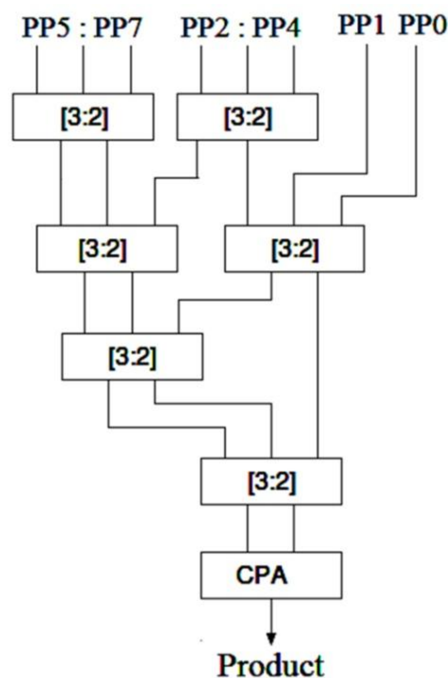


Figure 2.16 Irregular tree topology using [3:2] CSAs and CPA for last level

Dadda's tree (Reduction by column)

All previous adder topologies use the reduction by row scheme in partial product accumulation. Dadda's tree uses reduction by column scheme. Dadda Tree Combine as late as possible, while keeping the critical path length (number of levels) of the tree minimal which leads to simpler CSA tree structure, but wider CPA at the end. [12]

Figure 2.17 shows a comparison between wallace and dadda trees for four 4-bit partial products. [19]

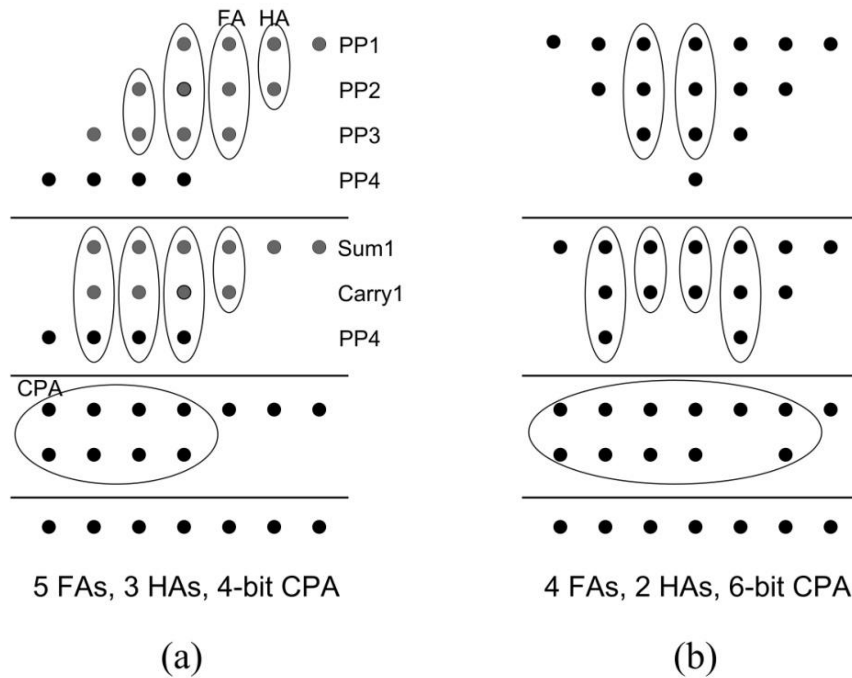


Figure 2.17 Addition of four 4-bit partial products
(a) using Wallace tree (b) using Dadda tree

2.3 "Composition of smaller multipliers" multiplication algorithm

Another way to multiply two numbers is to divide the multiplication operation into small similar multiplication operations. For example, the 8bit × 8bit multiplier can be implemented using four 4bit × 4bit multipliers, as shown in Figure 2.18. [7]

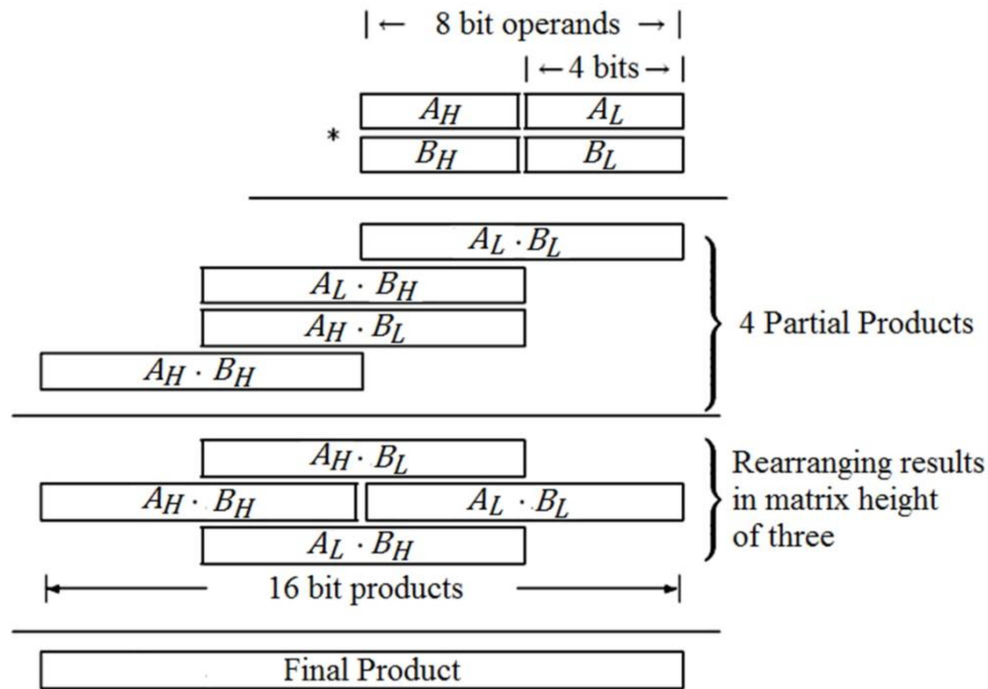


Figure 2.18 Implementation of 8 × 8 multiplier using four 4 x 4 multipliers

Generally, consider an $m \times m$ multiplier used to implement $2m \times 2m$. Denoting the high and low halves of the multiplicand and multiplier by A_H, A_L and B_H, B_L respectively. four $m \times m$ multipliers are used to compute the four partial products $A_L.B_L$, $A_L.B_H$, $A_H.B_L$, and $A_H.B_H$. These four values must then be added to obtain the final product. By rearranging the non-overlapping partial products, only three values need to be added as shown in Figure 2.18. So the $2m \times 2m$ multiplication problem has been reduced to four $m \times m$ multiplication and three operand addition problem. The $m \times m$ multiplication can be performed by smaller hardware multipliers or via lookup table, for example, a 8-bit × 8-bit multiplication can be implemented using four $2^8 \times 8$ ROMs (as lookup table), where each ROM performs 4×4 multiplication. The three partial products can be computed using single level of carry save adder, followed by a carry propagate adder. [19]

Larger multipliers, such as $3m \times 3m$ or $4m \times 4m$, can be similarly implemented from $m \times m$ multiplier building blocks. A generalization of this scheme is shown in Figure 2.19 for various multiplier arrays up to 64×64 multiplier. Each rectangle represents a 8-bit partial product as result of 4×4 multiplier. Assuming $m = 4$, it can be seen that the $4m \times 4m$ multiplication leads to seven partial products to be added, and $8m \times 8m$ multiplication produces fifteen partial products. [7]

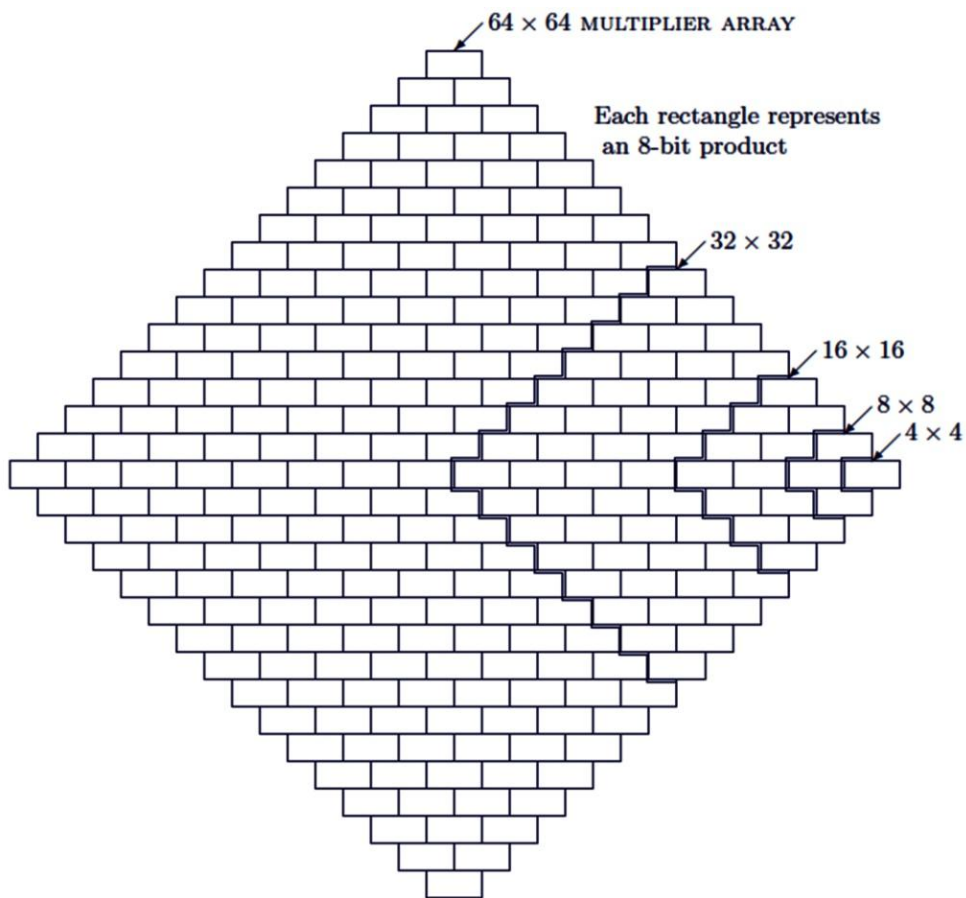


Figure 2.19 Using 4 × 4 multiplier with 8-bit product for various multiplier arrays up to 64 × 64

Table 5.3 summarizes the partial products matrix for various multipliers using 4×4 multiplier and 8×8 multiplier. It can be seen that the number of partial products decreases when using 8×8 multiplier as a basic building block. When using a lookup table basic block, the delay of the two schemes is the same but the 8×8 multiplier has double the area of the 4×4 multiplier. When using hardware multipliers basic block, the area and delay of the 8×8 multiplier are higher than the 4×4 multiplier. [7]

Basic building block	**Number of PPs (general formula)**	**Number of Partial Products(PPs)**							
		Number of bits							
		8	16	24	32	40	48	56	64
1×1 multiplier	N	8	16	24	32	40	48	56	64
4×4 multiplier	(n/2) – 1	3	7	11	15	19	23	27	31
8×8 multiplier	(n/4) – 1	1	3	4	7	9	11	13	15

Table 2.3 Summary of number of partial products for various multipliers using small multiplier where *n* is the operand size.

2.4 Bit/Digit serial multiplication algorithm

Serial arithmetic has the advantages of its smaller area and reduced wire length. In fact, the compactness of the design may allow us to run a serial multiplier at a clock rate high enough to make the unit almost competitive with much more complex designs with regard to speed. In addition, in certain application contexts inputs are supplied serially anyway. In such a case, using a parallel multiplier would be quite wasteful, since the parallelism may not lead to any speed benefit. Furthermore, in applications that call for a large number of independent multiplications, multiple serial multipliers may be more cost effective than a complex highly pipelined unit. [19]

A serial multiplier can be defined as a serial input/output pipelined sequential add-and-shift multiplier.

Bit/Digit serial multipliers can be designed as synchronous arrays of processing elements. Figure 2.20 shows a 4 × 4 bit serial multiplier. The multiplicand A is supplied in parallel from above and the multiplier B is supplied bit-serially from the right, with its least significant bit arriving first. Each bit b_i of the multiplier is multiplied by A and the result added to the cumulative partial product, kept in carry save form in the carry and sum latches. The carry bit stays in its current position, while the sum bit is passed on to the neighbouring cell on the right. This corresponds to shifting the partial product to the right before the next addition step. Bits of the result emerge serially from the right as they become available. [19]

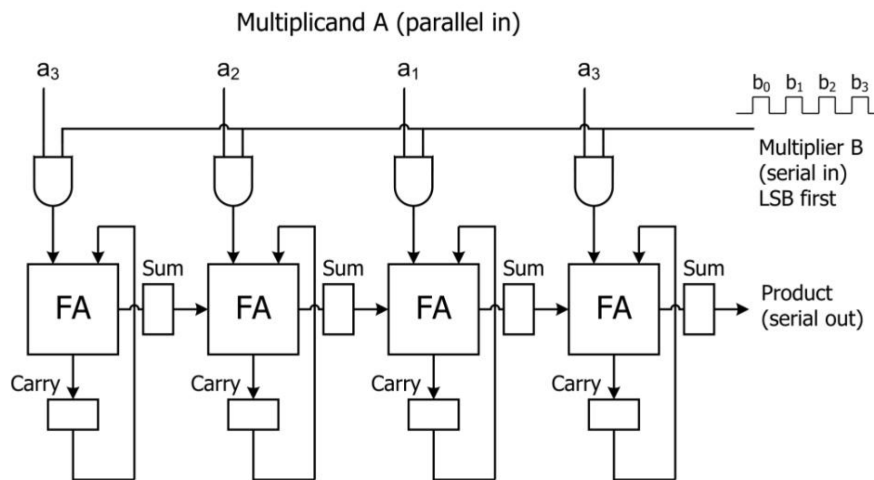


Figure 2.20 4×4 Bit Serial Multiplier

Figure 2.21 shows a digit serial multiplier. Multiplicand multiples generator is used to generate all possible multiplicand multiples of the multiplier digit (i.e. from A to $3A$ for 2-bit digit, from A to $15A$ for 4-bit digit, and so on). The suitable multiplicand multiple is selected according to multiplier digit using a

selector. Then the partial product is added and shifted to the right before the next multiplier digit is serially supplied. Digits of the result emerge serially from the right as they become available.

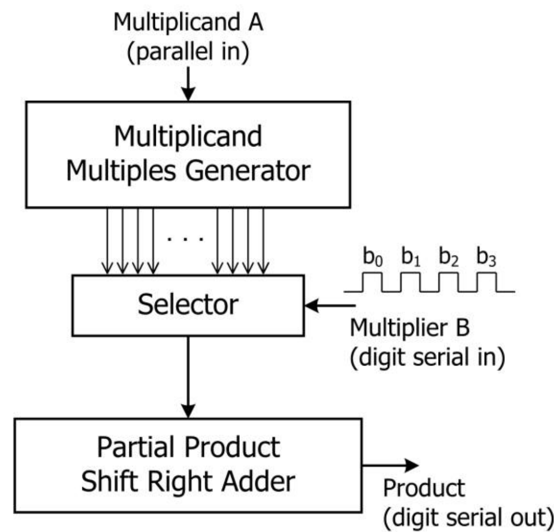


Figure 2.21 Digit Serial Multiplier.

2.5 Booth multiplication algorithm

Booth algorithm gives a procedure for multiplying binary integers in unsigned or signed two's complement representation.

2.5.1 Original Booth algorithm

The original Booth algorithm used for binary multiplication allows the multiplication operation to skip over any continuous string of all 1's and all 0's in the multiplier, rather than form a partial product to each bit. Skipping a string of 0's is straightforward, but in skipping over a string of 1's the following property is put to use: a string of k 1's is the same as 1 followed by k 0's less 1. [7]

How it works?

Consider a positive multiplier consisting of a block of 1s surrounded by 0s. For example, the product of a multiplicand A by a multiplier 00111110 is given by:

$$A \times (00111110)_2 = A \times (2^5 + 2^4 + 2^3 + 2^3 + 2^1) = A \times 62$$

The number of addition operations can be reduced to two by rewriting the same as

$$A \times (0100000\bar{1}0)_2 = A \times (2^6 - 2^1) = A \times 62$$

where $\bar{1}$ means negative 1. In fact, it can be shown that any sequence of 1's in a binary number can be broken into the difference of two binary numbers as

$$(\cdots 0 \overbrace{1 \cdots 1}^{k} 0 \cdots)_2 \equiv (\cdots 1 \overbrace{0 \cdots 0}^{k} \cdots)_2 - (\cdots 0 \overbrace{0 \cdots 1}^{k} 0 \cdots)_2$$

The multiplier is divided into substrings of 2 bits, with adjacent groups sharing a common bit. Table 2.4 shows the Original booth recoding scheme.

Bit		Meaning	Operation
2^0	2^{-1}		
b_i	b_{i-1}		
0	0	no string	0
0	1	end of string	+ A
1	0	beginning of string	− A
1	1	center of string	0

Table 2.4 Original booth recoding scheme

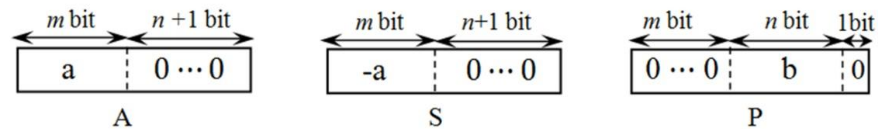
This works for negative multipliers as well. When the ones in a multiplier are grouped into long blocks, Booth algorithm performs fewer additions and subtractions than the normal multiplication algorithm.

Procedure

Let a and b be the multiplicand and multiplier, respectively. S is the negative value of A and P is the product. And let m and n represent the number of bits in the multiplicand and multiplier.

1. Set the values of A and S, and the initial value of P. All of these numbers should have a length equal to $(m + n + 1)$.

    a. A: Fill the most significant bits with the value of a. Fill the remaining $(n + 1)$ bits with zeros.
    b. S: Fill the most significant bits with the value of $(-a)$ in two's complement notation. Fill the remaining $(n + 1)$ bits with zeros.
    c. P: Fill the most significant m bits with zeros. To the right of this, append the value of b. Fill the least significant bit with a zero.



2. Determine the two most significant bits of P.

    a. If they are 01, find the value of $P + A$. Ignore any overflow.
    b. If they are 10, find the value of $P + S$. Ignore any overflow.
    c. If they are 00, do nothing.
    d. If they are 11, do nothing.

3. Arithmetically shift the value of P obtained in the 2nd step by a single place to the right. Let P now equal this new value.

4. Repeat steps 2 and 3 until they have been done n times.
5. Drop the least significant bit from P. The value of P is the product of $a \times b$.

Original Booth algorithm can be summarized as performing an addition when it encounters the first digit of a block of ones (0 1) and a subtraction when it encounters the end of a block of ones (1 0). Also an extra bit can be added to the left of A, S, and P, to represent the multiplicand if it has the largest negative number (i.e. if the multiplicand has 8 bits then this value is -128).

The disadvantages of this algorithm are that it generates a varying (at most n) number of partial products, depending on the bit pattern of the multiplier. The extreme, worst case, occurs when the multiplier is alternating between 1's and 0's. The number of addition or subtraction process is n instead of $n/2$ for add-and-shift algorithm. Of course, hardware implementation lends itself only to a fixed independent number of partial products.

Booth algorithm can be designed using sequential approach, as mentioned above. Or parallel approach, by recoding every two side by side bits to multiplicand or its negative value or zero.

2.5.2 Modified Booth algorithm

The modified version of Booth algorithm is more commonly used. The difference between the Booth and the modified Booth algorithm is that the modified booth always generates a fixed number of partial products. It encodes every k bit of multiplier into one partial product. So for n bit multiplier, it introduces n/k partial products as the high radix multiplication. Several versions of modified booth algorithm are introduced depending on the value of k. As the value of k is increased, the number of partial products decreases but the number of hard multiplicand multiples required to generate increases. [7]

Booth radix 4 recoding scheme, Booth 2

The modified Booth 2 multiplier encoding scheme encodes every 2-bit groups of multiplier. For an 8-bit multiplier, it produces four partial products for a signed multiplier, as the most significant input bit represents the sign, or five partial products for an unsigned multiplier number. The multiplier is divided into substrings of 3 bits, with adjacent groups sharing a common bit. It requires that the multiplier be padded with a 0 to the right, for unsigned or positive numbers, and with 1 to the right for negative numbers, in two's complement representation. Also it is padded with one or two zeros to the left. Table 2.5 is the encoding table of the eight possible combinations of the three multiplier bits. [7]

Bit			Operation
2^1	2^0	2^{-1}	
b_{i+1}	b_i	b_{i-1}	
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	−2A
1	0	1	−A
1	1	0	−A
1	1	1	0

Table 2.5 Booth radix 4 recoding scheme

By inspection of the table, only one action (addition or subtraction) is required for each two multiplier bits. Thus, the use of the algorithm insures that for an odd number of multiplier bits, only $n/2$ actions will be required for any multiplier bit pattern where the last action will be defined by $0.Y_{n-1}.Y_{n-2}$ for

unsigned numbers. And for an even number of multiplier bits, $n/2 + 1$ actions are required, the last action being defined by $0.0.Y_{n-1}$ for unsigned numbers. [7]

Figure 2.22 shows a 16 bit × 16 bit multiplication using Booth 2 algorithm [8].

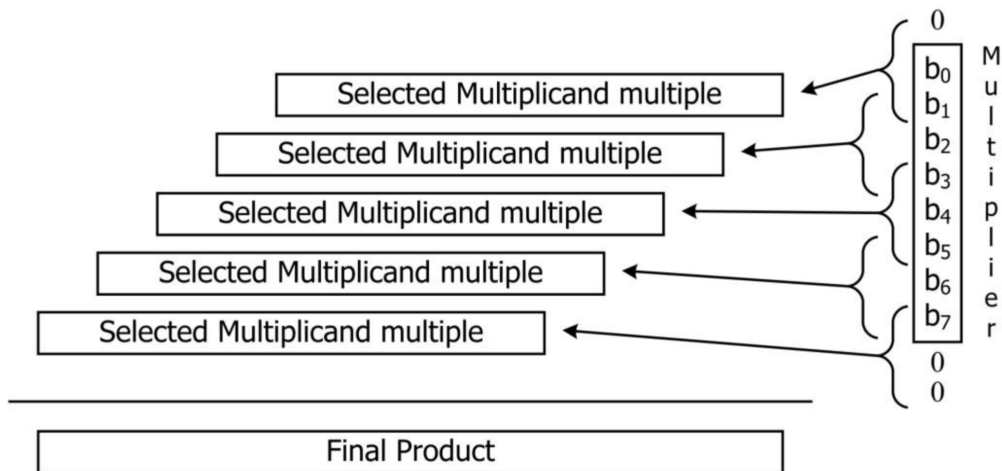


Figure 2.22 16 bit Booth 2 multiply

For example [7], suppose a multiplicand (A) is to be multiplied by an unsigned multiplier $B = (11101011)_2$ which is equivalent to decimal 235. When using modified Booth 2 algorithm. The multiplier must be decomposed into overlapping 3-bit segments and actions determined for each segment. Note that the first segment has an implied "0" to the right of the binary point. Thus, we can label each segment as follows:

(5) (3) (1)
0 0 1 1 1 0 1 0 1 1 .0
(4) (2)

while segment (1) is referenced to the original binary point, segment (2) is four times more significant. Thus, any segment (2) action on the multiplicand must be scaled by a factor of four. Similarly, segment (3) is four times more

significant than (2), and 16 times more significant than (1). Now, by using the table and scaling as appropriate, we get the following actions:

Segment number	Bits	Action	Scale factor	Result
(1)	110	−A	1	−A
(2)	101	−A	4	−4A
(3)	101	−A	16	−16A
(4)	111	0	64	0
(5)	001	+A	256	+256A
Total action				235A

The actions specified in the table are independent of one another so the five result actions can be summed in parallel using carry save adders.

Booth radix 8 recoding scheme, Booth 3

It is an extension of the modified Booth algorithm which involves an encoding of three bits at a time while examining four multiplier bits. This scheme would generate only $n/3$ partial products. [7]

However, its encoding requires the generation of $3A$ [8], which is not as trivial as generating $2A$. Thus, most hardware implementations use only Booth 2 scheme.

2.6 Conclusion

The multiplication operation can be designed in sequential or combinational approach. Sequential multipliers have less area while combinational multipliers have lower latency.

Also the multiplication is done in radix-2 or high radix scheme. Radix-2 lead to small area design nevertheless high radix has the advantage of low latency.

Four techniques are used for the multiplier implementation, add-and-shift, composition of small multipliers, bit/digit serial, and booth multiplication. Add-and-shift is the most common and simplest method of multiplication. It can be implemented sequentially or combinational in radix-2 or high radix. Composition of smaller multipliers has the same area and speed of complete multiplier. It only divides the large components to small ones and rearranges them. Bit/Digit serial multiplier (radix-2/high radix serial multiplier) has small area, but larger than sequential add-and-shift, high latency, high throughput, and can be easily pipelined. It is useful in applications where the inputs are supplied serially anyway. Original booth tries to decrease the number of partial product. It has a worst case number of partial products $n/2$, but it generates a variable number of partial products. Modified booth generates a fixed number of partial products n/k, the same as high radix add-and-shift. Booth multiplication can be implemented sequentially or combinationaly.

Chapter 3

Decimal Multipliers

The decimal multiplication is more complex than the binary multiplication. Since the multiplier 4-bit digit takes values between 0 and 9. Let the multiplicand A and the multiplier B be two signed numbers represented as sign and an n-digit magnitude. The multiplication $P = A \times B$ will create a sign and 2n-digit product P. The multiplication operation is described as:

$$\mathrm{sign}(P) = \mathrm{sign}(A) \cdot \mathrm{sign}(B) \quad (3.1)$$

$$|P| = |A|.|B| \quad (3.2)$$

The sign of the product is implemented using XOR gate. And the magnitude of the product is implemented using some algorithms similar to binary algorithms but here we deal with digits instead of bits. For example in 2846×3715, we assume 3715 a multiplier and 2846 a multiplicand and assume that we have all multiplicand multiples (2×2846, 3×2846, 4×2846, ……, 9×2846). The digits in the multiplier are examined one at a time and the suitable multiplicand multiple is selected according to the multiplier digit. A

number of shifted multiples are added according to multiplier digit position as shown in Figure 3.1 to form the final product.

				2	8	4	6
				3	7	1	5
				5 × 2	8	4	6
			1 × 2	8	4	6	
		7 × 2	8	4	6		
	3 × 2	8	4	6			
1	0	5	7	2	8	9	0

Figure 3.1 4-bit Decimal multiplication example

From this example we can divide the multiplication operation into three stages: Multiplicand multiples generation (from A to 9A), multiplier recoding to select the suitable multiple for each multiplier digit which generate the partial products (PPs), and partial products addition.

Decimal multipliers can be implemented using sequential or parallel approaches. Sequential multipliers have a small area compared to parallel ones. But, parallel multipliers have a significant low latency advantage over sequential multipliers. The choice between sequential and parallel approaches depends on the more important issue in the application, area or delay.

In the next sections the history of the decimal multiplication stages, which are multiplicand multiples generation, multiplier recoding for multiples selection, and partial products accumulation, are discussed.

3.1 Multiplicand Multiples Generation

Decimal multiples generation is more complex than binary multiples generation because it deals with Binary Coded Decimal (BCD) format so the left shifting of multiplicand, A, will not introduce $2A$ as in binary. The decimal multiplicand multiples are generated by successively adding the multiplicand using BCD adders[1], decimal adders, which has large area and delay, or via a lookup table for the multiplicand multiples, which has large area.

Generation of BCD_2A and BCD_5A

It can be seen that the generation of BCD_2A and BCD_5A is simpler than the other multiples where the carry propagate only to next digit [13][20]. When any BCD digit with a value from 0 to 9 is doubled, it gives two digits from 00 to 18. The least significant digit value is even, i.e. least significant bit is 0 and maximally equal 8 and the carry value is maximally 1. By adding the carry to next significant digit, the maximum value obtained is 9, this addition can be done by only putting the carry bit in the LSB of next digit. So the generation of BCD_2A for a digit a_i can be summarized as shown in Figure 3.2.

Also the BCD_5A multiple generation depends on that the fifth multiple of any digit, gives only two digits. The least significant digit value is 0 or 5 due to the input digit is even or odd, respectively. The most significant digit, carry digit, value is maximally 4 (since $9 \times 5 = 45$) so by adding the carry to next significant digit, the maximum value obtained is 9. So its carry is propagated only to next digit. The carry digit is equal to the value of input digit divided by two. Where $5x = 10x/2 = x/2$ shifted left one decimal digit. It can be implemented by right shifting the input digit and skips its carry bit. These two multiples have approximately the area of n-bit carry propagate adder and the

[1] In BCD addition, a correction of six must be added if a digit sum is greater than nine to skip over the invalid BCD digits, A_{16}-F_{16}. So the decimal CPA has a delay of O(2n) instead of O(n) for binary CPA.

delay is approximately of O(4), 4-bit carry propagation where carry propagate only to next digit.

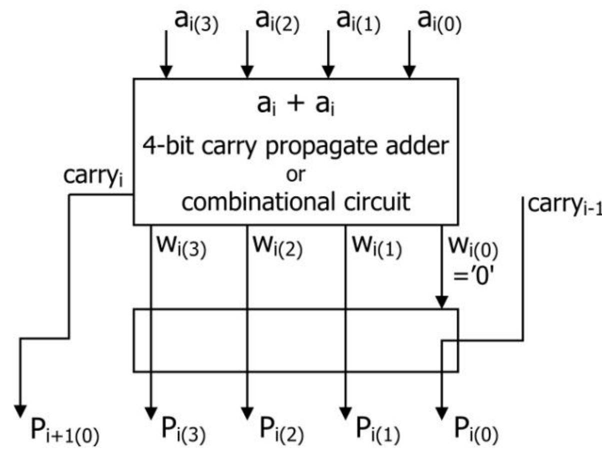


Figure 3.2 BCD multiplication by two

Multiplicand multiples generation stage

Simple decimal multipliers as those designed in the early days of decimal circuits [20]generate **all decimal multiplicand multiples**, from A to $9A$, and store them in registers before the start of the algorithm as shown in Figure 3.3. This technique needs a large area for the decimal carry propagate adders and for the registers needed to store the multiples. Also it has a large delay due to the $O(2n)$ delay of decimal carry propagate adders [20].

To reduce the area and delay, a **reduced set of decimal multiplicand multiples** is generated and stored in registers before the start of the algorithm then the remaining multiples are obtained dynamically during the algorithm using a decimal carry propagate adder. A secondary set or tertiary set is sometimes used. A secondary set of multiplicand multiples $(A, 2A, 3A, 4A, 8A)$ is proposed in [6](a) where only two members of the set are need to be added to generate missing multiples. It need only one decimal carry propagate adder of

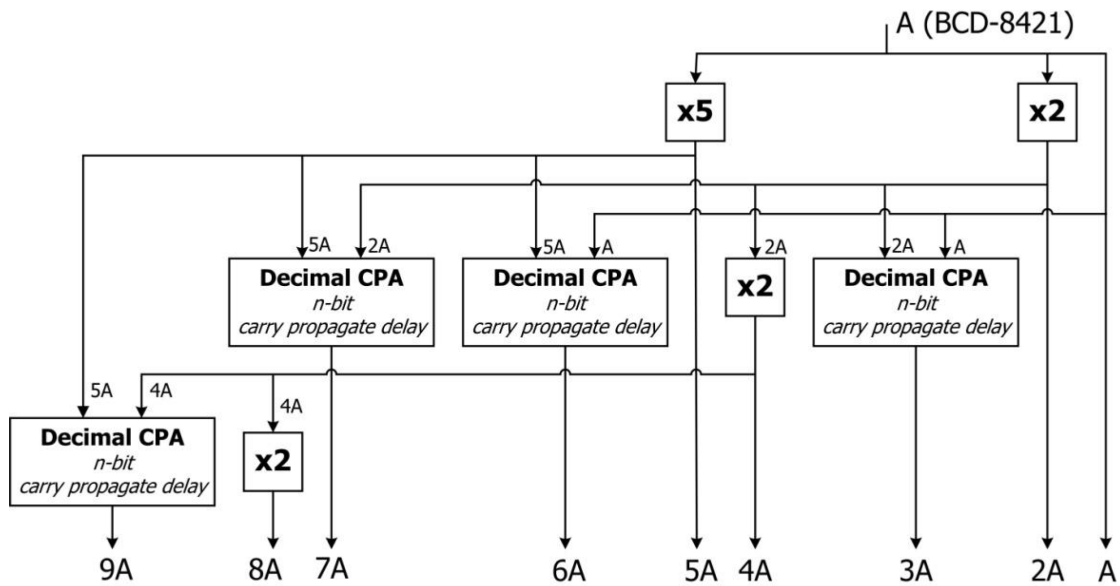


Figure 3.3 Multiplicand multiples generation (generate all multiplicand multiples)

$O(2n)$ for the $3A$ multiple generation, Figure 3.4(a). Also a tertiary set of multiples $(A, 2A, 4A, 8A)$ is proposed where at most three members of the set are added to generate missing multiples. The $(A, 2A, 4A, 8A)$ set does not need to generate the multiple $3A$ but it requires an extra addition for the generation of the missing multiple $7A = 1A + 2A + 4A$. The extra adder can be a decimal carry save adder which has less delay, $O(1)$, than the decimal carry propagate adder, Figure 3.4(b).

Another secondary multiplicand multiples set $(A, 2A, 4A, 5A)$ is introduced in [20][6](a)][10]. This set is generated faster than the previous sets. This set reduces the delay of the multiplicand multiples generation stage, Figure 3.4(c).

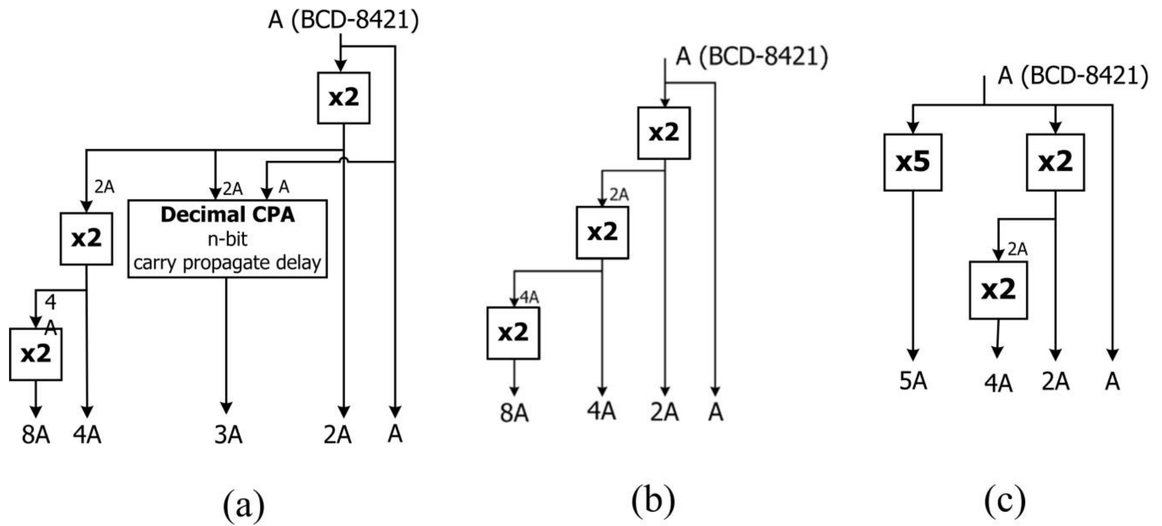


Figure 3.4 Decimal multiplicand multiples generation sets

For more reduction of the area and delay for the pre-calculated multiples, a signed digit recoding technique is proposed by Lang and Nannarelli in [13]. They generate the secondary set $(A, 2A, 5A, 10A)$ of multiples. The two groups $(0, 5A, 10A)$ and $(-2A, -A, 0, A, 2A)$ are used to generate the missing multiples, Figure 3.5. Using the $(\text{invert} \,\&\, \mathrm{C_{in}} = 1)$ block for signed digit recoding can convert the tertiary set $(A, 2A, 4A, 8A)$ to secondary set by generating the missing multiple $7A = 8A - A$.

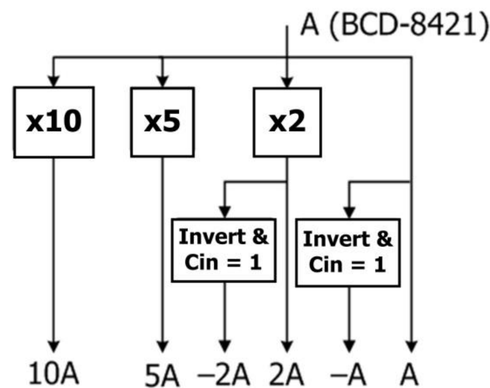


Figure 3.5 Signed digit recoding by Tomás Lang and Alberto Nannarelli

Erle et al. in [6](b) propose a different recoding technique for an efficient generation of partial products. He depends on digit-by-digit multiplication not word-by-digit as before. He recodes the two multiplication operands, multiplicand and multiplier, into signed digits from -5 to 5 to simplify the partial products generation process. And since the magnitude of product is independent on the sign of operands and the multiplication by zero and one can be done using multiplexer, the range of multiplied digits is reduced to [2→5] x [2→5]. Thereby he has only 10 different combinations of inputs to be multiplied. He shows the complexity of the digit-by-digit products for different ranges of decimal inputs (Table 3.1).

range of inputs	input combinations	unique products
[0→9] x[0→9]	100	37
[1→9] x[1→9]	81	36
[2→9] x[2→9]	64	30
[0→5] x[0→5]	36	15
[1→5] x[1→5]	25	14
[2→5] x[2→5]	16	10

Table 3.1 Complexity of digit-by-digit products for different ranges of decimal inputs

Figure 3.6 shows the block diagram of a digit multiplier block, where the superscript S indicates that the result of the recoding is a signed-magnitude digit, the superscript T indicates that the sub-function output is realized via a lookup table or a combinational circuit structure, and the superscript O indicates that the partial product is in an overlapped form since each digit multiplier block yields two digits.

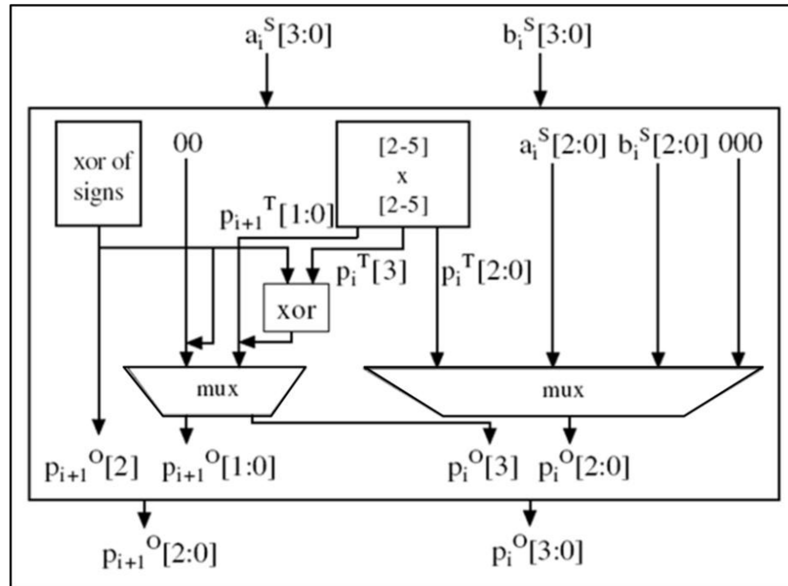


Figure 3.6 signed digit-by-digit multiplier block

The partial products are generated using a digit-by-digit multiplier on a word by digit basis, first in a signed digits form with two digits per position, Table 3.2, and then combined via a combinational circuit. Although the least significant digit has a negative sign in some instances, the most significant digit is always positive, and thus the two-digit product is a positive value. The signed digit partial products are developed one at a time while passing through the recoded multiplier operand from the LSD to the MSD in sequential form, and then each partial product is added along with the accumulated sum of previous partial products via a signed digit decimal adder.

	2	**3**	**4**	**5**
2	04	$1\bar{4}$	$1\bar{2}$	10
3	$1\bar{4}$	$1\bar{1}$	12	15
4	$1\bar{2}$	12	$2\bar{4}$	20
5	10	15	20	25

Table 3.2 Signed digit-by-digit products

The partial products generation process for a sequential multiplier design using this method takes n + 1 cycle, also its generation for a combinational design takes ten logic levels delay to convert the overlapped partial products form to non-overlapped form and recode them in a manner appropriate for the signed digit decimal adder.

Most of previous multiplicand multiples generation methods have a considerable delay because of the decimal correction stage in adders which increases the total delay of the multiplication operation. However, Vázquez et al. in [22] propose a new different signed digit, SD, decimal multiplicand multiples generation techniques. Firstly, they introduce three recoding schemes, SD-radix-10 which generates the secondary set $(A, 2A, 3A, 4A, 5A)$ multiples, SD-radix-5 which generates the secondary set $(A, 2A, 5A, 10A)$ multiples, and SD-radix-4 which generates the secondary set $(A, 2A, 4A, 8A)$ multiples.

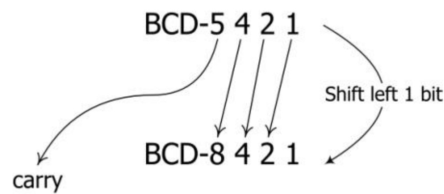
To simplify the decimal multiples generation process, different redundant BCD recoding formats are used. Table 3.3 shows various BCD coding formats such as BCD-5421, BCD-5211 and BCD-4221.

	BCD-8421	BCD-5421	BCD-5211	BCD-4221
0	0000	0000	0000	0000
1	0001	0001	0001 | 0010	0001
2	0010	0010	0100 | 0011	0010 | 0100
3	0011	0011	0101 | 0110	0011 | 0101
4	0100	0100	0111	1000 | 0110
5	0101	1000 | 0101	1000	1001 | 0111
6	0110	0110 | 1001	1010 | 1001	1100 | 1010
7	0111	0111 | 1010	1100 | 1011	1101 | 1011
8	1000	1011	1110 | 1101	1110
9	1001	1100	1111	1111

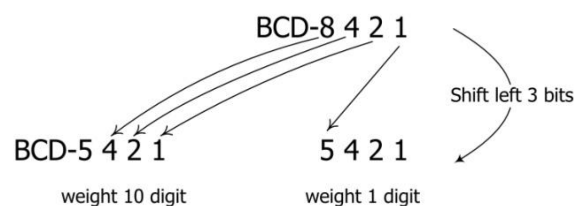
Table 3.3 BCD coding formats

These BCD formats depend on different binary bits weight. The table represents the formats as BCD-xxxx where x's is the weight of every binary bit. For example, 1111 has a value of 8+4+2+1 = 15 in BCD-8421 format, a value of 4+2+2+1 = 9 in BCD-4221 format, and a value of 5+4+2+1 =12 in BCD-5421 format.

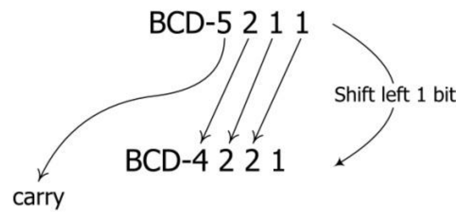
These formats allow the generation of 2A and 5A multiples in a few levels of logic gates using recoding block and wired left shifts. For example: BCD-5421 format allow a fast decimal 2A multiple generation in two steps. Firstly recode each BCD-8421 digit to BCD-5421 then left shift the recoded multiplicand by one obtaining the 2A multiple in BCD-8421.



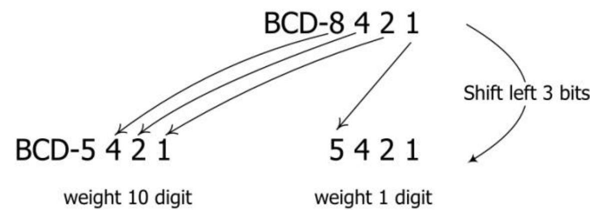
Also it ease the 5A multiple by left shifting the multiplicand A, BCD-8421, three bits then recode each digit of the shifted multiplicand from BCD-5421 to BCD-8421.



For the BCD-4221, the multiplication by two obtained by recoding each multiplicand digit to BCD-5211 then left shifts the recoded multiplicand by one, the 5A multiple in BCD-4221 format is obtained.



Also a three bit left shifting of the BCD-4221 multiplicand obtain a 5A in BCD-5211.



More multiplicand multiples generation in different BCD formats is discussed in [22].

Vázquez et al. in 123 use the BCD-4221 and BCD-5211 in decimal adders where they give a valid decimal digit values for all 16 combinations. These avoid the extra delay and area of the adders' decimal corrections. Also it allows binary addition/subtraction to be used for partial products accumulation where BCD-4221 format is self-complementing. So the addition of a negative value of the multiplicand can be obtained only using inverters and setting the carry-in bit of binary adder by 1.

3.2 Multiplier recoding for multiples selection

Multiplexers controlled by the multiplier digits are used to choose the correct multiplicand multiples to generate the partial products to be added in the next stage. If all decimal multiplicand multiples ($A \rightarrow 9A$) are generated in the previous stage, only one multiplexer, MUX, is needed. While, if a reduced set of multiplicand multiples are generated, a multiplier recoding will be needed to represent each multiplier digit into two digits. Thereby, two

MUXs are needed to choose the two suitable multiplicand multiples for each multiplier digit. Table 3.4 shows an example of multiplier digits recoding for the secondary set $(A, 2A, 4A, 8A)$ multiples.

b_i	b_i'	b_i''
0	0	0
1	1	0
2	2	0
3	1	2
4	2	2

b_i	b_i'	b_i''
5	4	1
6	4	2
7	8	-1
8	8	0
9	8	1

Table 3.4 Example of multiplier recoding

When all multiplicand multiples are generated, one partial product for each multiplier digit is selected so $n/4$ partial products are generated, where n is the number of multiplier bits. Nevertheless, when a reduced set of multiplicand multiples are generated, two partial products for each multiplier digit is selected so $n/2$ partial products are generated. It seems that the first scheme has less delay for next accumulation stage, but the generation of all decimal multiplicand multiples needs a large area and delay because it needs CPAs. The number of partial products is increased in the second scheme but it increases the delay by one level CSA only in the partial products accumulation stage, it will be discussed in next section.

3.3 Partial Products accumulation

Partial products accumulation stage in decimal multipliers can be implemented using any of high radix methods discussed in previous chapter, sequential designs, or parallel designs (array or tree topology). The only difference between decimal and high radix design is that a decimal CSAs and decimal CPAs is used. Decimal adders are like binary adders except an extra

correction block after each digit addition. A correction of six is added if a digit sum is greater than nine to skip over the invalid BCD digits, $A_{16} - F_{16}$.

3.3.1 Sequential accumulation approach

Several sequential decimal multipliers are proposed in [6](a)[14][13][6](b) [10][20]. The basic sequential approach of decimal multiplication is to iterate over the digits of the multiplier B and based on the value of the current digit, successively add multiples of the multiplicand A to a product register called intermediate product (IP) with shifting one digit in each iteration.

When one partial product is generated for each multiplier digit [6][20], the equation for the sequential partial products accumulation is as follows

$$P_{i+1} = (P_i + A \cdot b_i) \cdot 10^{-1} \qquad (3.3)$$

where $P_0 = 0$ and $0 \leq i \leq n/4 - 1$. And after $n/4$ iterations, P_n corresponds to the final product P.

To implement this, a decimal CPA is used as in Figure 3.7a. However, when two partial products are generated for each multiplier digit (secondary set of multiplicand), a decimal CSA is used before the decimal CPA [10] (see Figure 3.7b) and the following equation is used

$$P_i + 1 = (P_i + A \cdot b_i' + A \cdot b_i'') \cdot 10^{-1} \qquad (3.4)$$

where $A \cdot b_i' + A \cdot b_i'' = A \cdot b_i$. Although the secondary multiple approach reduces the delay and area of partial products generation (i.e. the area and delay of the decimal CPA), it introduces a delay overhead in the extra decimal CSA in each iteration.

Using decimal CSAs only in the iterations has a significant delay reduction as in Figure 3.8. The carry save adders are used in iterations then one decimal CPA is used at the end of process to add the sum and carry outputs of the decimal CSA from last iteration [15][16][6](b).

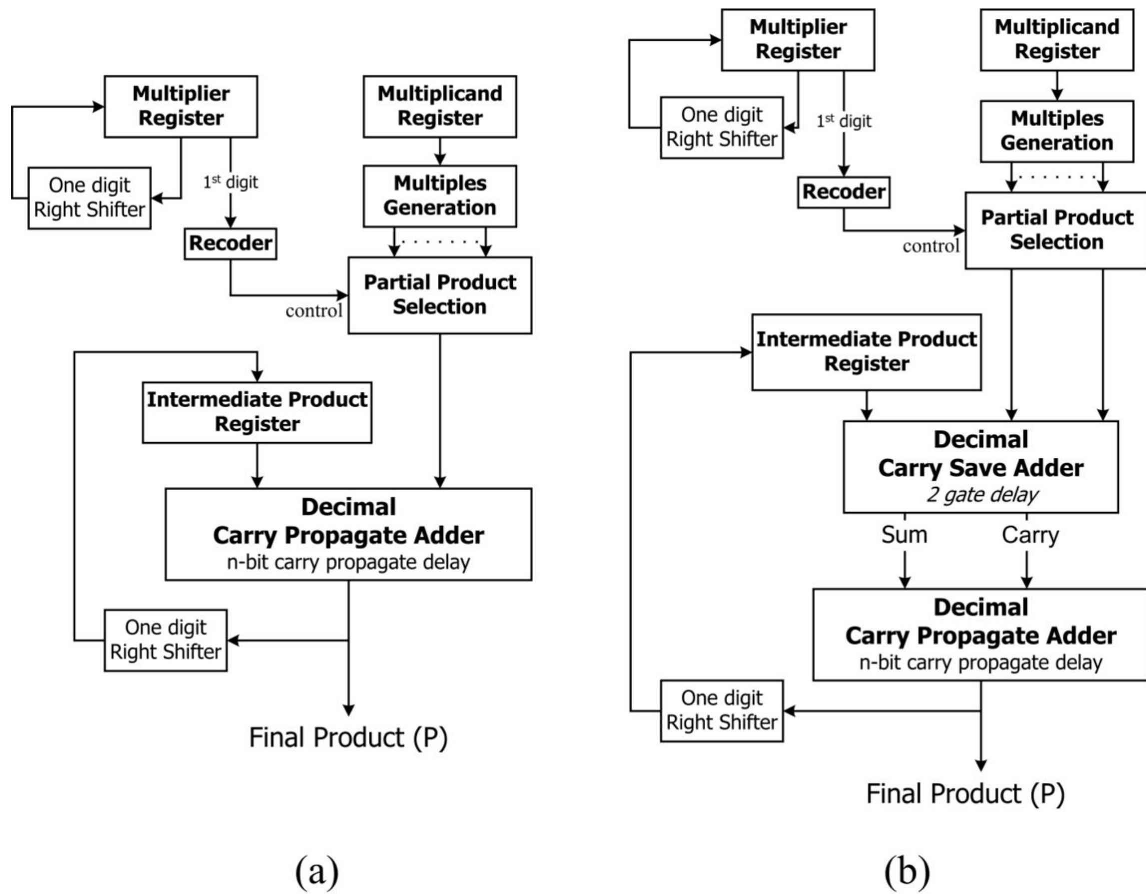


Figure 3.7 Sequential Decimal Multiplication Design

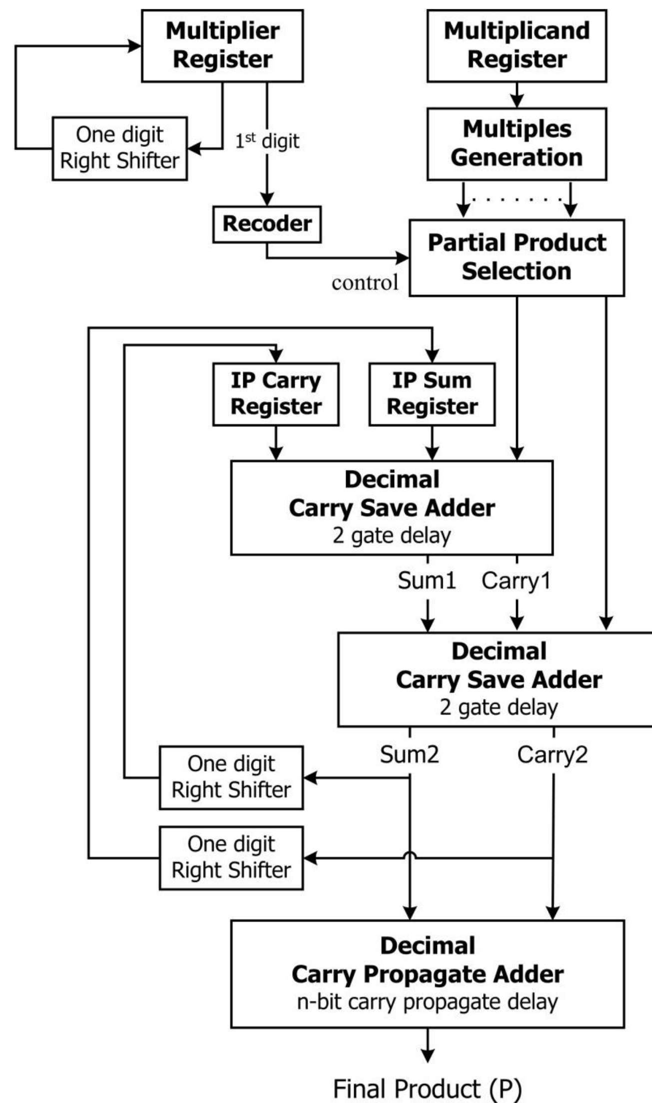


Figure 3.8 Sequential Decimal Multiplication Design

3.3.2 Parallel accumulation approach

Parallel decimal multiplication offers a good delay reduction with an increase in area [13][22]. In parallel decimal multiplication, all partial products are generated in parallel according to the multiplier digits then the partial products are accumulated using a decimal carry save adders tree. This tree reduce all partial products into two partial products then a carry propagate adder is used to obtain the final product as in Figure 3.9. The same as binary

trees, the delay of decimal CSAs tree depends on the number of input partial products and the design and arrangement of the decimal CSAs. For example, the accumulation of 16 partial products needs 6 levels of decimal CSAs.

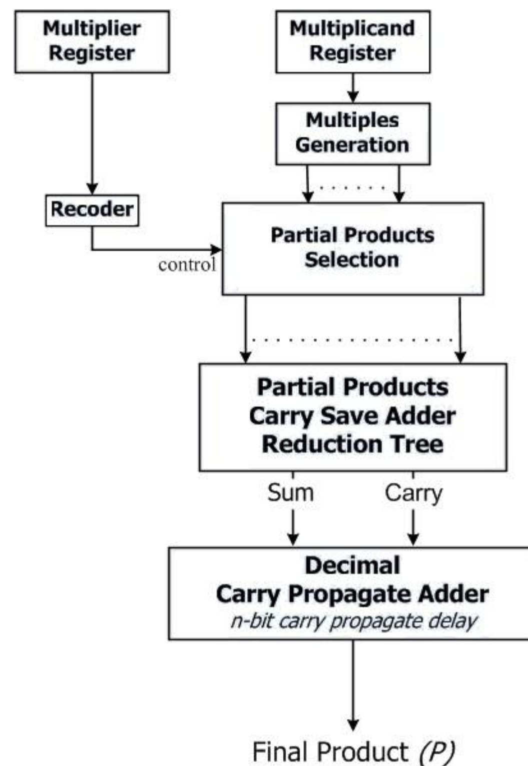


Figure 3.9 Parallel Decimal Multiplier Design

3.3.3 Decimal adder block implementation

Decimal addition can be implemented using binary adder with decimal correction block for every digit to correct the binary digits out of BCD range [15][16][17], or using direct decimal addition technique [21]. Vázquez et al. in [22] present a decimal adder using all-valid BCD formats, which have a valid decimal value for all 16 combinations. It gives a valid sum and carry values without the need of correction blocks which decrease the area and delay of the adder [22].

Binary addition with decimal correction block

The decimal addition of two decimal digits of the same order yields a two digit decimal sum in the range [0 → 18]. When using a binary adder for the addition, there are 5 bits out from the adder. The four LSBs represent the LSD of the decimal sum, and the MSB represent the carry bit or the MSD of the decimal sum. The LSD of the sum may actually range from [0 → 15], instead of BCD digit values from [0 → 9]. In the case of the LSD being in the range [10 → 15], the LSD need to be adjusted to bring it into the valid range for a decimal digit. This can be achieved by incrementing the LSD by six. Also in this case, the carry needs to be changed from zero to one to represent a carry of 10. Also, in the case of the carry is one and the LSD is in the range [0 → 2], the sum needs to be incremented by six to adjust the weight of the carry from 16 to 10. Thus, there are two different reasons for the correction, but both situations are handled by the addition of six. See [10] for more details in this type of decimal addition.

Direct decimal addition

Direct decimal addition implements logic units that accepts as inputs two 4-bit BCD digits with a 1-bit carry-in, and directly produces a 4-bit BCD sum digit and a 1-bit carry-out where the weight of carry-out bit is 10 times the weight of sum digit.

Binary addition with All-valid BCD formats

When BCD-4221 format or BCD-5211 format is used in CSA or CPA blocks, it gives a correct sum and carry values in the range of [0 → 9]. However a decimal multiplication by 2 is required before using the carry digit in the computations. The carry is multiplied by two, left shift in binary addition, using BCD-4221 or BCD-5211 as discussed in the decimal multiplicand multiples generation stage. Figure 3.10 shows a general design of

3:2 decimal CSA where A_i, B_i, and C_i are the three inputs of the CSA. S_i, and H_i are the sum and carry outputs of the CSA. W_i is the carry digit after recoding then it shifted left one bit for the multiplication by two generation.

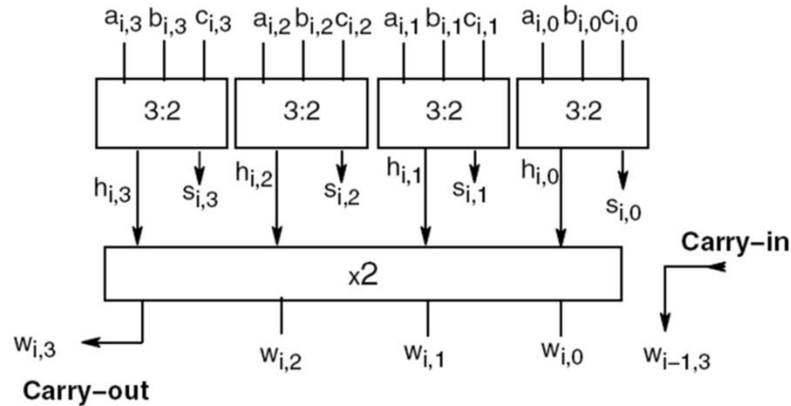


Figure 3.10 Generic design for the 3:2 decimal CSA

Figure 3.11 explain two examples for BCD-4221 and BCD-5211 decimal CSAs. The following equation describes the CSA operation: $A_i + B_i + C_i = S_i + 2H_i$. In Figure 3.11a, BCD-4221 format is used. The three input digits are added using binary CSA and give two digits sum and carry. The carry digit is recoded to BCD–5211, using combinational circuit, then left shifted one bit to obtain 2H in BCD-4221 format. However, Figure 3.11b uses BCD-5211 format. The carry digit is recoded to BCD–4221 then left shifted one bit to obtain 2H in BCD-5211 format.

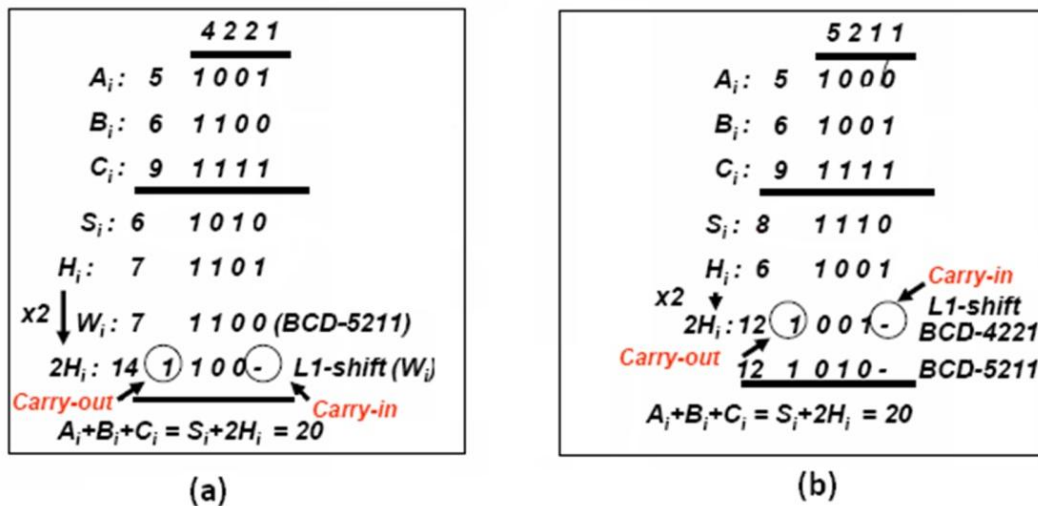


Figure 3.11 Decimal carry-save addition example (a) in BCD-4221 format (b) in BCD-5211 format

3.3.3 Decimal trees

Radix-10 CSA

Lang and Nannarelli in [13] use a radix-10 CSA tree to accumulate the partial products. Their multiplier uses SD-radix-5 for multiplicand multiples generation, which generates the (A, 2A, 5A, 10A) multiples set. This set generates two partial products for each digit and carry bit for negative. The 2 partial products and the carry bit are added using radix-10 CSA to generate 16 partial products in carry save format. Radix-10 CSA adds a carry save operand, sum and carry, plus another BCD operand to produce a carry save result, see Figure 3.12a. Also a carry counter adds an array of carry vectors of the same weight and produces a decimal digit as shown in Figure 3.12b.

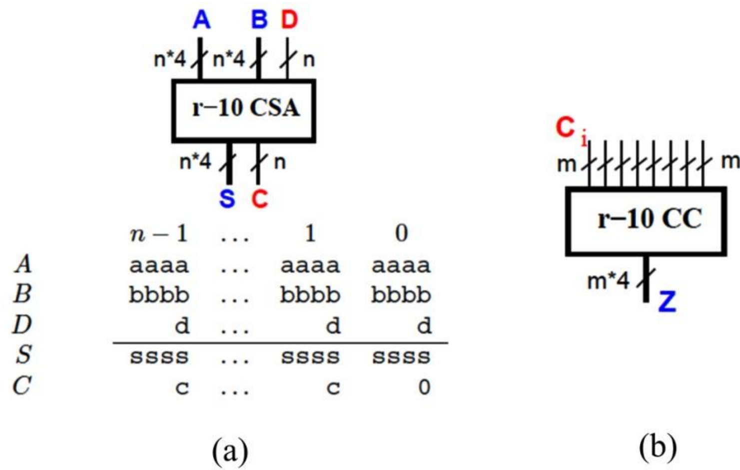


Figure 3.12 (a) n-digit radix-10 CSA (b) m-digit radix-10 counter.

For example to add 16 carry save partial products, xy_i, arranged as in Figure 3.13. The first level of the tree needs only 8 radix-10 CSAs leaving the carries of 8 partial products not accounted for. And by arranging the radix-10 carry-save adders and the carry counters as in Figure 3.14, the partial products are accumulated in a 6-level tree.

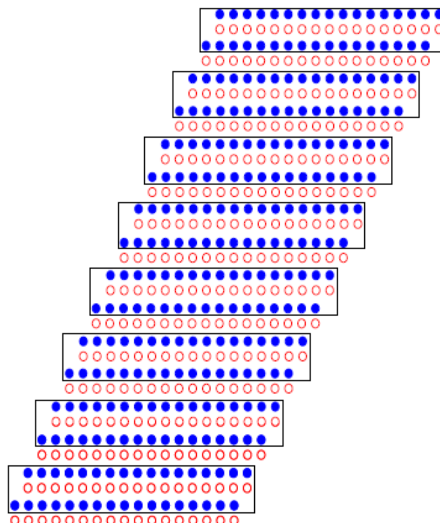


Figure 3.13 Array for partial products. Solid circles indicate BCD digits, hollow circles indicate carry bits.

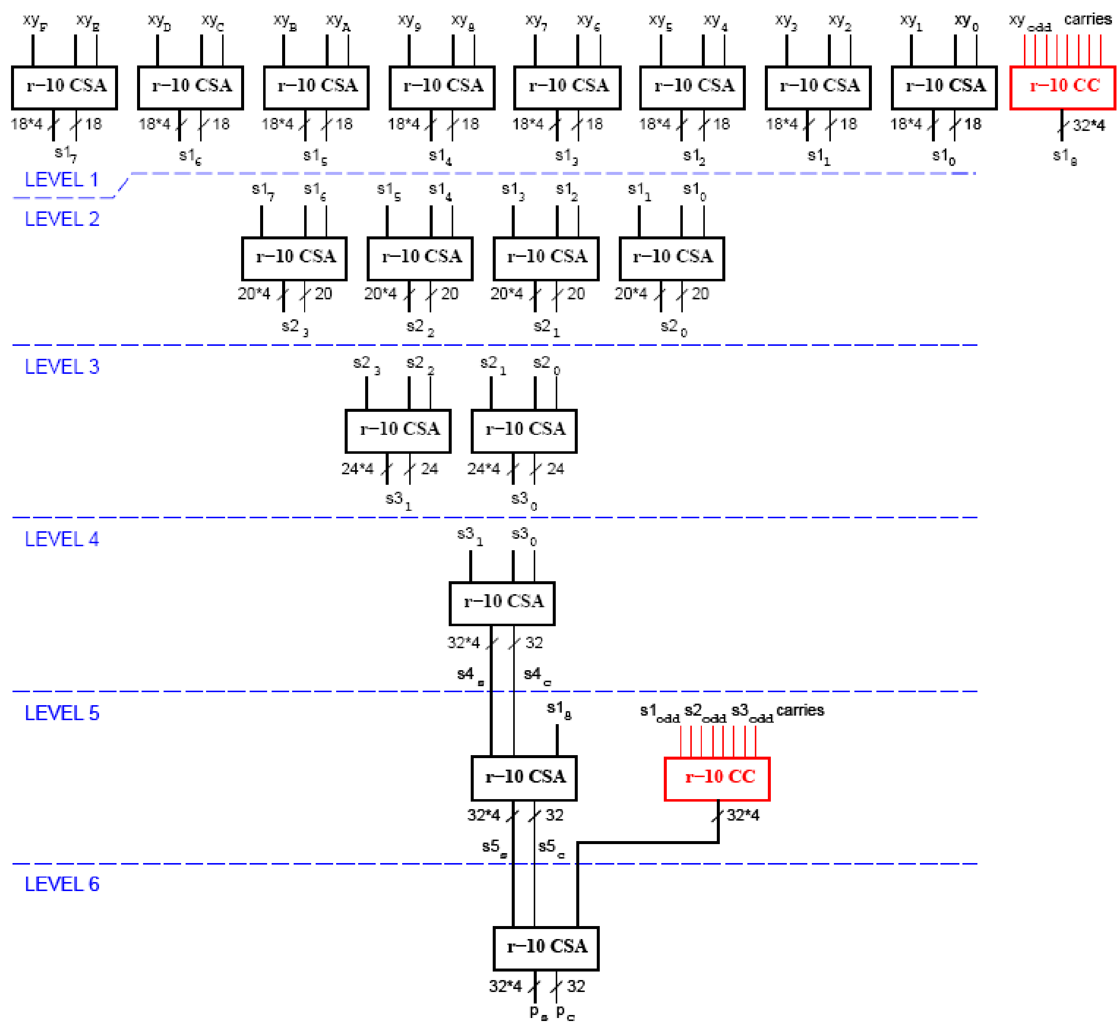


Figure 3.14 A Radix-10 Combinational Multiplier Adder tree

The final carry-propagating addition consists in converting the radix-10 carry save representation into BCD one. This is done with a radix-10 CPA in which the input is just a value in radix-10 carry-save representation and the output is the product represented in BCD.

Decimal 3:2 CSA

Vázquez in [22] proposes a 4-bit 3:2 decimal CSA using BCD-4221 format as shown in Figure 3.15a.

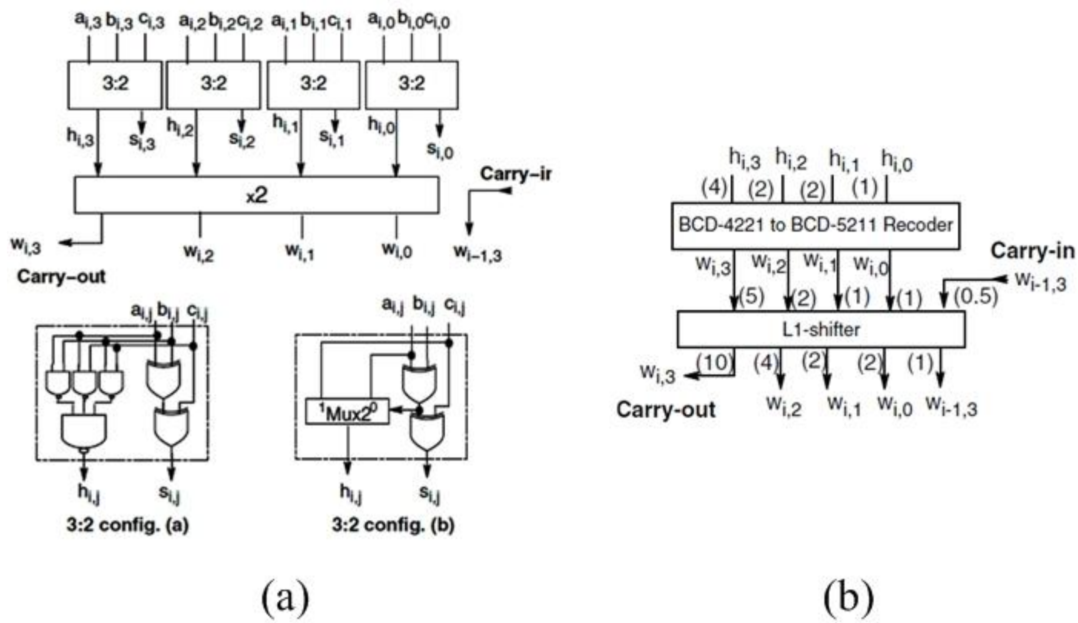


Figure 3.15 (a) 4-bit 3:2 decimal CSA (b) decimal multiplication by 2 for BCD-4221

For 16 partial products, a 16:2 decimal CSA tree is used. Figure 3.16 shows two examples of a 16:2 decimal CSA trees. The 3:2 blocks represent a 4-bit binary 3:2 CSA. The $\times 2$ blocks represent decimal multiplication by 2. In the first implementation, shown in Figure 3.16a, every carry output is multiplied by 2 before connecting to any other input. Since the carry path is slightly more complex than the sum path, outputs of block $\times 2$ are connected to fast inputs of the 3:2 CSA. The second implementation, Figure 3.16b reduces the hardware complexity by adding the carry outputs of the previous tree level before being multiplied by 2. Therefore it is necessary to perform several $\times 2$ operations in a row for some paths. Both implementations present similar critical path delays but the second implementation is preferable because of reduced hardware complexity.

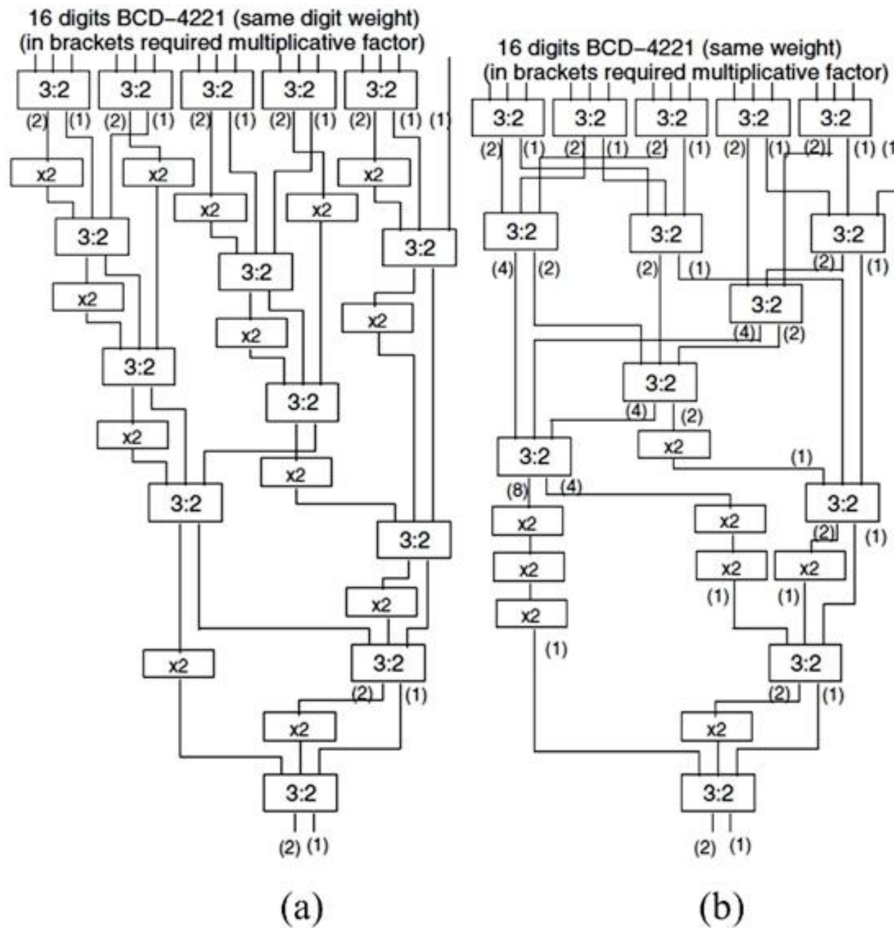


Figure 3.16 16:2 decimal CSA tree

Reduction by column tree (dadda tree) [3]

Dadda proposes a decimal column addition for the partial products via a network of carry save adders. The sum is converted to decimal format using binary to decimal converter. The decimal values are aligned then added to obtain the total sum through the addition of a few (2, 3, and at most 4) decimal numbers. This scheme, shown in Figure 3.17, is based on the following steps:

1. Binary addition of N (4-bit) column digits of equal decimal weight.

2. Binary to decimal conversion of each column sum.

3. Decimal column sums alignment according to their decimal weights. It forms an array of few (2, 3, or 4) decimal numbers, each of them n digits

long, Major Partial Sums, MPSs.

4. Decimal addition of the MPSs to obtain the final sum.

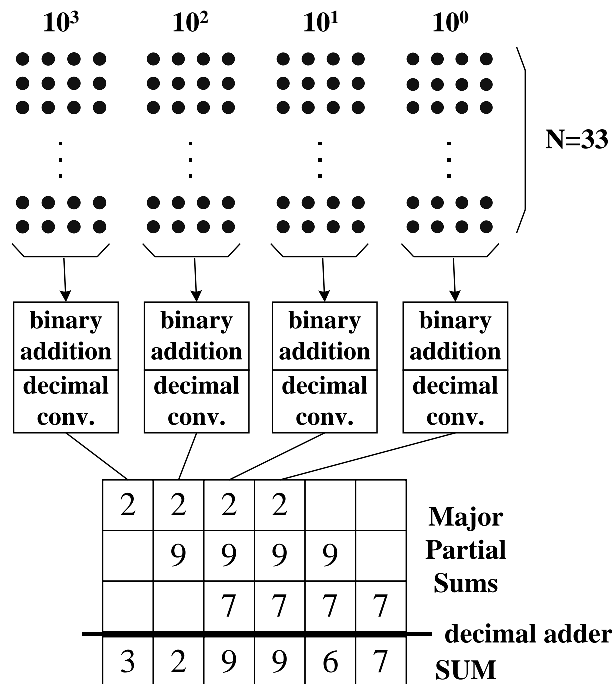


Figure 3.17 basic decimal column adder scheme for N=33 addends

3.4 Conclusion

Decimal multiplication can be implemented using sequential or parallel approaches. Parallel decimal multiplication offers a good delay reduction. It generates all partial products in parallel according to the multiplier digits then the partial products are accumulated using a decimal CSA tree. This tree reduces all partial products into two partial products then a CPA is used to obtain the final product. Decimal multiplication can be divided into three stages: multiplicand multiples generation, multiplier recoding for multiples selection, and partial products accumulation.

Chapter 4

Combined Binary/Decimal Multipliers

Recently, two combined binary/decimal multipliers are proposed. The first is proposed by Vázquez in 2007 and the other by Hickmann in 2008. This chapter introduces the two multipliers and their advantages and disadvantages.

4.1 Vázquez combined binary/decimal multiplier

Vázquez et al. [22] propose the first combined binary/decimal multiplier design approach. They use BCD-4221 format for decimal digits representation. Figure 4.1 shows a block diagram for Vázquez combined binary/decimal multiplier proposed in [22]. For multiplicand multiples generation, a binary SD radix–4 recoding and a decimal SD radix–4 or decimal SD radix–5 recoding is used. SD radix–4 generates $(A, 2A, 4A, 8A)$ multiples and SD radix–5 generates $(A, 2A, 5A, 10A)$ multiples. Two multiplexers, controlled by the multiplier digits, are used to select the suitable two multiplicand multiples for each digit. XOR gates are used for negative multiples.

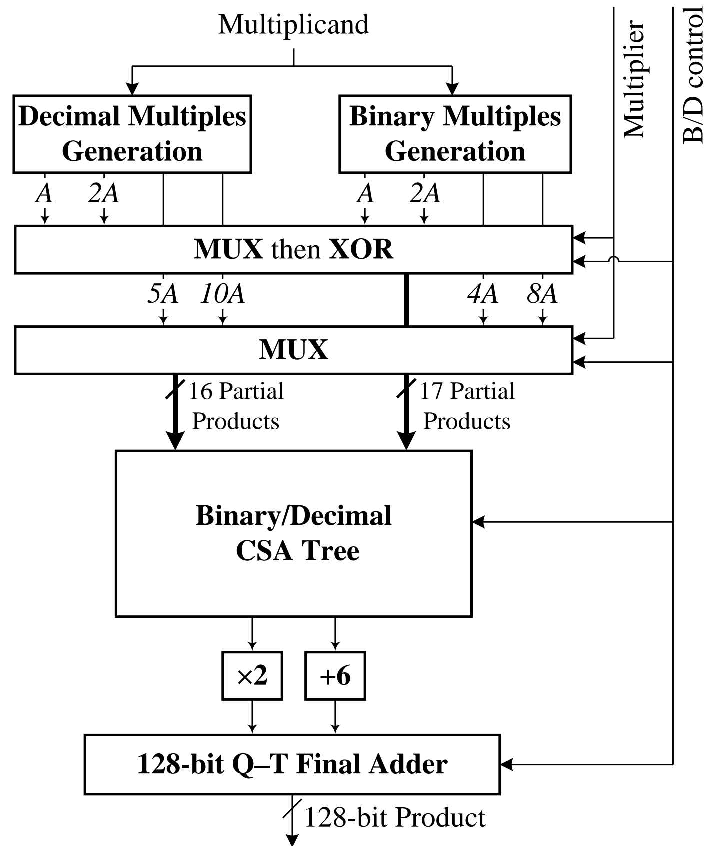


Figure 4.1 Vázquez binary/decimal multiplier.

For partial products accumulation, a shared binary/decimal carry save adder tree is used. The use of BCD-4221 format eliminates the decimal corrections needed to obtain correct decimal outputs from carry save adders. Figure 4.2 shows the carry save adder tree used. A binary/decimal multiplication by two block, $(\times 2)$, is used for carry outputs of the tree, Figure 4.3. Finally, a modified carry propagate quaternary tree adder, Q-T adder, is used to perform binary and decimal additions. To produce a correct decimal addition a conversion to BCD-8421 then a +6 operation is done to produce correct decimal digits before the Q-T addition.

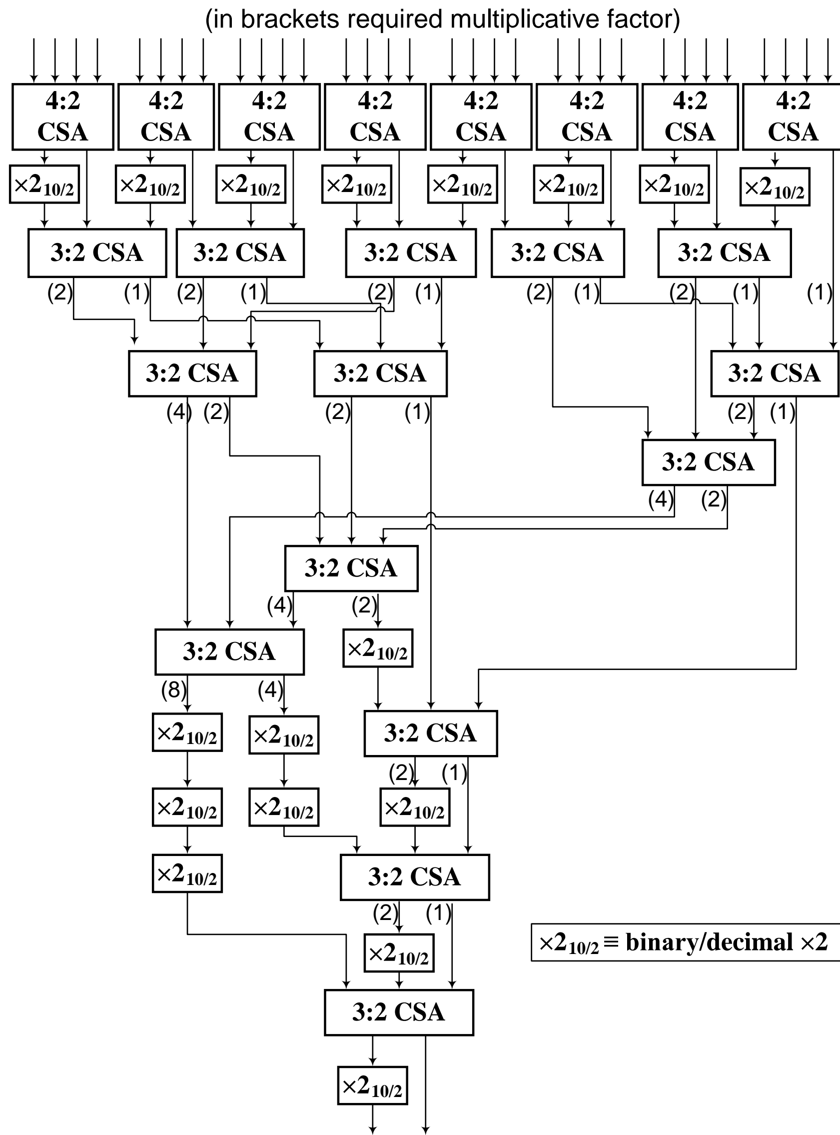


Figure 4.2 Vázquez binary/decimal CSA Tree.

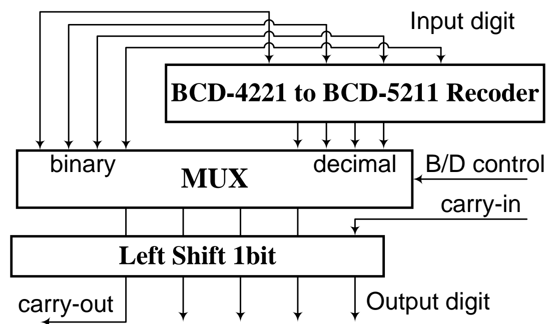


Figure 4.3 Binary/Decimal multiplication by two block.

4.2 Hickmann combined binary/decimal multiplier

Figure 4.4 shows Hickmann et al. multiplier proposed in [9]. They improve Vázquez multiplier trying to decrease the area and delay, specially the delay of binary path. They use only 3:2 CSAs in the carry save adder tree to reduce the number of binary/decimal($\times 2$) blocks. Sharing of $\times 2$ block for binary and decimal increases the area and delay of binary and decimal paths, because of the multiplexer used in it. They propose to split the binary/decimal tree at the beginning of using ($\times 2$) blocks, Figure 4.5, to avoid the extra multiplexers used compared to standalone multipliers. So the delay of binary and decimal paths are significantly reduced but with a reasonable area penalty.

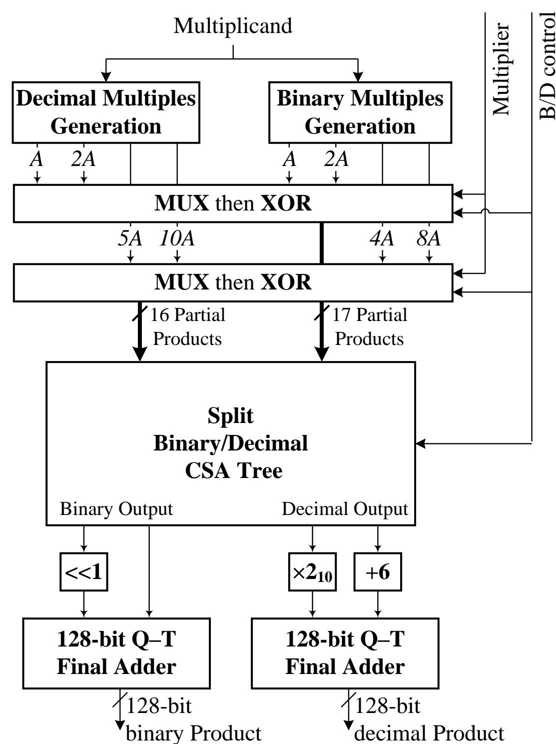


Figure 4.4 Hickmann binary/decimal multiplier.

For multiplicand multiples generation and selection, they use same Vázquez design. Also a +6 is added to sum before the final Q-T carry propagate adder. They split the carry propagate adder for binary and decimal final adder to decrease the delay but with more area increase.

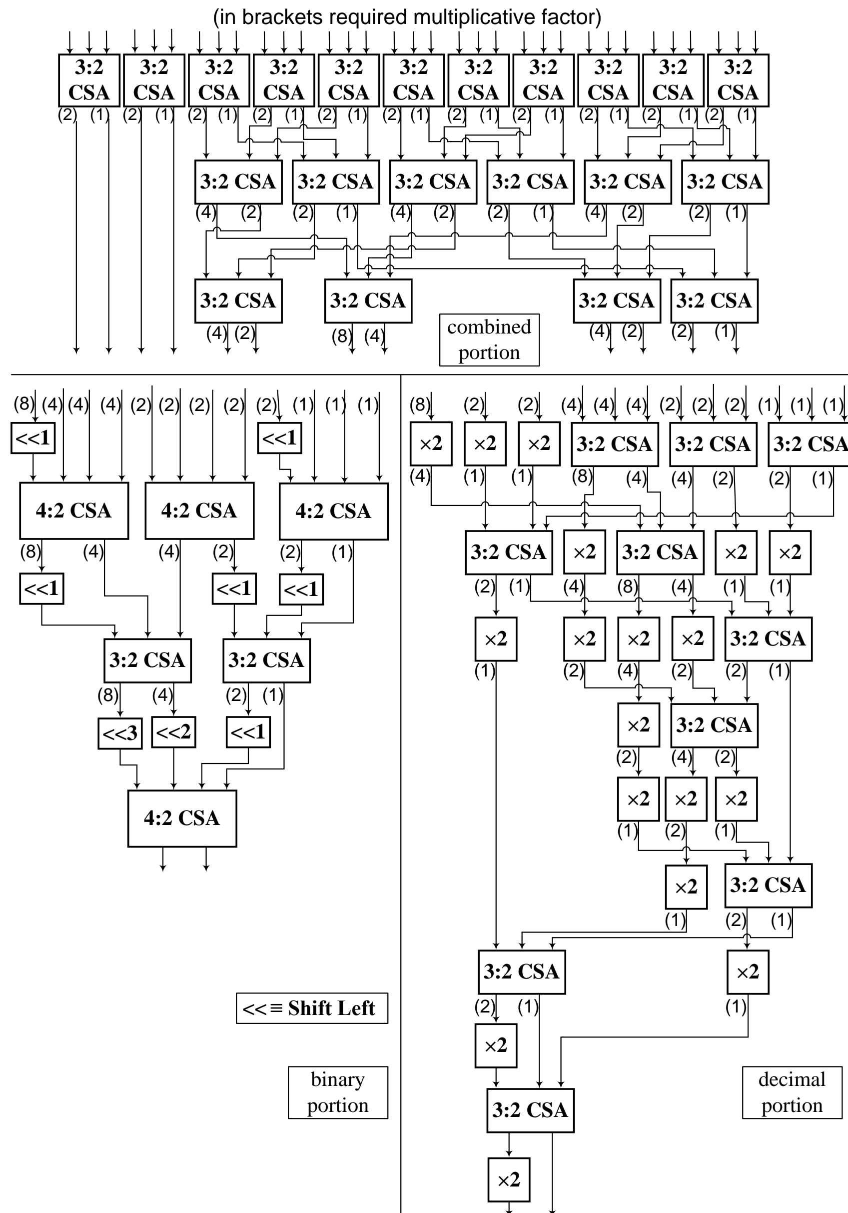


Figure 4.5 Hickmann split binary/decimal CSA Tree.

4.3 Conclusion

Vázquez et al. propose the first combined binary/decimal multiplier design. They propose a shared binary/decimal CSA tree. A multiplexer is used in each × 2 block to select between binary and decimal × 2, which lead to some increase in the area and delay specially in the binary path. Hickmann et al. split the binary and decimal CSA trees at the start of using × 2 blocks. They have some increase in area but the delay of binary and decimal paths are decreased.

Chapter 5

Proposed Combined Binary/Decimal Fixed-Point Multiplier

In this chapter, the proposed combined binary/decimal multiplier design is discussed. It allows the use of binary or decimal multiplication according to application. Binary multiplication is faster and decimal multiplication is more accurate in financial applications. We try to decrease the total area and delay of each path. The proposed multiplier takes two operands, 64-bit multiplicand (A) and 64-bit multiplier (B). It also takes a $B/D\ Control$ signal to determine whether the operands are binary or decimal, BCD-8421. When the B/D control signal has a $'0'$ value, operands are binary and when it has a $'1'$ value, operands are decimal, BCD-8421. The design consists of three stages: multiplicand multiples generation, partial products selection, and partial products accumulation. We deal with each multiplier four bits as a digit for binary and decimal multiplication so multiples from 1 to 9 for decimal and multiples from 1 to 15 for binary are generated in multiplicand multiples generation stage. The multiplier and multiplicand is presented by 16 digits.

We propose three designs for the multiplier. The first design uses $(A, 2A, 4A, 8A, 16A)$ binary multiplicand multiples and $(A, 2A, 5A, 10A)$ decimal multiplicand multiples. These multiples output three partial products

for each binary digit and two partial products for each decimal digit. Two trees are used in the partial products accumulation stage, one shared for binary and decimal, Dadda column tree. And the other for binary, Wallace tree. A final Kogge-Stone carry propagate adder is used to produce the final product.

Second design uses one column tree in the partial product accumulation stage to reduce the area. The $(A, 2A, 4A, 8A, 11A, 13A, 16A)$ binary multiplicand multiples and $(A, 2A, 5A, 10A)$ decimal multiplicand multiples are generated. These multiples generate only two partial products for binary and decimal digits.

Third design uses Booth 4 for binary recoding which reduce the number of multiplicand multiples need to be generated without increasing the number of partial products of each digit. The $(A, 2A, 4A, 8A)$ binary multiplicand multiples and $(A, 2A, 5A, 10A)$ decimal multiplicand multiples are generated. Two versions of this design is proposed, shared and split. Some improvements are used to decrease area and delay for the final proposed design.

5.1 First Proposed Design

Figure 5.1 shows the block diagram of the first proposed multiplier. The multiplier has three inputs, Multiplicand (A), Multiplier (B), and Binary/Decimal Control signal $(B/D\ Control)$.

5.1.1 Multiplicand Multiples Generation Stage

The first stage of the multiplier is generating basic multiplicand multiples. The remaining multiples is generating dynamically during the next stages by adding two/three basic multiplicand multiples. In order to decrease the delay of this stage, the tertiary set $(-2A, -A, A, 2A, 4A, 8A, 16A)$ is generated for binary multiplicand multiples and the secondary set $(-2A, -A, A, 2A, 5A, 10A)$ is generated for decimal multiplicand multiples. A tertiary set is used in binary, to eliminate the generation of 3A or 5A multiples which take a large delay O(n),

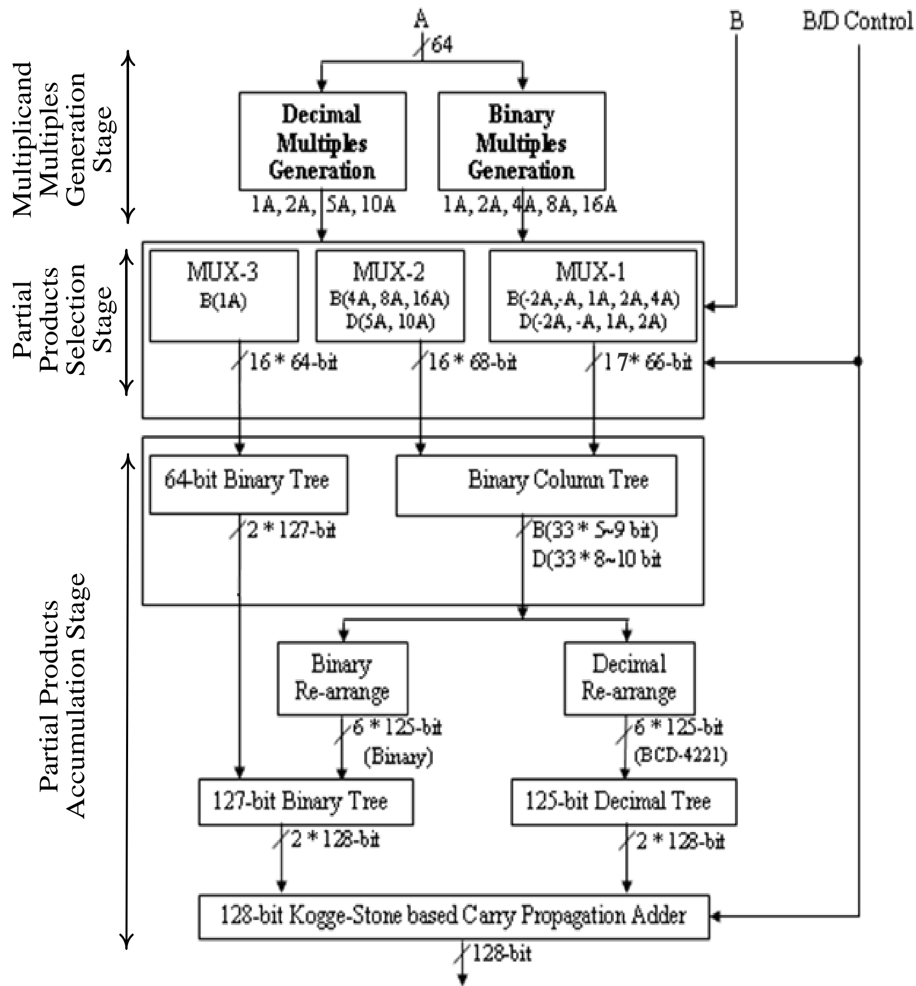


Figure 5.1 First combined binary/decimal multiplier block diagram.

where n is the number of multiplicand bits. So for each binary multiplier digit, three multiplicand multiples are selected and for each decimal multiplier digit, two multiplicand multiples are selected.

Binary multiples are generated using only shifting as shown in Figure 5.2. $2A, 4A, 8A,$ and $16A$ multiples are generated using 1bit, 2bit, 3bit, and 4bit left shifting. Negative multiples are generated using 2's complement operation obtained by generating 1's complement in this stage, by inverting each bit of positive multiple using NOT gate, then at the partial products selection stage a $plus\ one\ bit$, sign bit, is generated to be added to partial products in accumulation stage.

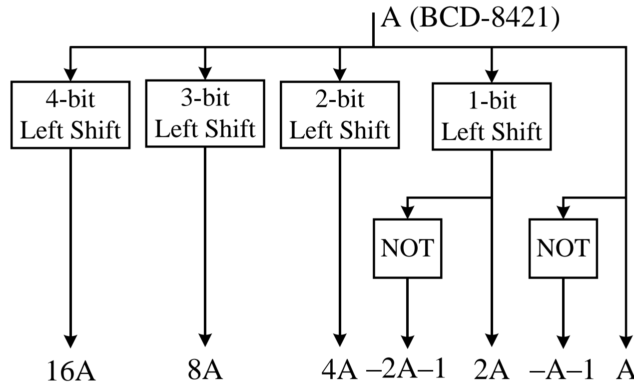


Figure 5.2 Binary multiples generation

Decimal multiples use BCD-8421 signed-digit radix-5 recoding [22], where it has a fast generation of multiplicand multiples 2A, 5A, and 10A. 2A and 5A multiples are generated using shifting and conversion between different BCD formats as shown in Figure 5.3. Decimal 10A multiple is generated using 4-bit shifting. For negative multiples, a 9's complement is obtained for each digit using two level gates combinational function. Then at the partial products selection stage a *plus one bit*, sign bit, is generated. Only $-A$ and $-2A$ negative multiples are needed and generated.

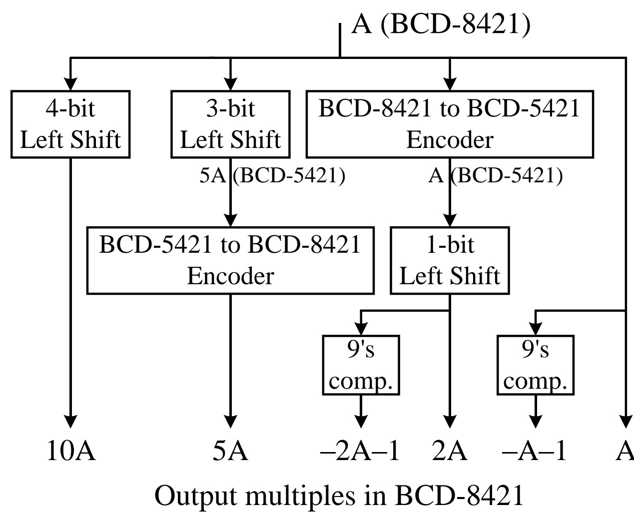


Figure 5.3 Decimal multiples generation

For the BCD-8421 to BCD-5421 encoder block, Table 5.1 shows a digit conversion from BCD-8421 to BCD-5421.

	BCD-8421				BCD-5421			
	x_3	x_2	x_1	x_0	h_3	h_2	h_1	h_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	0
3	0	0	1	1	0	0	1	1
4	0	1	0	0	0	1	0	0
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Table 5.1 BCD-8421 to BCD-5421 conversion

Each digit of this encoder is described by the following equations

$$h_0 = x_0\, \overline{x_2}\, \overline{x_3} + \overline{x_0}\, x_1\, x_2 + \overline{x_0}\, x_3 \quad (5.1)$$

$$h_1 = x_0 x_1 + x_1 \overline{x_2} + x_3 \overline{x_0} \quad (5.2)$$

$$h_2 = \overline{x_0}\, \overline{x_1}\, x_2 + x_0\, x_3 \quad (5.3)$$

$$h_3 = x_0\, x_2 + x_1\, x_2 + x_3 \quad (5.4)$$

For the BCD-5421 to BCD-8421 encoder block, Table 5.2 shows a digit conversion from BCD-5421 to BCD-8421. X means don’t care where these values do not appear after shifting the BCD-8421 three bits to the left.

	BCD-5421				BCD-8421			
	h_3	h_2	h_1	h_0	x_3	x_2	x_1	x_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	0
3	0	0	1	1	0	0	1	1
4	0	1	0	0	0	1	0	0
5	0	1	0	1	X	X	X	X
6	0	1	1	0	X	X	X	X
7	0	1	1	1	X	X	X	X
5	1	0	0	0	0	1	0	1
6	1	0	0	1	0	1	1	0
7	1	0	1	0	0	1	1	1
8	1	0	1	1	1	0	0	0
9	1	1	0	0	1	0	0	1
10	1	1	0	1	X	X	X	X
11	1	1	1	0	X	X	X	X
12	1	1	1	1	X	X	X	X

Table 5.2 BCD-5421 to BCD-8421 conversion

Each digit of this block is described by the following equations

$$x_0 = h_0 \oplus h_3 \quad (5.5)$$

$$x_1 = h_1 \overline{h_3} + \overline{h_0} h_1 + h_0 \overline{h_1} h_3 \quad (5.6)$$

$$x_2 = h_2 \overline{h_3} + \overline{h_1} \overline{h_2} h_3 + \overline{h_0} h_1 h_3 \quad (5.7)$$

$$x_3 = h_0 h_1 h_3 + h_2 h_3 \quad (5.8)$$

The decimal BCD-8421 9's complement block is implemented using two level logic gates. Table 5.3 shows a BCD-8421 9's complement truth table.

(BCD-8421)					9's complement (BCD-8421)				
	y_3	y_2	y_1	y_0	z_3	z_2	z_1	z_0	
0	0	0	0	0	1	0	0	1	9
1	0	0	0	1	1	0	0	0	8
2	0	0	1	0	0	1	1	1	7
3	0	0	1	1	0	1	1	0	6
4	0	1	0	0	0	1	0	1	5
5	0	1	0	1	0	1	0	0	4
6	0	1	1	0	0	0	1	1	3
7	0	1	1	1	0	0	1	0	2
8	1	0	0	0	0	0	0	1	1
9	1	0	0	1	0	0	0	0	0

Table 5.3 9's complement of BCD-8421 digits

It is described by the following equations

$$z_0 = \overline{y_0} \tag{5.9}$$

$$z_1 = y_1 \tag{5.10}$$

$$z_2 = y_1 \oplus y_2 \tag{5.11}$$

$$z_3 = \overline{y_1}\,\overline{y_2}\,\overline{y_3} \tag{5.12}$$

5.1.2 Partial Products Selection Stage

After generating the basic multiplicand multiples, the suitable two/three multiplicand multiples is selected according to multiplier digits using two/three

multiplexers for decimal/binary paths. Binary multiplicand multiples set is divided into three groups for the three multiplexers while decimal multiplicand multiples set is divided into two groups for the two multiplexers. Two multiplexers are shared between binary and decimal selection to choose the two suitable multiplicand multiples. The third multiplexer is used for the binary third group which chooses the third suitable binary multiplicand multiple.

Binary set is divided into $(-2A, -A, A, 2A, 4A)$, $(4A, 8A, 16A)$ and (A) groups. The third group is to generate 11A and 13A multiples without the need to generate 3A or 5A multiples. The binary partial products are selected according to Table 5.4.

Multiple	MUX1 selection	MUX2 selection	MUX3 selection
0	0	0	0
A	A	0	0
2A	2A	0	0
3A	−A	4A	0
4A	0	4A	0
5A	A	4A	0
6A	2A	4A	0
7A	−A	8A	0
8A	0	8A	0
9A	A	8A	0
10A	2A	8A	0
11A	2A	8A	A
12A	4A	8A	0
13A	4A	8A	A
14A	−2A	16A	0
15A	−A	16A	0

Table 5.4 Binary multiplicand multiples selection.

Decimal set is divided into $(-2\text{A}, -\text{A}, \text{A}, 2\text{A})$ and $(5\text{A}, 10\text{A})$ groups. Table 5.5 shows the decimal partial products selection from the two multiplexers.

Multiple	MUX1 multiple selection	MUX2 multiple selection
0	0	0
A	A	0
2A	2A	0
3A	−2A	5A
4A	−A	5A
5A	0	5A
6A	A	5A
7A	2A	5A
8A	−2A	10A
9A	−A	10A

Table 5.5 Decimal multiplicand multiples selection.

This stage outputs 49 binary partial products, $3 \times (16 \text{ multiplier digit}) + 1$ (for $plus\ one$ sign bits). 33 decimal partial products, $2 \times (16 \text{ multiplier digit}) + 1$ (for $plus\ one$ sign bits) are output.

A two level gate multiplexer is used as shown in Figure 5.4. Where cond1 means condition of selecting A multiple, cond2_B means condition of selecting $2A$ multiple for binary operands, cond2_D means condition of selecting $2A$ multiple for decimal operands, Inv means condition of inverting for negative multiples, and so on. The conditions that control them depend on multiplier digits (b_i) and $B/D\ Control$ signal, (c), where $c = '0'$ means binary input operands and $c = '1'$ means decimal input operands. The conditions that control the multiplexers are similar for all digits. First digit equations are:

For MUX1

$$cond1 = (b_0\,\overline{b_3} + b_0\,\overline{b_1}\,\overline{b_2} + b_0\,b_1\,b_2)\cdot\bar{c} + \left(\overline{b_0}\,b_2 + b_0\,\overline{b_1}\,\overline{b_2}\right)\cdot c \tag{5.13}$$

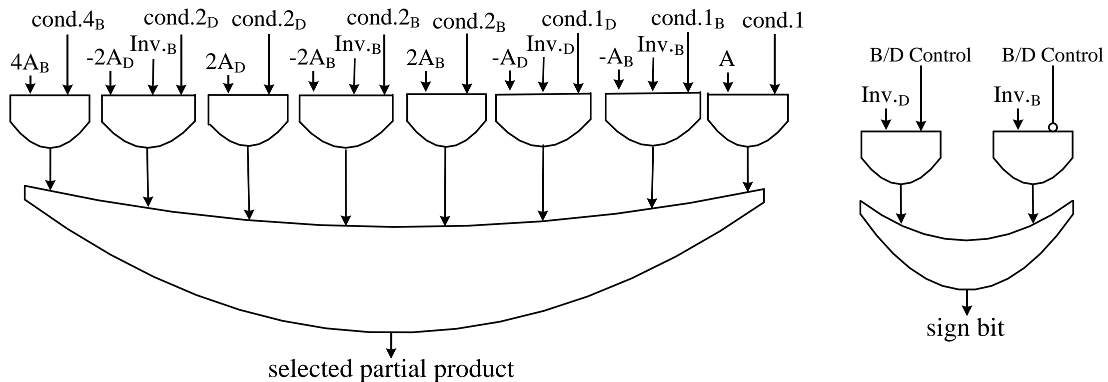
$$cond2_B = \left(\overline{b_0}\,b_1 + b_1\,\overline{b_2}\,b_3\right)\cdot\bar{c} \tag{5.14}$$

$$cond4_B = \left(\overline{b_1}\,b_2\,b_3\right)\cdot\bar{c} \tag{5.15}$$

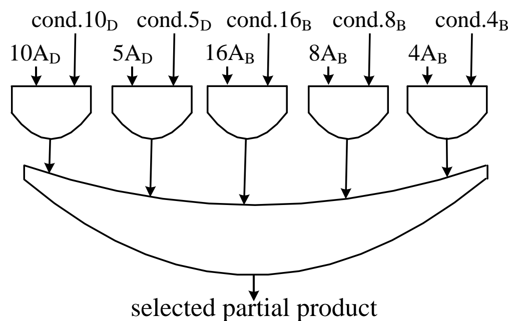
$$cond2_D = \left(\overline{b_0}\,b_3 + b_0\,b_1 + b_1\,\overline{b_2}\right)\cdot c \tag{5.15}$$

$$\text{Inv}_B = \left(b_0\,b_1\,\overline{b_3} + b_1\,b_2\,b_3\right) \tag{5.16}$$

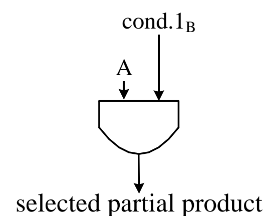
$$\text{Inv}_D = (b_3 + b_0\,b_1\,\overline{b_2} + \overline{b_0}\,\overline{b_1}\,b_2) \tag{5.17}$$



(a) MUX1



(b) MUX2



(c) MUX3

Figure 5.4 Multiplexers design for each multiplier digit.

For MUX2

$$cond4_B = \left(\overline{b_0}\, b_2\, \overline{b_3} + \overline{b_1}\, b_2\, \overline{b_3} + b_0\, b_1\, \overline{b_2}\, \overline{b_3}\,\right) \cdot \bar{c} \quad (5.18)$$

$$cond8_B = \left(\overline{b_2}\, b_3 + \overline{b_1}\, b_3 + b_0\, b_1\, b_2\, \overline{b_3}\right) \cdot \bar{c} \quad (5.19)$$

$$cond16_B = (b_1\, b_2\, b_3) \cdot \bar{c} \quad (5.20)$$

$$cond5_D = \left(b_0\, b_1 + b_2\, \overline{b_3}\right) \cdot c \quad (5.21)$$

$$cond10_D = (b_3) \cdot c \quad (5.22)$$

For MUX3

$$cond1_B = (b_0\, b_3(b_1 \oplus b_2)) \cdot \bar{c} \quad (5.23)$$

5.1.3 Partial Products Accumulation Stage

After generating all partial products, a tree of adders is used to add them. Irregular tree topologies are used in order to minimize the total delay. MUX1 output 17 partial products and MUX2 output 16 partial products. They are common for binary and decimal. Each partial product is shifted to its right weight according to its multiplier digit position. Then they enter to a binary column tree, shared for binary and decimal. A binary tree is used here to save the correction delays of decimal addition due to the six invalid BCD-8421 digits, from 10 to 15. Column tree did not allow the pass of carry bit to next digit which should be of order 16 in binary addition and of order 10 in decimal addition. These different between binary and decimal need a multiplexer to choose the correct carry for each path. Binary tree solve this problem which

save delay and area. The binary column tree used here is similar to Dadda's tree in [3]. Nevertheless, the proposed tree replaces the binary addition by a carry save addition to decrease the delay. Also, each column has different number of digits to be added according to partial products different weights. It adds every 4-bit digit for the 33 partial products out from MUXs1 and MUXs2 using binary carry save adders. Each column out sum and carry output. Figure 5.5 shows the scheme of the binary column tree used.

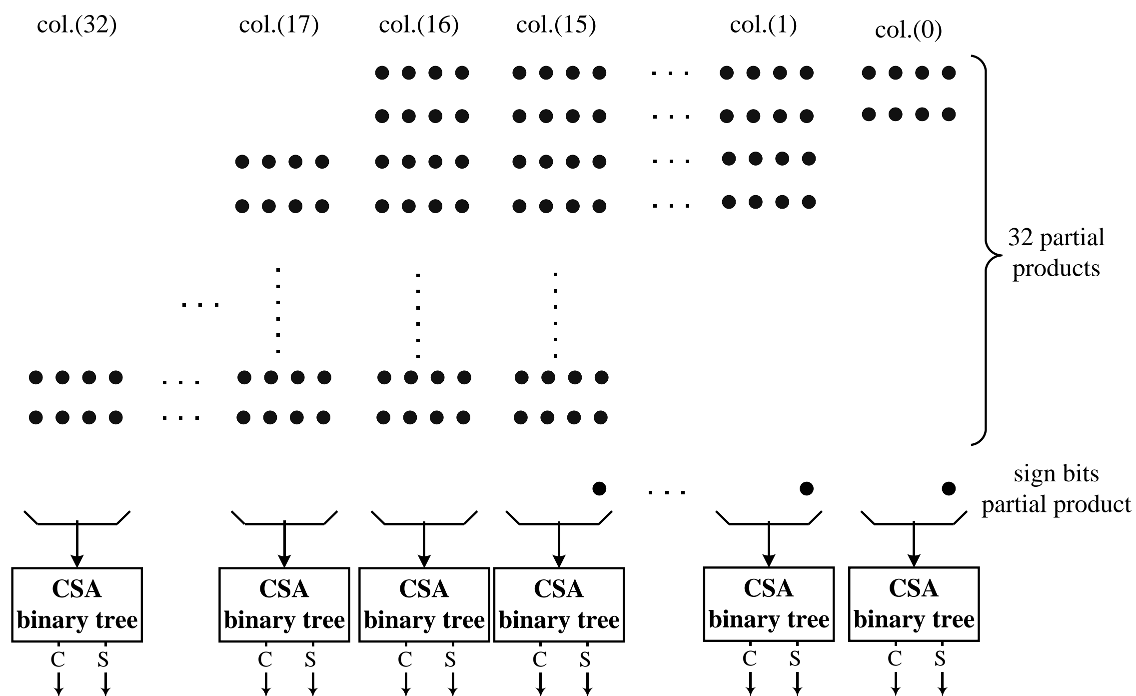


Figure 5.5 Binary column tree scheme.

Every column has a different number of digits. A CSA tree used for each column to add its digits as in [3]. Column number 0 has two digits from first two partial products, for first multiplier digit, and one sign bit for negative multiplicand multiples. Column number 15 has the maximum number of digits to be added, 32 digits plus 1 sign bit. Figure 5.6 shows the binary CSA tree of digit(15). Sign bit is added to first bit of shifted carry digits in the tree. Sign

extension is not needed here where multiplicand and multiplier are positive, there output sign is calculated separately. Two partial products are generated for each multiplier digit, there summation is positive.

In parallel to the column tree, a binary row tree, Wallace tree, is used to add the binary partial products out from MUXs3. Figure 5.7 shows a scheme of the row tree. Figure 5.8 shows the CSA binary tree block diagram.

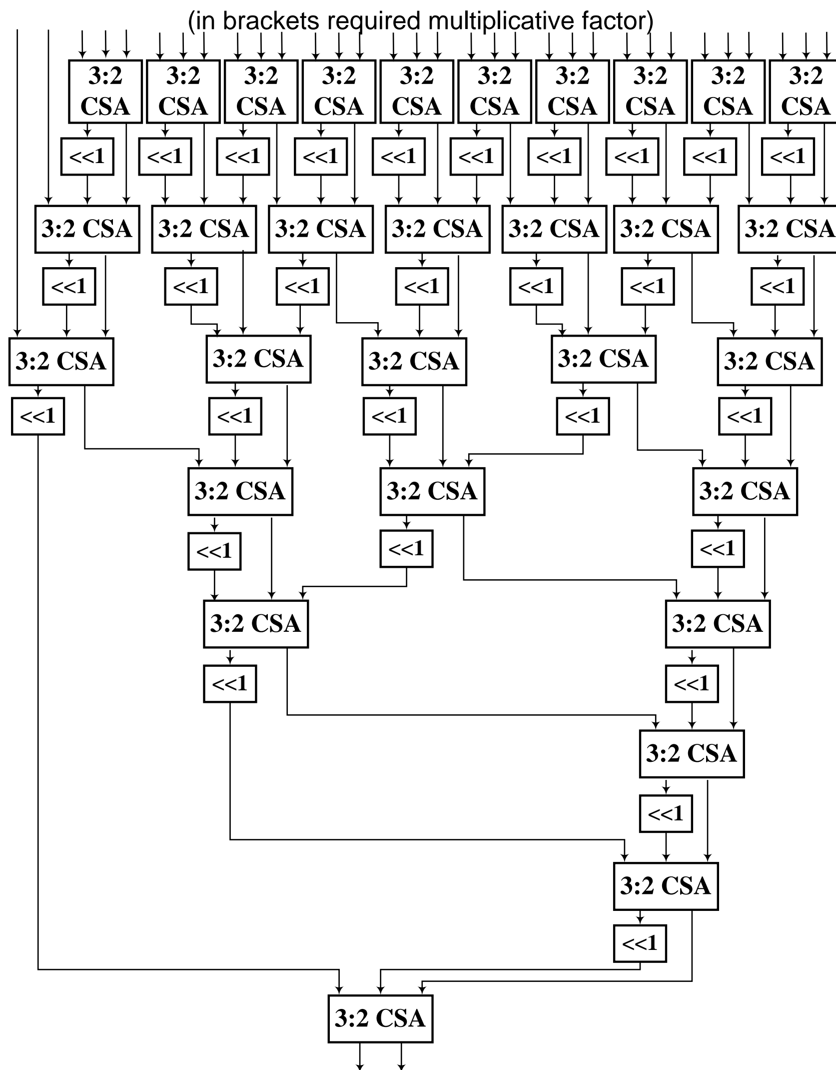


Figure 5.6 CSA binary tree (for 32 digits, 4-bit).

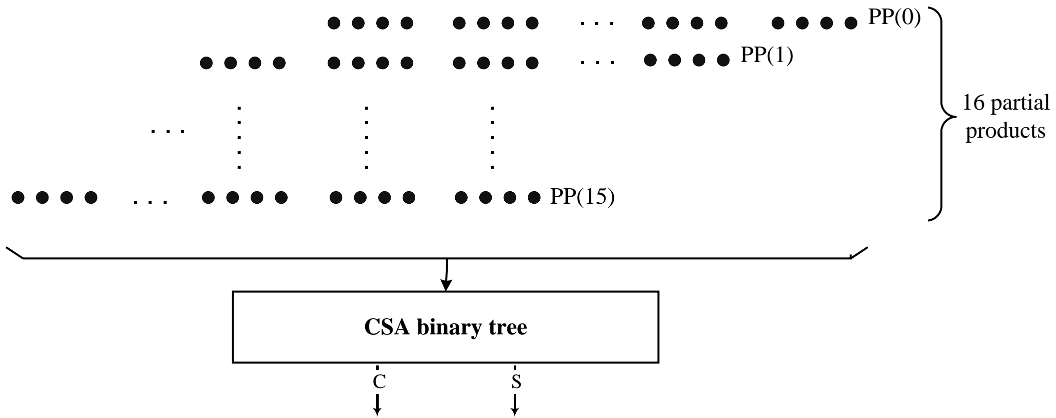


Figure 5.7 64-bit binary CSA tree for the 16 partial products out of MUXs3.

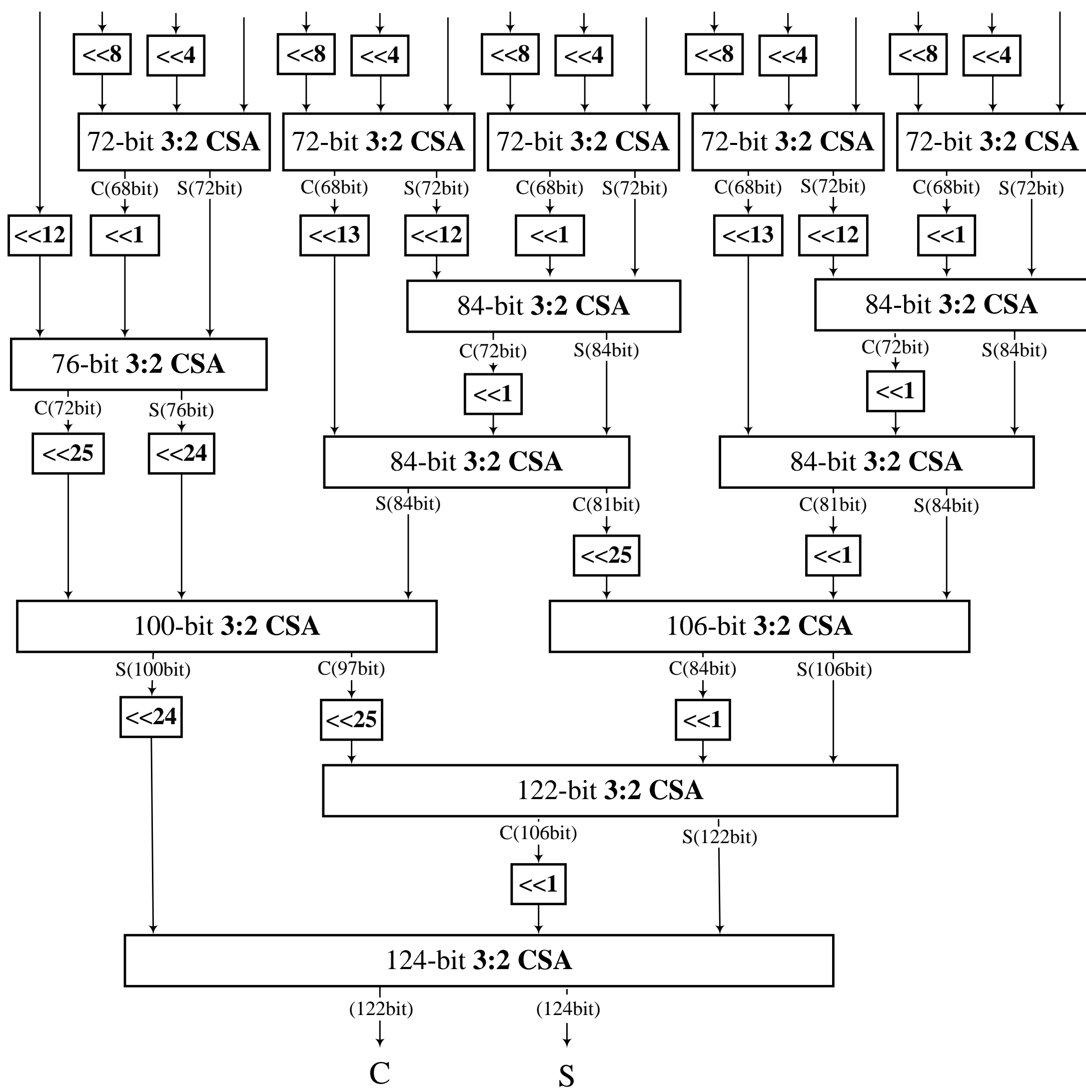


Figure 5.8 CSA binary tree (for 16 partial products, 64-bit).

The output of column tree is rearranged [3], according to the B/D Control signal. In case of Binary, it is rearranged to 2 Major Partial Sums and 2 Major Partial Carries. In case of Decimal, it is firstly converted to BCD-4221, all valid BCD code, to use binary CSA without decimal correction. Then they rearranged to 3 Major Partial Sums and 3 Major Partial Carries. Two separate trees are used after that, a decimal one to add the 6 decimal Major Partial Sums and Carries, and a binary one to add the 4 binary Major Partial Sums and Carries with the sum and carry out from the binary CSA tree.

Finally a two parallel Kogge-Stone carry propagate adders are used to add the final sum and carry partial products to produce the final product (P). The separation between binary and decimal trees eliminates the latency of decimal corrections from the binary multiplication path.

5.2 Second proposed Design

Partial products accumulation stage is the most significant multiplier stage since it has the largest area and delay. In the second design, we try to decrease its area with small increase of the delay by generating 11A and 13A multiplicand multiples in the first stage. These two multiples need an addition of three multiplicand multiples from basic multiples $(A, 2A, 4A, 8A, 16A)$, which generated using only shifting. So we use a secondary set for binary and decimal multiplicand multiples generation. The second tree, binary CSA row tree, is not needed. We use only one tree, binary column tree. Figure 5.9 shows the second combined binary/decimal multiplier design block diagram.

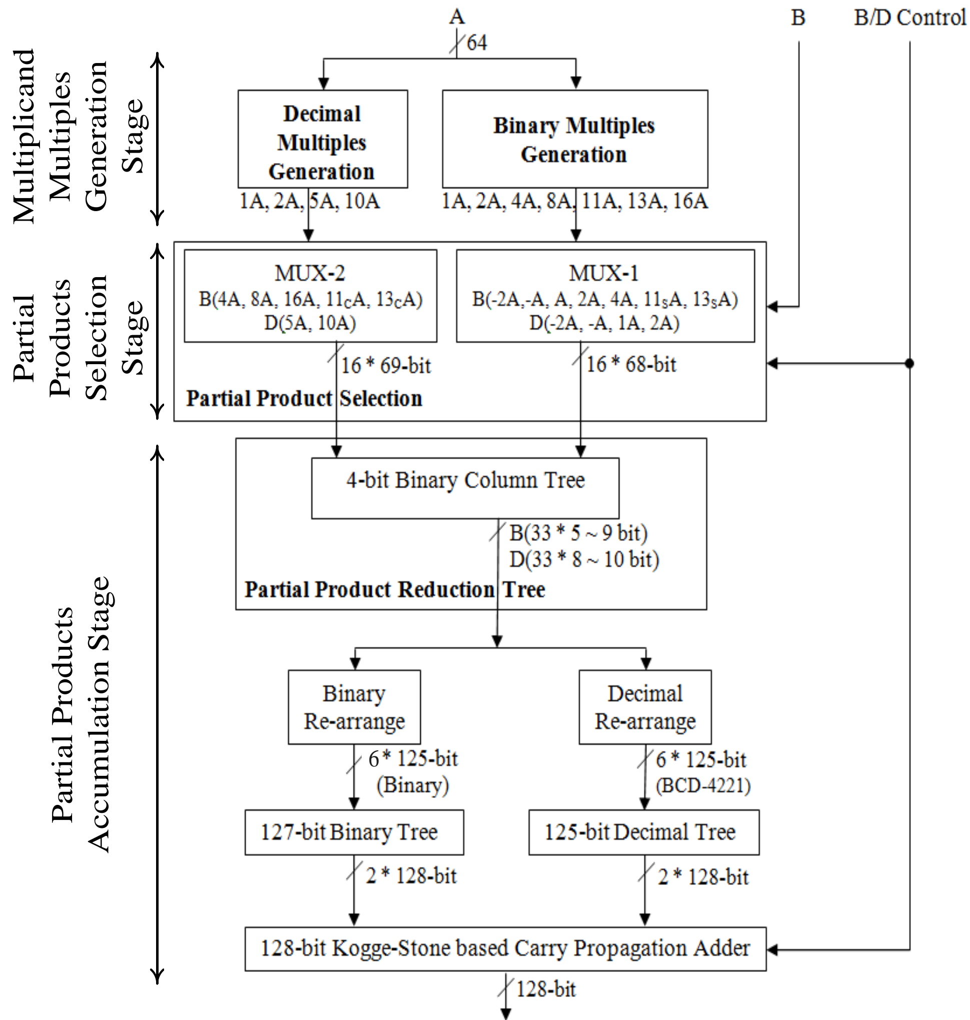


Figure 5.9 Second combined binary/decimal multiplier block diagram.

5.2.1 Multiplicand Multiples Generation Stage

Secondary sets are used which generate 32 partial products for the 16 digits multiplier plus 1 partial product for sign bits. Tertiary sets are not considered where it generates 49 partial products which need an extra tree in the partial product accumulation stage and increase the area by high factor.

For binary multiples, $(-2A, -A, A, 2A, 4A, 8A, 11_SA, 11_CA, 13_SA, 13_CA, 16A)$ secondary set is generated. The subscript S indicates sum and the subscript C indicates carry, for the output of the CSA. Two carry save adders

are used to generate the multiples 11A and 13A in CSA format where these two multiples need the addition of three multiples ($8A + 2A + A = 11A$ and $8A + 4A + A = 13A$). Figure 5.10 and Figure 5.11 show the design of the CSA. They take a delay of CSA, four gate delays, but they save 16 extra partial products for using a tertiary set. Negative multiples are generated using 2's complement operation obtained by generating 1's complement in this stage then at the partial products selection stage a $plus\ one\ bit$, sign bit, is generated to be added to partial products in accumulation stage. Figure 5.12 shows the binary multiplicand multiples generation.

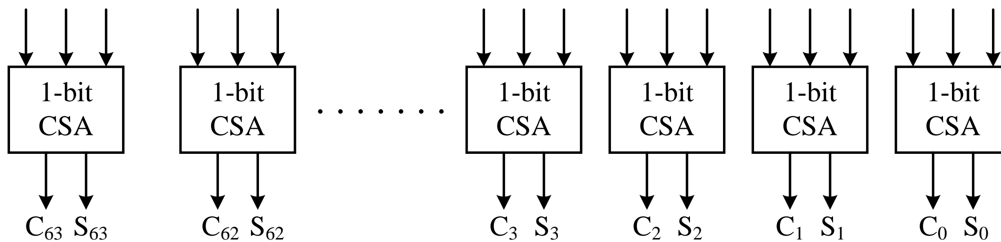


Figure 5.10 Three input, 64-bit, Carry Save Adder

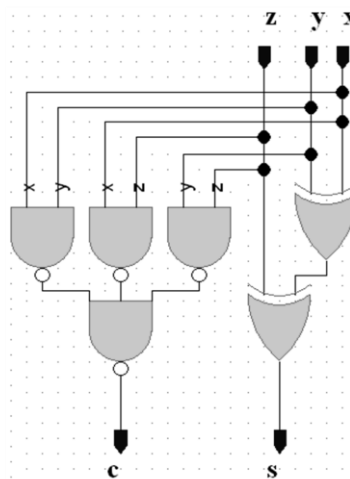


Figure 5.11 1-bit Carry Save Adder

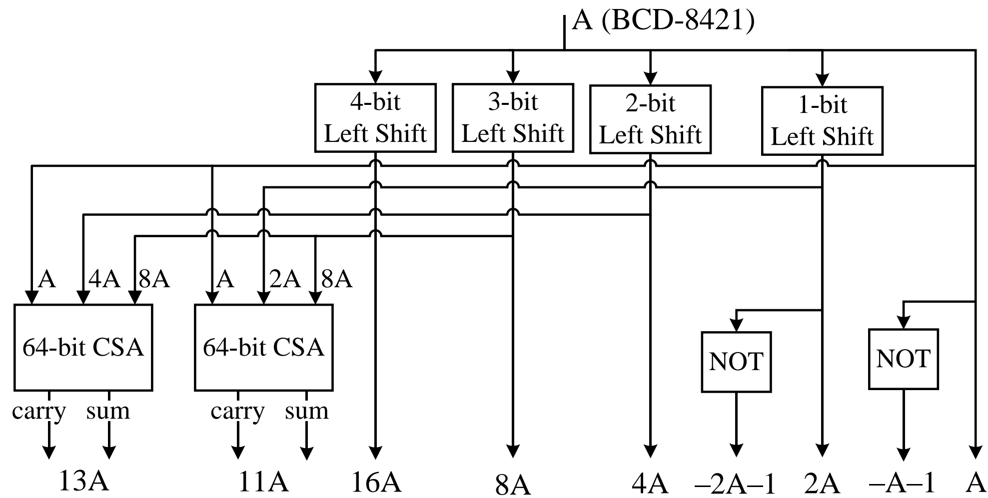


Figure 5.12 binary multiples generation

For decimal, we generate the multiplicand multiples $(-2A, -A, A, 2A, 5A, 10A)$ as in first design, Figure 5.3.

5.2.2 Partial Products Selection Stage

Each binary and decimal multiplicand multiples set is divided into two groups, where secondary sets are used for binary and decimal. Two multiplexers are used to choose the two suitable multiplicand multiples for each multiplier digit.

Binary set is divided into $(-2\text{A}, -\text{A}, \text{A}, 2\text{A}, 4\text{A}, 11_{\text{S}}\text{A}, 13_{\text{S}}\text{A})$ and $(4\text{A}, 8\text{A}, 11_{\text{C}}\text{A}, 13_{\text{C}}\text{A}, 16\text{A})$ groups. The binary partial products are selected according to Table 5.6.

Decimal set is divided into $(-2\text{A}, -\text{A}, \text{A}, 2\text{A})$ and $(5\text{A}, 10\text{A})$ groups as in first design. Table 5.5 shows the decimal partial products selection from the two multiplexers.

Multiple	MUX1 multiple selection	MUX2 multiple selection
0	0	0
A	A	0
2A	2A	0
3A	−A	4A
4A	0	4A
5A	A	4A
6A	2A	4A
7A	−A	8A
8A	0	8A
9A	A	8A
10A	2A	8A
11A	11_SA	11_CA
12A	4A	8A
13A	13_SA	13_CA
14A	−2A	16A
15A	−A	16A

Table 5.6 Binary multiplicand multiples selection.

This stage outputs 33 partial products. Figure 5.13 shows the partial products selection block diagram for binary and decimal multiplication.

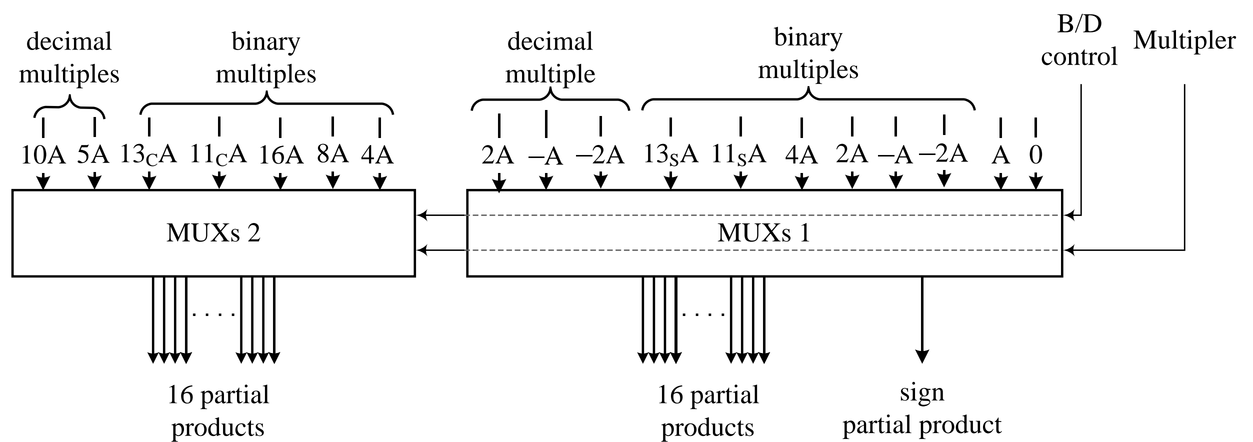


Figure 5.13 Partial products selection.

The conditions that control the multiplexers depend on multiplier digits (b_i) and B/D Control signal (c), where $c = '0'$ for binary input operands and $c = '1'$ for decimal input operands. The equations that control the two multiplexers are:

For MUXs1

$$cond1 = (b_0\,\overline{b_3} + b_0\,\overline{b_1}\,\overline{b_2} + b_0\,b_1\,b_2)\cdot\bar{c} + \left(\overline{b_0}\,b_2 + b_0\,\overline{b_1}\,\overline{b_2}\right)\cdot c \tag{5.24}$$

$$cond2_B = \left(\overline{b_0}\,b_1\right)\cdot\bar{c} \tag{5.25}$$

$$cond4_B = \left(\overline{b_0}\,\overline{b_1}\,b_2\,b_3\right)\cdot\bar{c} \tag{5.26}$$

$$cond11S_B = \left(b_0\,b_1\,\overline{b_2}\,b_3\right)\cdot\bar{c} \tag{5.27}$$

$$cond13S_B = \left(b_0\,\overline{b_1}\,b_2\,b_3\right)\cdot\bar{c} \tag{5.28}$$

$$cond2_D = \left(\overline{b_0}\,b_3 + b_0\,b_1 + b_1\,\overline{b_2}\right)\cdot c \tag{5.29}$$

$$\mathrm{Inv}_B = \left(b_0\,b_1\,\overline{b_3} + b_1\,b_2\,b_3\right) \tag{5.30}$$

$$\mathrm{Inv}_D = (b_3 + b_0\,b_1\,\overline{b_2} + \overline{b_0}\,\overline{b_1}\,b_2) \tag{5.31}$$

For MUXs2

$$cond4_B = \left(\overline{b_0}\,b_2\,\overline{b_3} + \overline{b_1}b_2\,\overline{b_3} + b_0\,b_1\,\overline{b_2}\,\overline{b_3}\right)\cdot\bar{c} \tag{5.32}$$

$$cond8_B = \left(\overline{b_0}\,\overline{b_1}\,b_3 + \overline{b_1}\,\overline{b_2}\,b_3 + \overline{b_0}\,\overline{b_2}\;b_3 + b_0\,b_1\,b_2\,\overline{b_3}\right)\cdot\bar{c} \tag{5.33}$$

$$cond11C_B = \left(b_0\,b_1\,\overline{b_2}\,b_3\right)\cdot\bar{c} \tag{5.34}$$

$$cond13C_B = \left(b_0\ \overline{b_1}\ b_2\ b_3\right) \cdot \bar{c} \quad (5.35)$$

$$cond16_B = (b_1\ b_2\ b_3) \cdot \bar{c} \quad (5.36)$$

$$cond5_D = \left(b_0\ b_1 + b_2\ \overline{b_3}\right) \cdot c \quad (5.37)$$

$$cond10_D = (b_3) \cdot c \quad (5.38)$$

5.2.3 Partial Products Accumulation Stage

One binary column tree is used to add the 33 binary/decimal partial products outputs from MUXs1 and MUXs2, Figure 5.5 and Figure 5.6.

The output of the column tree is rearranged according to the B/D Control signal. In case of Binary, it is rearranged into two Major Partial Sums and two Major Partial Carries. In case of Decimal, it is firstly converted to BCD-4221, all valid BCD code, to use binary CSA without decimal correction. Then they rearranged to 3 Major Partial Sums and 3 Major Partial Carries. Two separate trees are used after that, a decimal one to add the 6 decimal Major Partial Sums and Carries, and a binary one to add the 4 binary Major Partial Sums and Carries.

Finally a binary/decimal Kogge-Stone based carry propagate adder is used to add the final sum and carry partial products which produce the final product (P).

5.3 Third Proposed Design

To eliminate the delay of the two CSAs that generate $11A$ and $13A$ without the need to use tertiary sets in multiplicand multiples generation, Booth-4 binary recoding is used. It reduces the number of multiplicand multiples needed to be generated without increasing the number of partial products of each digit. Only the secondary set $(A, 2A, 4A, 8A)$ multiplicand multiples are generated for

binary multiplication. 33 partial products are generated and added using the binary column tree for binary and decimal partial products. After the binary column tree, columns output is rearranged in 4/6 bit vectors for binary/decimal multiplication. Two schemes are implemented for the addition of these bit vectors, shared and split. Figure 5.14 shows the block diagram of the proposed combined binary/decimal multiplier design.

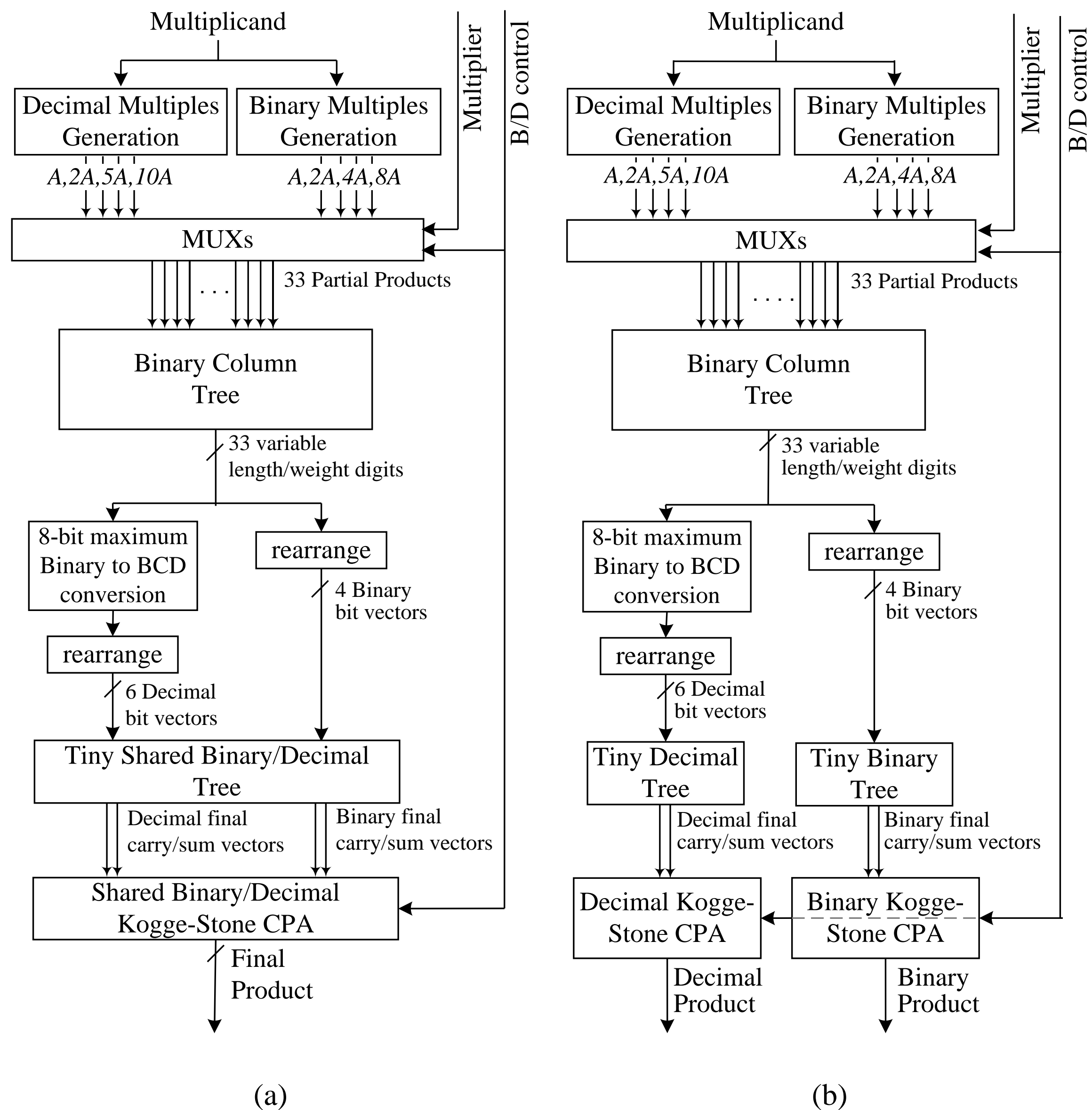


Figure 5.14 Proposed combined binary/decimal multiplier (a) shared design, (b) split design.

5.3.1 Multiplicand Multiples Generation Stage

For binary multiplicand multiples, Booth4 recoding is used which only need the generation of $(-8\text{A}, -4\text{A}, -2\text{A}, -\text{A}, \text{A}, 2\text{A}, 4\text{A}, 8\text{A})$ multiplicand multiples, Figure 5.15. All multiplicand multiples are generated using only shifting. Negative multiples are generated using 2's complement operation.

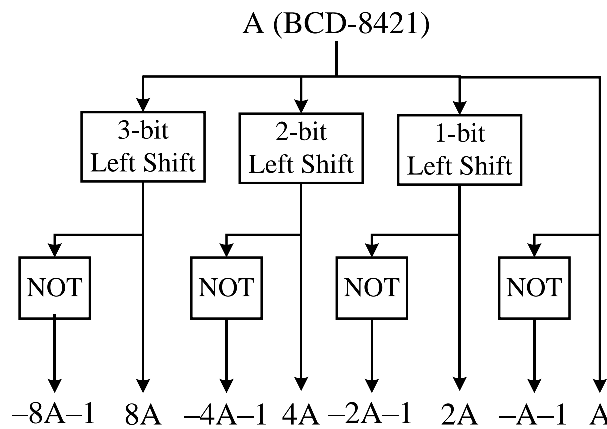


Figure 5.15 Used binary multiples generation.

Decimal multiplicand multiples are generated as in first design, Figure 5.3.

5.3.2 Partial Products Selection Stage

Each binary and decimal multiplicand multiples set is divided into two groups, where secondary sets are used for binary and decimal. Binary set is divided into $(-2\text{A}, -\text{A}, \text{A}, 2\text{A})$ and $(-8\text{A}, -4\text{A}, 4\text{A}, 8\text{A})$ groups. Table 5.7 shows the partial products selection according to Booth4 recoding, where $(b_{i+3}. b_{i+2}. b_{i+1}. b_i)$ represent the present multiplier digit, and b_{i-1} represent the most significant bit of the previous digit. Negative multiples are needed in the two groups, so two sign bits are generated for each multiplier

digit. For binary booth4, the multiplier is padded with one ‘0’ bit to the right and four ‘0’ bits to the left, so it divided into 17 digits. For the first 16 multiplier digits, two multiplicand multiples are selected. The last digit, last five bits, is ‘00000’ or ‘00001’, so it needs only one partial product, where it selects between 0 or A multiplicand multiple.

Bit					Operation	MUX1 multiple selection	MUX2 multiple selection
2^3	2^2	2^1	2^0	2^{-1}			
b_{i+3}	b_{i+2}	b_{i+1}	b_i	b_{i-1}			
0	0	0	0	0	0	0	0
0	0	0	0	1	+A	A	0
0	0	0	1	0	+A	A	0
0	0	0	1	1	+2A	2A	0
0	0	1	0	0	+2A	−2A	4A
0	0	1	0	1	+3A	−A	4A
0	0	1	1	0	+3A	−A	4A
0	0	1	1	1	+4A	0	4A
0	1	0	0	0	+4A	0	4A
0	1	0	0	1	+5A	A	4A
0	1	0	1	0	+5A	A	4A
0	1	0	1	1	+6A	2A	4A
0	1	1	0	0	+6A	−2A	8A
0	1	1	0	1	+7A	−A	8A
0	1	1	1	0	+7A	−A	8A
0	1	1	1	1	+8A	0	8A
1	0	0	0	0	−8A	0	−8A
1	0	0	0	1	−7A	A	−8A
1	0	0	1	0	−7A	A	−8A
1	0	0	1	1	−6A	2A	−8A
1	0	1	0	0	−6A	−2A	−4A
1	0	1	0	1	−5A	−A	−4A
1	0	1	1	0	−5A	−A	−4A

Bit					Operation	MUX1 multiple selection	MUX2 multiple selection
2^3	2^2	2^1	2^0	2^{-1}			
b_{i+3}	b_{i+2}	b_{i+1}	b_i	b_{i-1}			
1	0	1	1	1	$-4A$	0	$-4A$
1	1	0	0	0	$-4A$	0	$-4A$
1	1	0	0	1	$-3A$	A	$-4A$
1	1	0	1	0	$-3A$	A	$-4A$
1	1	0	1	1	$-2A$	2A	$-4A$
1	1	1	0	0	$-2A$	$-2A$	0
1	1	1	0	1	$-A$	$-A$	0
1	1	1	1	0	$-A$	$-A$	0
1	1	1	1	1	-0	0	0

Table 5.7 Binary partial products selection according to Booth4 recoding.

Decimal set is divided into $(-2A, -A, A, 2A)$ and $(5A, 10A)$ groups as in previous designs, Table 5.5. Figure 5.16 shows the partial products selection block diagram for binary and decimal multiplications. MUXs1 output 17 partial products plus 1 partial product for sign bits, MUXs2 output 16 partial products plus 1 partial product for sign bits.

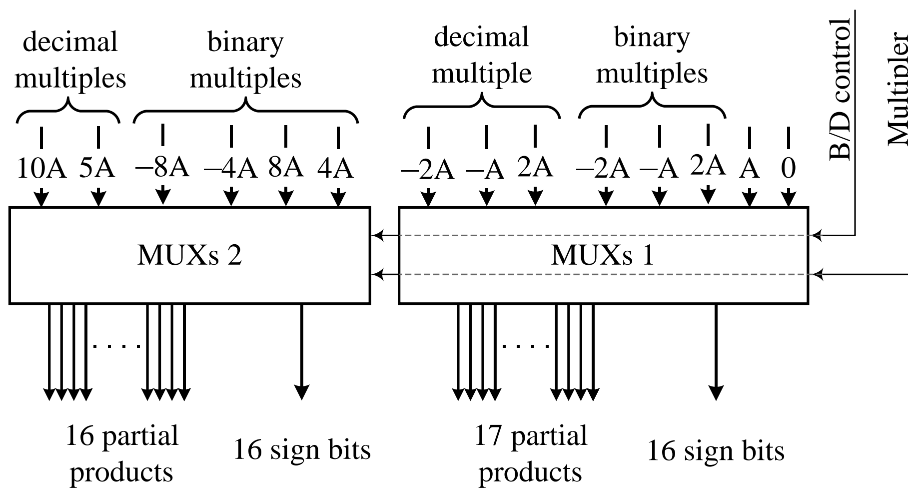


Figure 5.16 Partial products selection.

This stage outputs 33 binary/decimal partial products plus two sign partial products. A two level multiplexer design is used as shown in Figure 5.17.

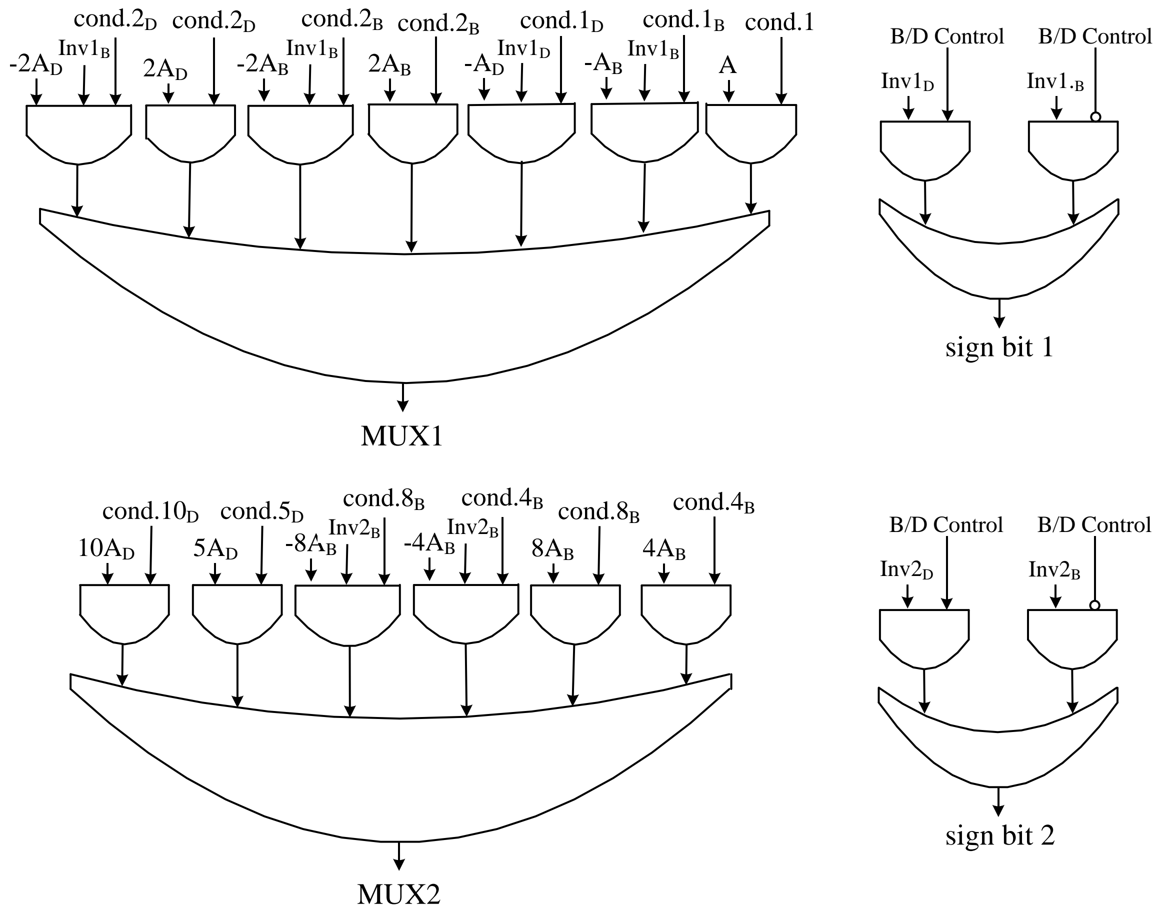


Figure 5.17 Multiplexers design for each multiplier digit.

The conditions that control the multiplexers depend on multiplier current digit $(b_3.b_2.b_1.b_0)$, most significant bit of previous multiplier digit (b_{-1}) and B/D Control signal (c), where c $= '0'$ for binary input operands and c $= '1'$ for decimal input operands. The equations that control the two multiplexers are

For MUXs1

$$cond1 = (b_0\,\overline{b_{-1}} + \overline{b_0}\,b_{-1}) \cdot \bar{c} + \left(\overline{b_0}\,b_2 + b_0\,\overline{b_1}\,\overline{b_2}\right) \cdot c \qquad (5.39)$$

$$cond2_B = \left(\overline{b_0}\,b_1\,\overline{b_{-1}} + b_0\,\overline{b_1}\,b_{-1}\right) \cdot \bar{c} \qquad (5.40)$$

$$cond2_D = \left(\overline{b_0}\,b_3 + b_0\,b_1 + b_1\,\overline{b_2}\right) \cdot c \qquad (5.41)$$

$$\mathrm{Inv1}_B = \left(\overline{b_0}\,b_1 + b_1\,\overline{b_{-1}}\right) \qquad (5.42)$$

$$\mathrm{Inv1}_D = (b_3 + b_0\,b_1\,\overline{b_2} + \overline{b_0}\,\overline{b_1}\,b_2) \qquad (5.43)$$

For MUXs2

$$cond4_B = \left(b_1\,\overline{b_2} + \overline{b_1}b_2\right) \cdot \bar{c} \qquad (5.44)$$

$$cond8_B = \left(\overline{b_1}\,\overline{b_2}\,b_3 + b_1\,b_2\,\overline{b_3}\right) \cdot \bar{c} \qquad (5.45)$$

$$cond5_D = \left(b_0\,b_1 + b_2\,\overline{b_3}\right) \cdot c \qquad (5.46)$$

$$cond10_D = (b_3) \cdot c \qquad (5.47)$$

$$\mathrm{Inv2}_B = (b_3) \qquad (5.48)$$

Sign bits outputs from MUX1 and MUX2, Inv1 and Inv2 signals, are added in column trees. It is entered to first bit of shifted carry digits.

5.3.3 Partial Products Accumulation Stage

This stage consists of four steps as shown in Figure 5.14. Binary column tree, rearrange column tree outputs, tiny binary/decimal tree, and final carry propagate adder.

Binary column tree

A CSA binary column tree is used for the 35 partial products. It is the first step of partial products accumulation for binary and decimal multiplication paths. Figure 5.18 shows the scheme of the proposed binary column tree. Columns output different size sums and carries according to number of digits added.

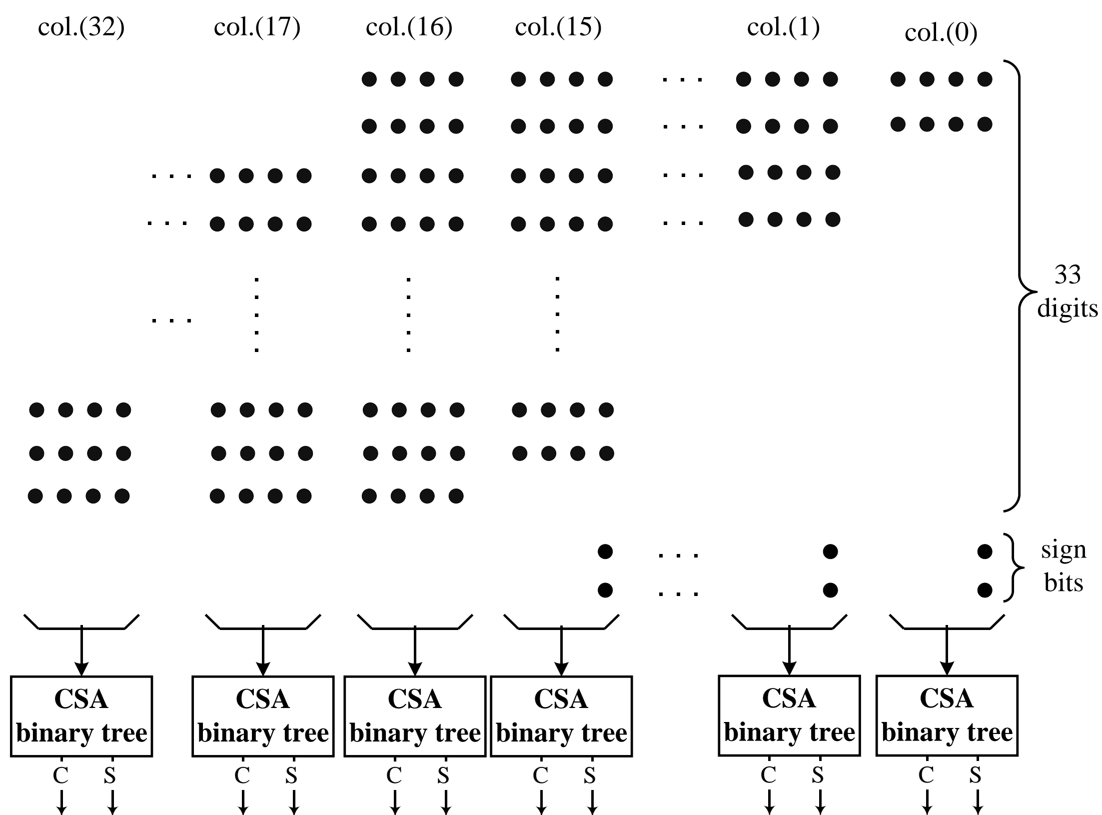


Figure 5.18 Proposed binary column tree scheme (S and C maximally 8 bits).

The worst case number of digits to be added is 33 plus 2 sign bits. Figure 5.19 shows the 33 digits CSA binary column tree. Sign bits added to the first bit of shifted carry digits in the tree levels.

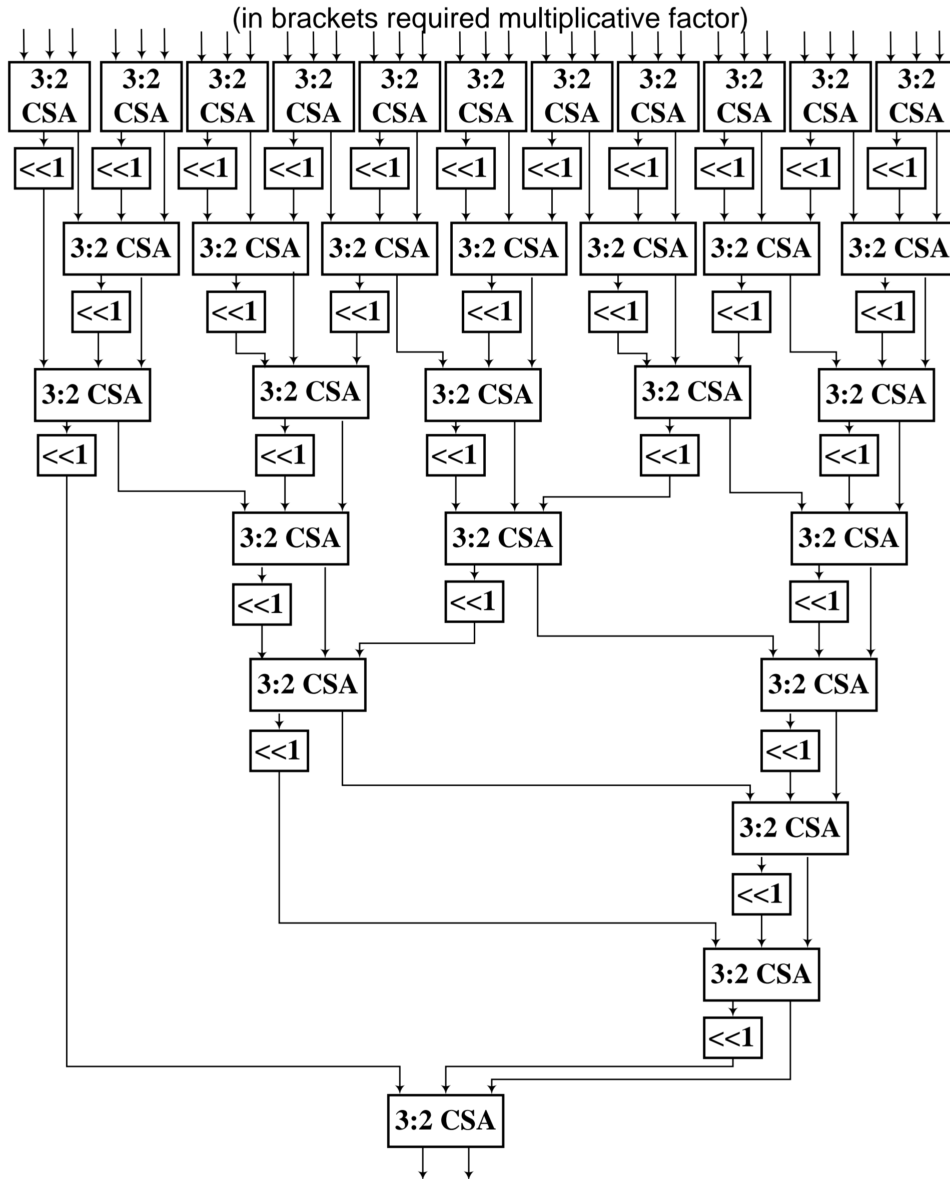


Figure 5.19 33 digits CSA binary tree.

Rearrange column tree outputs

For Binary operands, column trees output 4 bits to 8 bits sums and carries. Columns outputs are rearranged to multiply each one by its relevant weight. Four bit-vectors are produced, two for output sums and two for output carries, Figure 5.20.

32	31	30		8	7	6	5	4	3	2	1	0	digit no.
S(31)	S(31)	....		S(7)	S(7)	S(5)	S(5)	S(3)	S(3)	S(1)	S(1)	S(0)	S1
C(31)	C(31)	....		C(7)	C(7)	C(5)	C(5)	C(3)	C(3)	C(1)	C(1)	C(0)	C1
	S(30)	S(30)	....		S(6)	S(6)	S(4)	S(4)	S(2)	S(2)			S2
	C(30)	C(30)	....		C(6)	C(6)	C(4)	C(4)	C(2)	C(2)			C2

Figure 5.20 The four binary bit-vectors after rearranging.

For decimal operands, column trees output 4 bits, 1 BCD digit, to 10 bits, 3 BCD digits, sums and carries. A binary to decimal converters is used to convert each binary column output to BCD-8421, and then converted to BCD-4221, and then it is rearranged to multiply each column output by its relevant weight. After rearranging column trees output six bit-vectors, three for output sums and three for output carries, Figure 5.21.

32	31	30	29	28	27	26		9	8	7	6	5	4	3	2	1	0	digit no.
S(31)	S(31)	S(29)	S(29)	S(26)	S(26)	S(26)	....			S(5)	S(5)	S(5)	S(3)	S(3)	S(1)	S(1)	S(0)	S1
C(31)	C(31)	C(29)	C(29)	C(26)	C(26)	C(26)	....			C(5)	C(5)	C(5)	C(3)	C(3)	C(1)	C(1)	C(0)	C1
	S(30)	S(30)	S(27)	S(27)	S(27)	....			S(6)	S(6)	S(6)	S(4)	S(4)	S(2)	S(2)			S2
	C(30)	C(30)	C(27)	C(27)	C(27)	....			C(6)	C(6)	C(6)	C(4)	C(4)	C(2)	C(2)			C2
			S(28)	S(28)	....			S(7)	S(7)	S(7)								S3
			C(28)	C(28)	....			S(7)	S(7)	C(7)								C3

Figure 5.21 The six decimal bit-vectors after rearranging.

The conversion to decimal, BCD-8421, takes some delay. However, this extra decimal delay is approximately equal to decimal $\times 2$ blocks delay used in decimal path in Hickmann design[9]. But it is separated from binary path without significant increase in area. Where the area of binary to decimal converters is much less than the area of another tree as in [9]. The conversion from binary to decimal, BCD-8421, is discussed in next section. the decimal bit-vectors is converted from BCD-8421 to BCD-4221 before the tiny tree to allow the use of binary CSA design and out valid decimal values for all 4-bit combinations. So we need not a decimal correction after the addition.

The BCD-8421($x_3x_2x_1x_0$) to BCD-4221($h_3h_2h_1h_0$) converter equations are

$$h_0 = x_0 \quad (5.49)$$

$$h_1 = x_3 \quad (5.50)$$

$$h_2 = x_1 + x_3 \quad (5.51)$$

$$h_3 = x_2 + x_3 \quad (5.52)$$

Binary to BCD-8421 Conversion (Shift and Add-3 Algorithm) [1]

The algorithm starts with the three most significant bits of the binary number. If the three bits value is greater than or equal to five, add binary three to the number and shift the result one bit to the left. If the three bits value is less than five, shift to the left without adding. Then take the next bit from the right and repeat the operation till we reach the least significant bit. Figure 5.22 shows the steps to convert 8-bit binary number to BCD-8421 using the Shift and Add-3 Algorithm [1]. The steps of this example:

1. Shift the binary number left three bits.

2. If the binary value in any of the BCD columns is 5 or greater, add 3 to that value in that BCD column.

3. Shift the binary number left one bits

4. Go to 2.

5. After 8 shifts, the BCD number is obtained**.**

BCD Digits				Binary	
Operation	Hundreds	Tens	Units		
Start				1 1 1 1	1 1 1 1
Shift 1			1	1 1 1 1	1 1 1
Shift 2			1 1	1 1 1 1	1 1
Shift 3			1 1 1	1 1 1 1	1
Add 3			1 0 1 0	1 1 1 1	1
Shift 4		1	0 1 0 1	1 1 1 1	
Add 3		1	1 0 0 0	1 1 1 1	
Shift 5		1 1	0 0 0 1	1 1 1	
Shift 6		1 1 0	0 0 1 1	1 1	
Add 3		1 0 0 1	0 0 1 1	1 1	
Shift 7	1	0 0 1 0	0 1 1 1	1	
Add 3	1	0 0 1 0	1 0 1 0	1	
Shift 8	1 0	0 1 0 1	0 1 0 1		
BCD	2	5	5		

Figure 5.22 8-bit binary to decimal converter example.

Figure 5.23 shows the block diagram of the 8-bit binary to BCD-8421 converter using this algorithm.

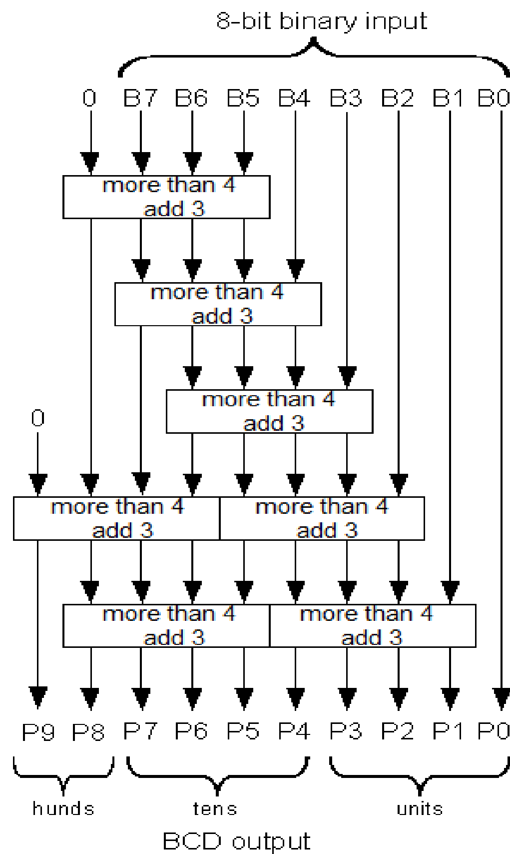


Figure 5.23 8-bit binary to decimal converter block diagram.

In the final proposed design a binary to BCD-8421 converters are used for each column output of the binary column tree. A 5-bit to 9-bit binary to BCD-8421 converters are used for different outputs. The 9-bit one has a largest delay. It adds 33 decimal digit plus 2 sign bits. Its maximum value of it is 299_D, 100101011_B. Its most significant three bits are always less than five so its first level can be eliminated. Figure 5.24 shows the binary to decimal converter used for 9-bit binary input.

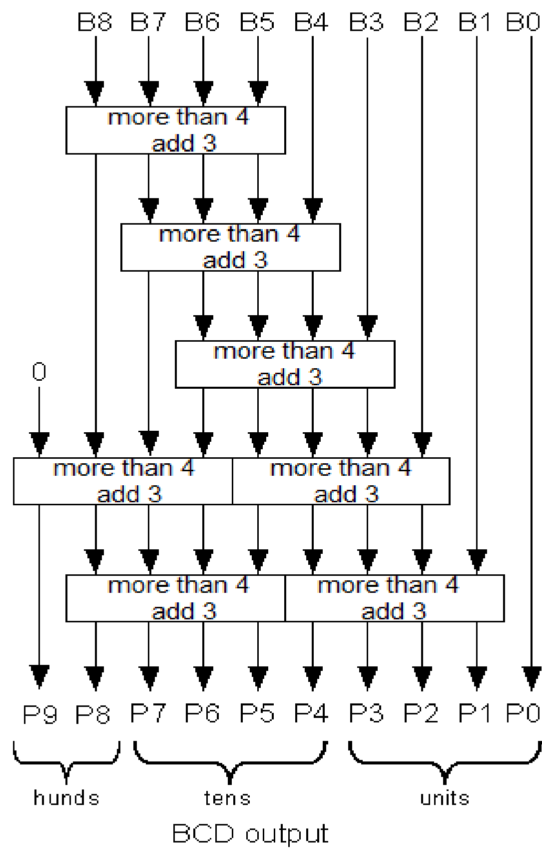


Figure 5.24 9-bit, max. value 319, binary to decimal converter block diagram.

Tiny Binary/Decimal Tree

Two designs are implemented for this step, which add the 4/6 bit-vectors output from the column tree for binary/decimal multiplication. A tiny split binary/decimal tree design is obtained to eliminate the latency of the decimal ($\times 2$) from the binary multiplication path, Figure 5.25. A tiny shared binary/decimal tree design is obtained to decrease the area, Figure 5.26, where one 128-bit CSA is shared between binary and decimal, using multiplexer.

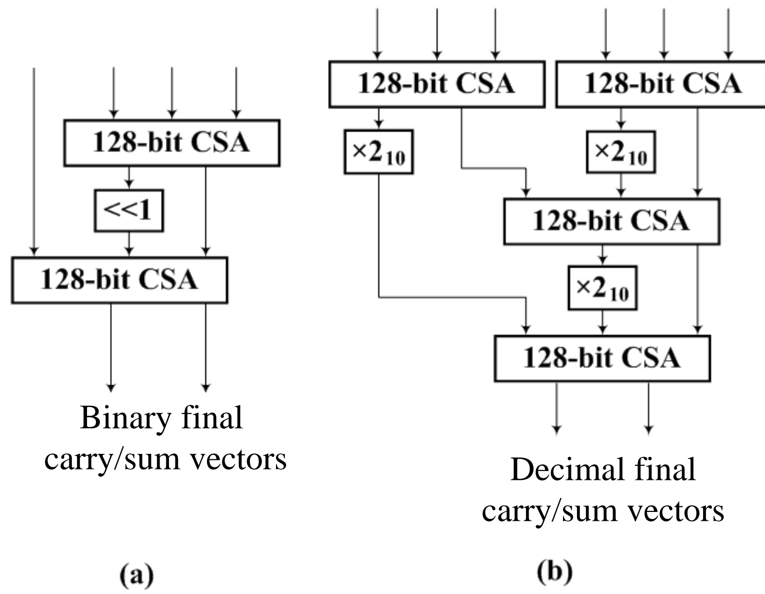


Figure 5.25 (a) Tiny split binary tree. (b) Tiny split decimal tree.

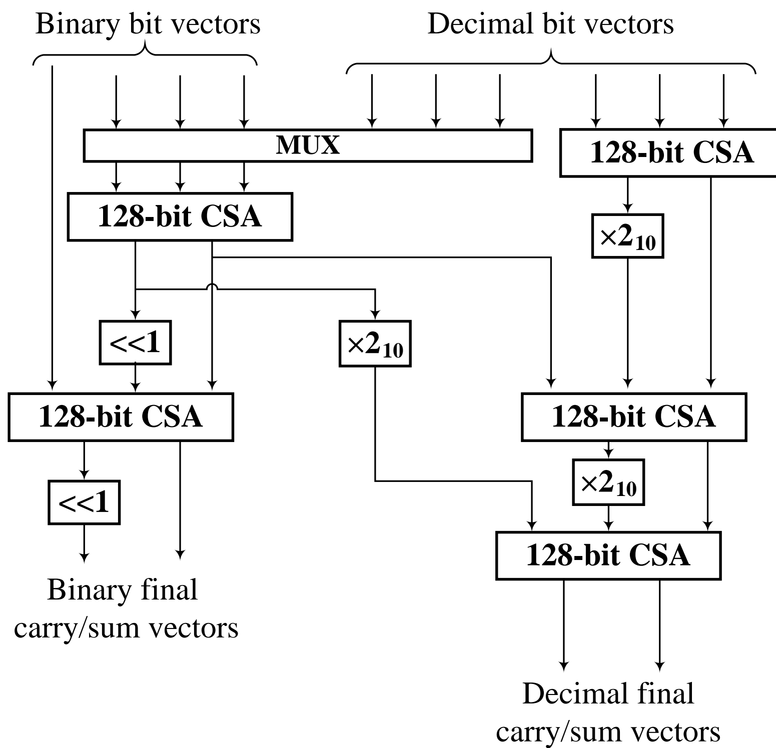


Figure 5.26 Tiny shared binary/decimal tree

It was found that the difference in area between the split and shared tiny binary/decimal trees is not large. The area of shared design is slightly less than split design, where one 128-bit CSA is replaced by a three 128-bit MUXs for three binary/decimal bit vectors.

Final Carry Propagate Adder

The final carry/sum vectors are added using Kogge-Stone based carry propagate adder [11]. Two designs are implemented, shared and split binary/decimal Kogge-Stone based carry propagate adder. Figure 5.27 shows the proposed split binary/decimal Kogge-Stone based carry propagate adder scheme. In this scheme binary and decimal carry propagate adders are separated to decrease the delay of binary and decimal multiplication paths.

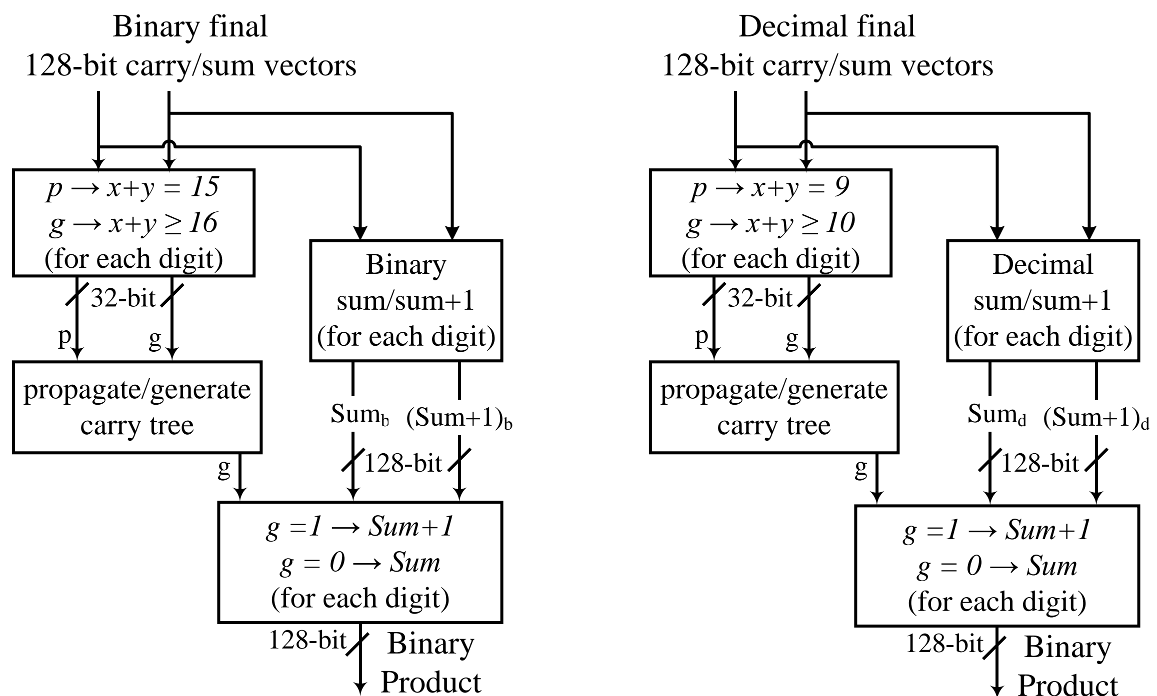


Figure 5.27 Split Binary/Decimal Kogge-Stone based carry propagate adder

Another scheme is implemented to decrease the area, novel shared binary/decimal Kogge-Stone based carry propagate adder, Figure 5.28.

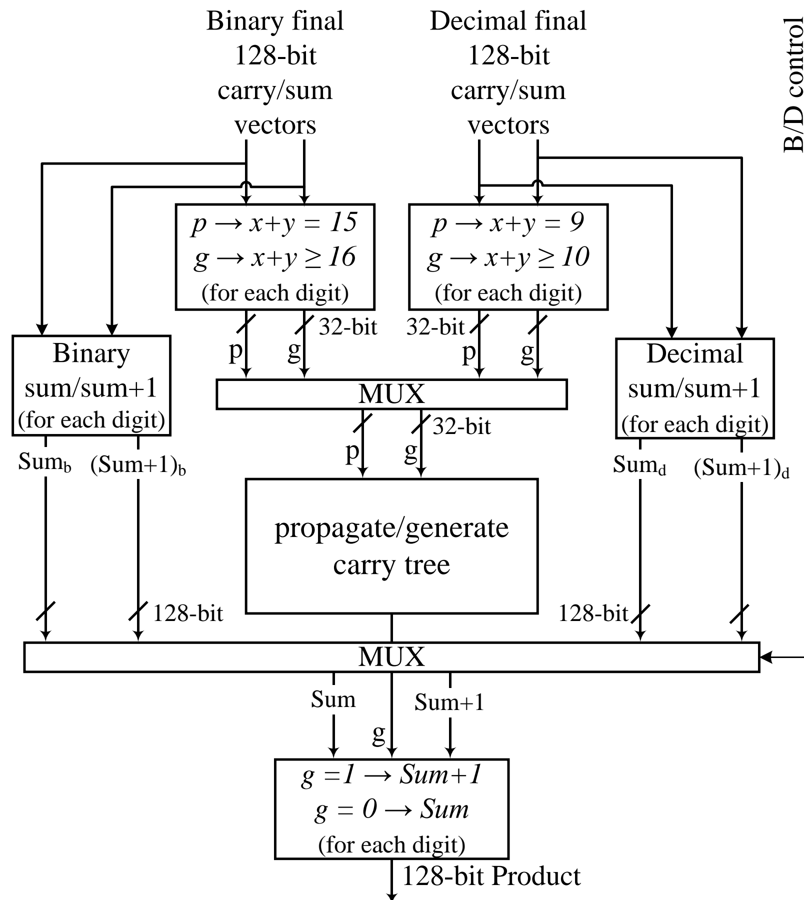


Figure 5.28 Shared Binary/Decimal Kogge-Stone based carry propagate adder

This permits the use of the same generate/propagate carry tree for decimal and binary multiplications. Only first level of Kogge-Stone adder is split for binary and decimal. This level outputs the first propagate and generate signals for each digit. The remaining levels are shared for decimal and binary. Parallel to these levels, the decimal and binary sum and sum + 1 for every digit are generated. After the generate/propagate carry tree are finished, the correct sum according to last level carry and B/D control signal is chosen. A BCD-4221 format is used

to eliminate decimal correction delay and area. At the end of design a BCD-4221 to BCD-8421 conversion is performed to produce the final product in BCD-8421format. The equations of the conversion from BCD-4221($x_3x_2x_1x_0$) to BCD-8421($h_3h_2h_1h_0$) are

$$h_0 = x_0 \quad (5.53)$$

$$h_1 = \overline{x_1}.x_2 + x_1.\overline{x_2} \quad (5.54)$$

$$h_2 = \overline{x_1}.x_3 + \overline{x_2}.x_3 + x_1.x_2.\overline{x_3} \quad (5.55)$$

$$h_3 = x_1.x_2.x_3 \quad (5.56)$$

5.4 Final Proposed Design

Figure 5.29 shows the final proposed binary/decimal multiplier design. It is similar to third proposed combined binary/decimal multiplier discussed with a carry/sum addition block before the rearranging of decimal operands output from column tree.

This block adds every sum/carry output from binary column tree using binary Kogge-Stone CPAs, maximum 8 bit. After sum/carry addition, tree columns output 5 binary bits, 2 BCD digits, to 9 binary bits, 3 BCD digits, sums and carries. A binary to decimal converters based on [1] is used to convert each binary column output to BCD-8421, and then converted to BCD-4221, and then it is rearranged to multiply each column output BCD digits by its relevant weight. Figure 5.30 shows the three bit-vectors output after rearranging.

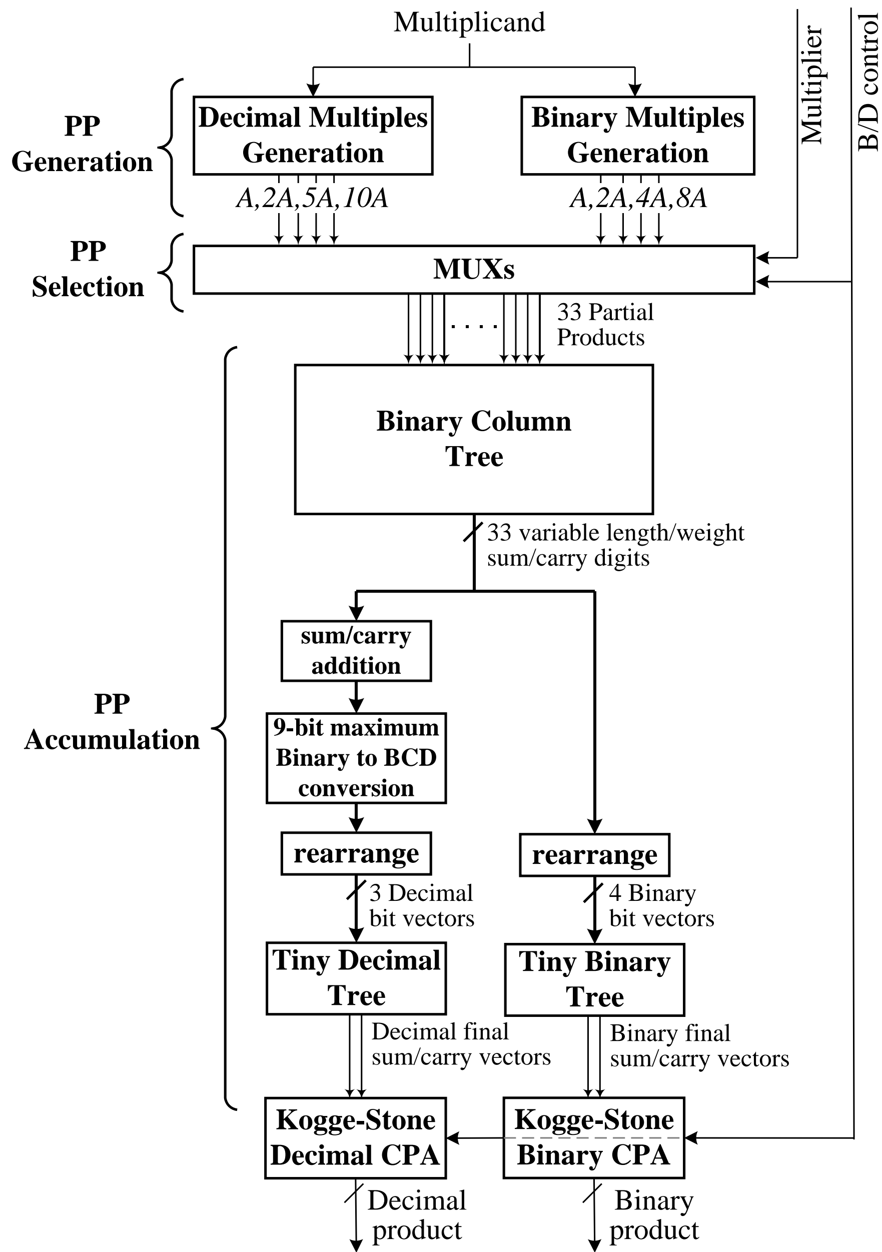


Figure 5.29 Final proposed Binary/Decimal Multiplier design (split scheme).

32	31	30	29	28	27				11	10	9	8	7	6	5	4	3	2	1	0	digit no.
S(31)	S(31)	S(29)	S(29)	S(27)	S(27)	....			S(9)	S(9)	S(9)	S(6)	S(6)	S(6)	S(4)	S(4)	S(2)	S(2)	S(0)	S(0)	S1
	S(30)	S(30)		S(26)	S(26)	S(26)	....			S(8)	S(8)	S(8)	S(5)	S(5)	S(5)	S(3)	S(3)	S(1)	S(1)		S2
			S(28)	S(28)	S(25)	S(25)	S(25)	....			S(7)	S(7)	S(7)								S3

Figure 5.30 The three decimal bit-vectors after rearranging.

Two schemes are used for the binary and decimal tiny tree, Split and shared, to add the four and three bit-vectors output from the column tree, respectively, Figure 5.31 and Figure 5.32.

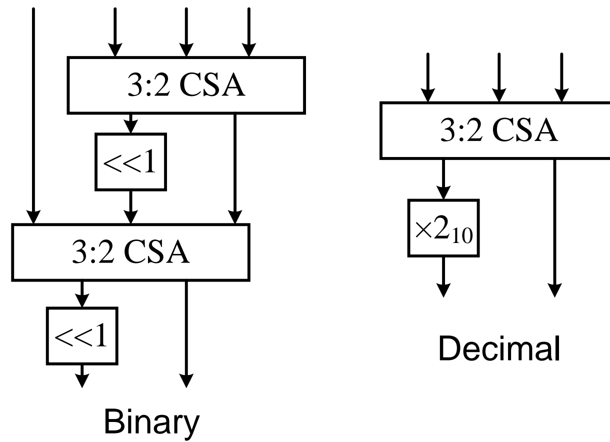


Figure 5.31 Tiny split binary and decimal trees.

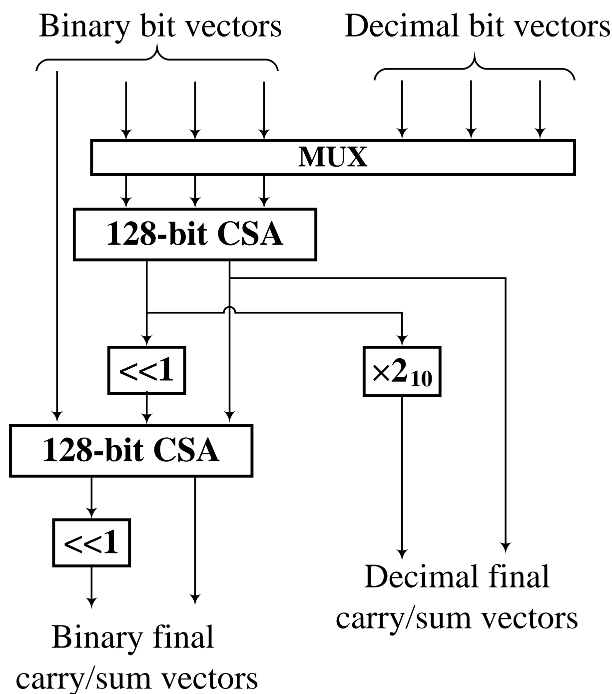


Figure 5.32 Tiny shared binary/decimal tree.

The two final vectors output from tiny binary/decimal tree are added using binary/decimal Kogge-Stone based carry propagate adder, Figure 5.27, and Figure 5.28.

5.5 Conclusion

Four designs are proposed for the combined binary/decimal multiplier. The final proposed design groups all good ideas for the previous proposals within this thesis. We use the smallest area and delay binary multiplicand multiples generation method, booth4 recoding. We also include the decimal SD radix-5 recoding for decimal multiplicand multiples generation from Vazquez, which has a small area and delay. Dadda binary column tree is included to add the binary/decimal partial products. It saves the use of decimal adders and corrections which increase the area and delay of the multiplier. Two schemes are used for the addition of the three/four bit vectors output from column tree, shared and split. The shared one tries to decrease the area of the multiplier, and the split one tries to decrease the delay of the multiplier. Next we discuss the testing of our designs and the implementation results.

Chapter 6

Verification and Results

6.1 Testing

The proposed combined binary/decimal multiplier, Vázquez, and Hickmann designs are implemented in FPGAadv tool. In order to verify the implementations, test cases are generated then a test bench is developed for the implemented design using VHDL language. A C program is written equivalent to multiplier design. Using ActiveFileCompare program, the test bench results and the C program results is compared.

Test Cases

The multiplier has two 64-bit inputs, multiplicand and multiplier, and a control bit. It is difficult to test all possible combinations of the inputs where 2^{128} = 3.4e38. So a group of test cases is generated trying to handle all possible errors. We divided the 128-bit inputs to 8 parts, each one 16-bit. Table 6.1 shows the test cases used. (i.e. we use C character to represent a truth table of all possible combination of the 16-bit, from 0000000000000000 to 1111111111111111, 2^{16} row). First part of test cases shown in Table 6.1 represents a binary 8-bit truth table, but instead of '1' logic, a C is used. So each row of the truth table represents 2^{16} rows. In the second and last part of the

table the C truth table is used to test each part when the other bits are 0 and 1 respectively.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	C
0	0	0	0	0	0	C	0
0	0	0	0	0	0	C	C
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
C	C	C	C	C	C	0	0
C	C	C	C	C	C	0	C
C	C	C	C	C	C	C	0
C	C	C	C	C	C	C	C
0	0	0	0	0	0	0	C
0	0	0	0	0	0	C	0
.	.	.	.	.	.	.	.
C	0	0	0	0	0	0	0
1	1	1	1	1	1	1	C
.	.	.	.	.	.	.	.
1	C	1	1	1	1	1	1
C	1	1	1	1	1	1	1

Table 6.1 Test cases

Testing Approach

A test bench is developed for each block of the multiplier then a test bench for the whole multiplier is implemented. A C program is written for each block

and for the multiplier equivalent to VHDL design. ActiveFileCompare program is used to compare between test benches results and C programs results.

6.2 Results

This section presents an area/delay comparison between the proposed binary/decimal multiplier designs and the two previous ones [9][22]. The main difference between them is the accumulation stage. Vázquez et al. propose a shared binary/decimal CSA tree using binary/BCD-4221 format. The CSAs for binary and decimal are shared. A MUX is used in each $\times 2$ block to select between binary and decimal. They lead to some increase in the delay of binary and decimal paths. Hickmann et al. split the binary and decimal CSA trees at the start of using $\times 2$ blocks. They have an increase in area but the decimal $\times 2$ and binary $\times 2$ is separated, so no MUXs are used. A comparison between these designs and third design, shared and split, and final proposed design, shared and split is introduced.

Third proposed design

The proposed designs use two tree stages. The first is a binary tree used to decrease the binary/decimal partial products to four/six bit-vectors. Then a tiny split/shared CSA tree is used to add the four/six bit-vectors for binary/decimal multiplication. Table 7.1 presents area-delay figure for the different binary/decimal multipliers on FPGA virtex5. The area of the two proposed designs, split and shared, is less than Vázquez and Hickmann designs.

		Vázquez	Hickmann	Proposed Design (split)	Proposed Design (shared)
Worst path delay	Binary	≈ 55 ns	≈ 42 ns	≈ 43 ns	≈ 56 ns
	Decimal		≈ 48 ns	≈ 51 ns	
Total equivalent gate count		116081	108472	92658	92098

Table 7.1 Area/Delay figure for different Binary/decimal multipliers using FPGA virtex5.

Final proposed design

In the final proposed design, a carry/sum addition of the column outputs from binary column tree is included before rearranging, to decrease the delay of binary and decimal paths. Table 7.2 shows an area-delay figure for the different binary/decimal multipliers on FPGA virtex5.

		Vázquez	Hickmann	Proposed Design (split)	Proposed Design (shared)
Worst path delay	Binary	≈ 55 ns	≈ 42 ns	≈ 43 ns	≈ 54 ns
	Decimal		≈ 48 ns	≈ 48 ns	
Total equivalent gate count		116081	108472	87605	88613

Table 7.2 Area/Delay figure for different Binary/decimal multipliers using FPGA virtex5.

Also each design is synthesized on the low power CMOS 130nm technology. Table 7.3 shows an area/delay figure for the different binary/decimal multipliers [15].

<table>
<tr><th colspan="2"></th><th>Vázquez</th><th>Hickmann</th><th>Proposed Design (split)</th><th>Proposed Design (shared)</th></tr>
<tr><td rowspan="2">Worst path delay (ns)</td><td>Binary</td><td rowspan="2">8.03</td><td>4.55</td><td>4.61</td><td rowspan="2">7.86</td></tr>
<tr><td>Decimal</td><td>7.18</td><td>7.05</td></tr>
<tr><td colspan="2">Area (µm²)</td><td>932044</td><td>782643</td><td>567681</td><td>596330</td></tr>
</table>

Table 7.3 Area/Delay figure for different binary/decimal multipliers using ASIC low power CMOS 130nm technology.

The proposed design has almost the same delays as the fastest known multiplier, Hickmann multiplier design, but significantly reduces the required area.

6.3 Conclusion

This thesis presents a parallel combined binary/decimal fixed-point multiplier design with novel partial product accumulation design to decrease the area of the multiplication without increasing the delay. In this stage, a binary column tree is used for binary and decimal multiplication. Then a tiny binary/decimal CSA tree is used to generate the final product in sum/carry format. A comparison between proposed and previous binary/decimal multipliers shows that the proposed design has the smallest area. It is 16% less than Vázquez design area and 27.5% less than Hickmann design area. For the delay, the proposed design is almost the same as Hickmann multiplier (fastest in the literature) and less than Vázquez multiplier for decimal and binary paths.

6.4 Future Work

Finally, this section presents some suggestions for future work. These suggestions are as follows:

- Design a floating point binary/decimal multiplier.

- Implement a combined binary/decimal floating point adder and a combined floating point binary/decimal divider then include them into a processor.

References

[1] Benedek, M., “Developing Large Binary to BCD Conversion Structures”, IEEE Transactions on Computers, vol. C-26, pages: 688 – 700, July 1977.

[2] Cowlishaw, M. F., “Decimal Floating-Point: Algorism for Computers”, IEEE 3rd Symposium on Computer Arithmetic, 2003.

[3] Dadda, L., “Multioperand Parallel Decimal Adder: A Mixed Binary and BCD Approach”, IEEE Transactions on Computers, Volume 56, No. 10, October 2007.

[4] Duale, A.Y. et al, “Decimal floating-point in z9: An implementation and testing perspective”, IBM Journal of Research and Development , vol. 51, pages: 217-227, 2007.

[5] Ercegovac, M. D., and Lang, T., “Digital Arithmetic”, Morgan Kaufmann Publishers, 2004.

[6] (a) Erle, M. A. and Schulte, M. J., “Decimal Multiplication Via Carry-Save Addition”, Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors, June 2003.

[6] (b) Erle, M. A. et al., “Decimal Multiplication With Efficient Partial Product Generation”, Proceedings of IEEE Symposium on Computer Arithmetic, June 2005.

[7] Fahmy, H. et al, “Computer Arithmetic”, Not Yet Published.

[8] Flynn, M. J. and Oberman, S. F., “Advanced Computer Arithmetic Design”, Wiley, John & sons Incorporated, April 2001.

[9] Hickmann, B. et al., “Improved Combined Binary/Decimal Fixed-Point Multipliers”, IEEE International Conference on Computer Design, October 2008.

[10] Kenney, R. D. et al., “A High-Frequency Decimal Multiplier”, Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors, October 2004.

[11] Kogge, P. M. and Stone, H. S., “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations”, IEEE Transactions on Computers, August 1973.

[12] Koren, I., “Computer arithmetic algorithms”, A K Peters, Ltd., 2002.

[13] Lang, T. and Nannarelli, A., “A Radix-10 Combinational Multiplier”, Proceeding in Asilomar Conference on Signals, Systems and Computers, November 2006.

[14] Lu, M., “Arithmetic and logic in computer systems”, John Wiley and Sons, 2004.

[15] Mahmoud, M. and Fahmy, H., “A Parallel Combined Binary/Decimal Fixed-Point Multiplier with Binary Partial Products Reduction Tree”, 21st International Conference on Computer Theory and Applications, October 2011

[16] Nicoud, J.D., “Iterative Arrays ror Radix Conversion”, IEEE Transactions on Computers, December 1971.

[17] Ohtsuki, T. et al., “Apparatus for Decimal Multiplication”, United States Patent, no. 4677583, June 1987.

[18] Palmam, B., “High Performance Computing for Computational Science”, Springer, 2003.

[19] Parhami, B., “Computer Arithmetic: Algorithms and Hardware Designs”, Oxford University Press, USA, September 1999.

[20] Richards, R. K., “Arithmetic Operations in Digital Computers”, Van Nostrand Company, 1955.

[21] Schmookler, M. S. and Weinberger, A. W., “High Speed Decimal Addition”, IEEE Transactions on Computers, Volume C-20, Issue 8, pages 862-866, August 1971.

[22] Vázquez, A. et al., “A New Family of High–Performance Parallel Decimal Multipliers”, Proceedings of IEEE Symposium on Computer Arithmetic, June 2007

[23] http://speleotrove.com/decimal/decifaq1.html#emphasis

نبذة

أصبحت الحسابات بالنظام المشترك الثنائي/العشري في الأجهزة موضوعا هاما لدعم التطبيقات العشرية والثنائية بسرعة عالية ومساحة صغيرة. هذه الرسالة تقدم تصميم لمضاعف ثنائي/عشري للأعداد الصحيحة. وبما أن مرحلة جمع النواتج الجزئية في المضاعف تأخذ أكبر مساحة وأطول وقت ، فهي المرحلة الأكثر أهمية. لذلك تم استخدام النظام الثنائي لجمع النواتج الجزئية الناتجة في المضاعف بالنظام الثنائي والعشري. ايضا تم مقارنة التصميم المقترح مع التصاميم المنشورة سابقا. التصميم المقترح انتج انخفاضا كبيرا في المساحة مع نفس السرعه تقريبا للتصميم الأسرع في التصاميم المعروفة سابقا.

هيكل الرسالة يتكون على النحو التالي. الفصل الاول يعرض مقدمة عن النظم الحسابية الثنائية والعشرية. الفصل الثاني يلخص الطرق المستخدمه لعمليات الضرب. الفصل الثالث يعرض تقنيات الضرب بالنظام العشري. الفصل الرابع يركز على المضاعفات الثنائية/العشرية المقترحة سابقا. الفصل الخامس يشرح التصميم المقترح لتحقيق المضاعف الثنائي/العشري. الفصل السادس يتناول المقارنة بين التصميم المقترح والتصاميم السابقة. وأخيرا نقدم الاستنتاجات والعمل المستقبلي.

مضاعف ثنائي/عشري للأعداد الصحيحة
باستخدام النظام الثنائي في جمع النواتج الجزئية

إعداد

مرفت محمد عادل محمود

رسالة مقدمة إلى كلية الهندسة ، جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
في الألكترونيات والأتصالات الكهربية

تحت إشراف

أمين نصار

أستاذ
قسم الأتصالات والألكترونيات الهندسية
جامعة القاهرة

حسام فهمي

أستاذ مساعد
قسم الأتصالات والألكترونيات الهندسية
جامعة القاهرة

كلية الهندسة ، جامعة القاهرة

الجيزة ، جمهورية مصر العربية

2011

مضاعف ثنائي/عشري للأعداد الصحيحة
باستخدام النظام الثنائي في جمع النواتج الجزئية

إعداد

مرفت محمد عادل محمود

رسالة مقدمة إلى كلية الهندسة ، جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
في الألكترونيات والأتصالات الكهربية

كلية الهندسة ، جامعة القاهرة

الجيزة ، جمهورية مصر العربية

2011