ADAPTIVE AND HYBRID SCHEMES FOR EFFICIENT PARALLEL SQUARING AND CUBING UNITS

By

Son Viet Bui

Bachelor of Science in Electrical Engineering
Hanoi University of Science and Technology
Hanoi, Vietnam
2000

Master of Science in Electrical Engineering
Hanoi University of Science and Technology
Hanoi, Vietnam
2002

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 2014

ADAPTIVE AND HYBRID SCHEMES FOR EFFICIENT PARALLEL SQUARING AND CUBING UNITS

Dissertation Approved:

Dr. James E. Stine Jr.

Dissertation Advisor

Dr. Keith A. Teague

Dr. Qi Cheng

Dr. Jaeun Ku

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr. James E. Stine for his fundamental role in my Ph.D study and research. His patience, motivation, and immense knowledge helped me in all the time of research and writing this thesis.

My sincere thanks also goes to all committee members Dr. Keith A. Teague, Dr. Qi Cheng, and Dr. Jaeun Ku. Thank you all for everything you have done to help through

I am deeply thankful to my wife and my children for their love, support, and sacrifices. Without them, this thesis would never have been written.

Last but not the least, I would like to thank my parents for giving birth to me and supporting me spiritually throughout my life.

Acknowledgements reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: SON VIET BUI

Date of Degree: December, 2014

Title of Study: **ADAPTIVE AND HYBRID SCHEMES FOR EFFICIENT PARALLEL SQUARING AND CUBING UNITS**

Major Field: Electrical and Computer Engineering

Squaring (X^2) and cubing (X^3) units are special operations of multiplication used in many application, such as image compression, equalization, decoding and demodulation, 3D graphics, scientific computing, artificial neural networks, logarithmic number system, and multimedia application. They can also be an efficient way to compute other basic functions. Therefore, improving their performances is a goal for many researchers. This dissertation will discuss modification to algorithms to compute parallel squaring and cubing units in both signed and unsigned representation. After that, truncated technique is applied to improve their performance. Each unit is modeled and estimated to obtain its area, delay by using linear evaluation model. A C program was written to generate Hardware Description Language files for each unit. These units are simulated and verified in simulation. Moreover, area, delay, and power consumption are calculated for each unit and compared with those ones in previous approaches for both Virtex 5 Xilinx FPGA and IBM 65nm ASIC technologies.

TABLE OF CONTENTS

Chapter Page

1 Introduction **2**

1.1 Motivation 2

1.2 Research Contributions 4

1.3 Dissertation Organized 5

2 Multiplication **6**

2.1 Carry Save Concept 9

2.2 Carry-Save Array Multipliers (CSAM) 10

2.3 Tree Multipliers 13

2.3.1 Wallace Tree Multipliers 14

2.3.2 Dadda Tree Multipliers 17

2.3.3 Reduced Area (RA) Multipliers 18

2.4 Baugh and Wooley Multiplication 20

2.5 Modified Booth Multipliers with Radix 4 24

3 Truncated Multipliers **28**

3.1 Truncated Multiplication with Correction Constant (CCT) 31

3.2 Truncated Multiplication with Variable Correction (VCT) 37

3.3 Truncated Multiplication with Hybrid Correction(HCT) 41

4 Squaring Unit Architectures **45**

4.1 Unsigned Squaring Unit Architectures 45

4.1.1 Folded Squaring Units 45

4.1.2 Merged Squaring Units 46
4.1.3 Divide and Conquer Squaring Unit 49
4.1.4 Proposed Squaring Unit 50
4.2 Signed Squaring Unit Architectures 55
4.2.1 Booth Folding Squaring Unit 55
4.2.2 Booth 2 Left-to-Right Dual Recoding Squaring Unit 59

5 Cubing Units **62**
5.1 Liddicoat and Flynn Cubing Units 65
5.2 Stine and Blank Cubing Units 67
5.3 Divide and Conquer Cubing Units 68
5.4 Proposed Cubing Unit. 71

6 Truncated Squaring and Cubing Units **74**
6.1 Truncated Squaring Units 74
6.1.1 Truncated Squaring Units using Booth 2 Folding Technique . 74
6.1.2 Truncated Squaring Unit using Booth 2 Left-to-Right Encoding Technique. 76
6.1.3 Truncated Squaring Unit using Divide and Conquer Technique. 78
6.1.4 Proposed Truncated Squaring Unit 79
6.2 Truncated Cubing Units 81
6.2.1 Truncated Cubing Unit using Divide and Conquer Technique. 81
6.2.2 Proposed Truncated Cubing Unit 83
6.3 Errors Comparison 88

7 Area and Delay Analyses **93**
7.1 Area Estimation of Squaring Units. 94
7.1.1 Folded Squaring Unit. 94
7.1.2 Merged Squaring Unit. 95

7.1.3 Divide & Conquer Squaring Unit 96
7.1.4 Proposed Squaring Unit. 97
7.1.5 Booth 2 Folding Squaring Unit. 98
7.1.6 Booth 2 Left-to-Right Recoding 99
7.2 Delay Estimation of Squaring Units 100
7.3 Area Estimation of Cubing Units 103
7.3.1 Liddicoat and Flynn Method. 103
7.3.2 Stine and Blank method. 105
7.3.3 Divide and Conquer method. 106
7.3.4 Proposed method. 110
7.4 Delay Estimation of Cubing Units 112

8 Hardware Implementation and Conclusions 113

8.1 Hardware Implementation 113
8.1.1 Implementation on FPGA Technology 113
8.1.2 Implementation on ASIC Technology 115
8.2 Summary and Future Work 116

REFERENCES 124

LIST OF TABLES

Table Page

2.1 Partial Product Selection 25

4.1 Truth Table of Stage 1 51

4.2 Truth Table of A Special Adder in Stage 2 52

4.3 Truth Table of Stage 3 54

4.4 Truth Table for Partial Product P_i in Booth 2 Folding Technique . . 58

4.5 Truth Table of P_i in Left-to-Right Dual Recoding Method 60

6.1 Errors Comparison for Truncated Squaring Units in [11] 88

6.2 Errors Comparison for Truncated Squaring Units in [74] 89

6.3 Errors Comparison for Truncated Squaring Units in [71] 90

6.4 Errors Comparison for Truncated Squaring Units in [77] 90

6.5 Rounding to Nearest Even (RNE) Errors of Squaring Unit 91

6.6 Errors Comparison for Truncated Cubing Unit in [75] 91

6.7 Errors Comparison for Truncated Cubing Unit in [76] 91

6.8 Error Comparison of Rounding to Nearest Even(RNE) 92

8.1 Area, Delay, and Power Comparison of Squaring Units on Virtex 5 FPGA 118

8.2 Performances Truncated Squaring Units on FPGA Virtex 5. 119

8.3 Gates, Delay, and Power Comparison of Cubing Units on FPGA Virtex 5 120

8.4 Comparison for Truncated Cubing Units in Virtex 5 FPGA 120

8.5 Area, Delay, and Power Comparison of Squaring Units on $ASIC\ IBM$ $65nm$ 121

8.6 Performances Truncated Squaring Units on ASIC IBM $65nm$. 122
8.7 Gates, Delay, and Power Comparison of Cubing Units in ASIC IBM $65nm$ 123
8.8 Comparison of Truncated Cubing Units on ASIC 123

LIST OF FIGURES

Figure Page

2.1 Partial Products Matrix of Multiplication. 7

2.2 $8x8$-bit Dot Matrix Diagram of Multiplication. 8

2.3 A Carry-Save Adder or (3, 2) Counter. 9

2.4 An Example of Carry-Save Concept Adder. 11

2.5 A Four Operands 4-bit Multi-Operand Adder. 11

2.6 8×8-bit Carry-Save Array Multiplier. 12

2.7 Modified Half Adder and Full Adder. 13

2.8 (2, 2) and (3, 2) Counters in Tree Multiplier. 14

2.9 Wallace Tree Multiplier. 16

2.10 Dadda Tree Multiplier. 19

2.11 Reduce Area Multiplier. 21

2.12 PPM of Baugh and Wooley Multiplier. 22

2.13 Implementation of Baugh and Wooley Multiplier. 23

2.14 PPM of Booth 2 Multiplier. 26

2.15 Sign Extension Bit of MBP. 26

2.16 Partial Product Generation of Booth 2. 27

3.1 Product Matrix of Truncated Multiplier 31

3.2 PPM of 8×8-bit Truncated Multiplier in CCT with $k = 2$. 34

3.3 8×8-bit CCT with $k = 2$ in CSAM Implementation. 35

3.4 Special Half Adder. 36

3.5 8×8-bit CCT with $k = 2$ in Dadda Tree Reduction. 36

3.6 PPM of 8×8-bit VCT with $k = 2$. 38
3.7 8×8-bit VCT with $k = 2$ in CSAM Implementation. 39
3.8 8×8-bit VCT with $k = 2$ in Dadda Tree Reduction. 40
3.9 8×8-bit HCT with $k = 2$, $p = 0.5$ in CSAM Implementation. 43
3.10 8×8-bit HCT with $k = 2$, $p = 0.5$ in Dadda Tree Implementation. . . 44

4.1 PPM of Folded Squaring Unit with $n = 7$ 47
4.2 PPM of Folded Squaring Unit with $n = 8$ 48
4.3 PPM of Merged Squaring Unit with $n = 7$ 49
4.4 PPM of Merged Squaring Unit with $n = 8$ 49
4.5 PPM of a Squaring Unit in Divide and Conquer Method 50
4.6 PPM of 4-bit Squaring Unit in Divide and Conquer Method 51
4.7 Optimization of a Squaring Unit at Stage 1 53
4.8 Optimization of a Squaring Unit at Stage 2 53
4.9 Optimization of a Squaring Unit at Stage 3 55
4.10 PPM of Folding Squaring Unit in Booth 2 56
4.11 PPM of a Squaring Unit in Booth 2 Folding Technique 59
4.12 PPM of Squaring Unit in Booth 2 Left-to-Right Dual Recoding . . . 61

5.1 Partial Product Matrix of Cubing Unit 64
5.2 Liddicoat and Flynn Method for Step 1 66
5.3 Liddicoat and Flynn Method for Step 2 66
5.4 Stine and Blank Method Stage 1 68
5.5 Stine and Blank Method Stage 2 69
5.6 PPM of a Cubing Unit in Divide and Conquer Technique 70
5.7 PPM of 4-bit Cubing Unit in Divide and Conquer Technique 70
5.8 Partial Product Matrix for Proposed Cubing Unit at Step 1, 2 72
5.9 Partial Product Matrix for Proposed Cubing Unit at Step 3 72

6.1 PPM of Truncated Squaring Unit in [56] 74
6.2 Truncation of P_i in [56] 76
6.3 Partial Product Matrix of Truncated Squaring Unit In [74] 77
6.4 Truncation of P_i in Left-to-Right Encoding 78
6.5 PPM of Truncated Squaring Unit in [71] 79
6.6 PPM of Proposed Truncated Squaring Unit 80
6.7 PPM of Truncated Cubing Unit in [75] 81
6.8 Correction Constant for Truncated Cubing Unit in [75] when $k = 2m$. 82
6.9 Correction Constant for Truncated Cubing Unit in [75] when $0 < k < m$. 83
6.10 Dadda Tree of Truncated Cubing Unit in [75] when $k = 2m$. 84
6.11 Dadda Tree of Truncated Cubing Unit in [75] when $0 < k < m$. . . . 85
6.12 PPM for Truncated Proposed Cubing Unit at Step 1, 2 86
6.13 Partial Product Matrix for Proposed Cubing Unit at Step 3 87

7.1 Half Adder Implementation 94
7.2 Dadda Tree Reduction of a 4 Cubing Unit in Vedic Mathematics . . . 108
7.3 Implementation of $X_iX_jX_k$ 111

8.1 FPGA Flow Diagram 114
8.2 ASIC Flow Diagram 115
8.3 Research Flow Diagram 117

CC

CHAPTER 1

Introduction

1.1 Motivation

In computer arithmetic, multiplication is not commonly utilized as much as addition, but it plays an important role for microprocessors, digital signal processors, and graphics engines [1]. Computers spend a large amount of time and power when performing multiplication, whether it is utilized in software or hardware. Early computers used only parallel adders and a few storage registers, so multiplication was often performed via a sequence of add and shift instructions [2]. Subsequently, future designs utilized high-speed multipliers by employing faster adders as well as better and more optimized memory.

With a multiplication operation, difficulty occurred when the input operands are two's complement. Shaw [2] developed an algorithm for two's complement multiplication, but these algorithms usually require more hardware, therefore, the complexity of distorted systems increased. To solve this problem, some designers utilize different numeric recoding such as Booth multiplier [3]. This algorithm utilizes a number shift and serial multiplication operations along with some recoding to handle two's complement number.

Another multiple algorithm proposed in [4] is a fully parallel multiplier called a *Wallace* tree. This algorithm uses a group of $(3, 2)$ and $(2, 2)$ counters to reduce the number of partial product bits to 2 operands (carry and sum), then they are added by using a faster adder. A new scheme presented in [5], called *Dadda* tree, utilized the lower bound of combining different types of counters to optimize the number of $(3, 2)$

and $(2, 2)$ counters. Later, in [6], this tree was optimized to find optimum building block for implementing of a multiplier with a larger input size.

Over the several decades, Very Large Scale Integration (VLSI) technology and integrated circuit processing have dramatically improved. According to Moore's Law, the number of transistors in a die doubles approximately every 18 months [1]. As the number of transistors and speed in a chip increase, more specific units and elementary functions are added to computer's arithmetic unit, allowing it to run faster. Therefore, computer's performance is improved and more complex scientific problems can be solved.

Squaring is a special operation of multiplication. It plays an important role in a computer's arithmetic unit and many applications such as, image compression [7, 8], equalization [9], multimedia applications [10–12], and decoding and demodulation [13, 14]. It can also be an efficient way to compute other basic functions [15–17]. Therefore, improving the performances of squarer is a goal for many researchers.

Another type of multiplication is powering function (X^Y). It is also needed for computer 3D graphics, digital signal processing (DSP), scientific computing, artificial neural networks, logarithmic number systems, and multimedia applications [12], and other important functions [15]. Cubing (X^3) is a special instance of powering and plays an important role in many algorithms, like signal processing, image processing, cryptography [18, 19], and elementary function approximation [20–23].

Recently, with the increment of density of gates in each die, parallelism through single instruction multiple data (SIMD) processing can optimize processing for more than one operation. By enhancing application specific and embedded processors are ways to provide higher levels of parallelism [24]. These enhancements are combined with an extended instruction set that takes advantage of additional hardware by allowing more operations to be encoded in a single instruction. With this increased

parallelism and the shrinking feature sizes, the next generation of application-specific processor can execute significantly more work. Consequently, new techniques for calculating the squaring and cubing units in parallel without the use of basic multipliers or tables becomes an attractive alternative.

For these reasons, squaring and cubing operations still are objects of many researchers. They should be calculated with a minimum amount of latency as to not slow down these calculations, especially for real-time systems [25].

1.2 Research Contributions

This dissertation will discusses modification to algorithms to compute parallel squaring and cubing units in both unsigned and signed representation. After that truncated techniques are applied to get an optimal performance.

Each unit is modeled and estimate its area, delay by linear evaluation. Hardware design using $Verilog$ code is used in this dissertation. A C program is used to generate $Verilog$ code for each unit. These units are simulated and verified in Hardware Description Language (HDL) simulators. Area, delay, and power consumption are calculated for each of the designs and compared with previous approaches in both $Virtex\ 5\ Xilinx\ FPGAs$ and $IBM\ 65nm\ ASIC$ standard-cell libraries. The following are contributions of this work

1. Propose a new model for unsigned squaring and cubing unit.

2. Investigate signed squaring, cubing units.

3. Estimate the Area, Delay, and Power of these units by using linear model.

4. Propose a model for truncated squaring, cubing units.

1.3 Dissertation Organized

The remaining portion of this thesis is structured as follows : Chapter 2 will review the background information on the parallel multiplication. Chapter 3 presents various ideas of truncated multiplication . Chapter 4 illustrates squaring units and proposed a new model to improve their performance. Chapter 5 gives information on parallel cubing units and propose a new way to implement it. Chapter 6 analyses truncated squaring and cubing units with previous technique presented in Chapter 3. Chapter 7 used *Linear-Delay* model for analyzing area and delay of each units. Chapter 8 is an implementation of squaring and cubing units in both techniques FPGA and ASIC and conclusion.

CHAPTER 2

Multiplication

Multiplication is frequently required in digital signal processing applications (digital filtering, FFT, convolution,...). Parallel multipliers [4, 5] provide fast multiplication for these applications, but require large amount of area. This parameter increases proportionally with the square of the word size. Therefore a large number of transistors are needed, hence, multipliers consume a large amount of power [26].

A multiplier is one of vital parts in application-specific processors, such as digital signal processing (DSP) systems. This is because many signal processing algorithms heavily utilize multiplication. To improve performances of multipliers, high speed parallel architectures are often used. These architectures contribute significantly to the overall power dissipation of these systems [27]. Consequently, reducing power consumption of parallel multipliers is important in design of digital signal processing systems. In the past, several techniques have been proposed to reduce power dissipation of parallel multipliers. Some of them tried to reduce dynamic power dissipation by eliminating spurious transitions [28]. The others tried to develop multiplier architectures [29]. Although these techniques help reduce power dissipation, but further reductions are needed for future DSP systems [28].

Since current and future portable devices using DSP hardware have become popular. As more of these devices increase, so does the power consumption. Hence, long battery life is a higher priority feature for most users.

To illustrate multiplication algorithms, a partial product matrix is often used. A $m \times n$ bits multiplication can be viewed as forming n partial products of m bits each

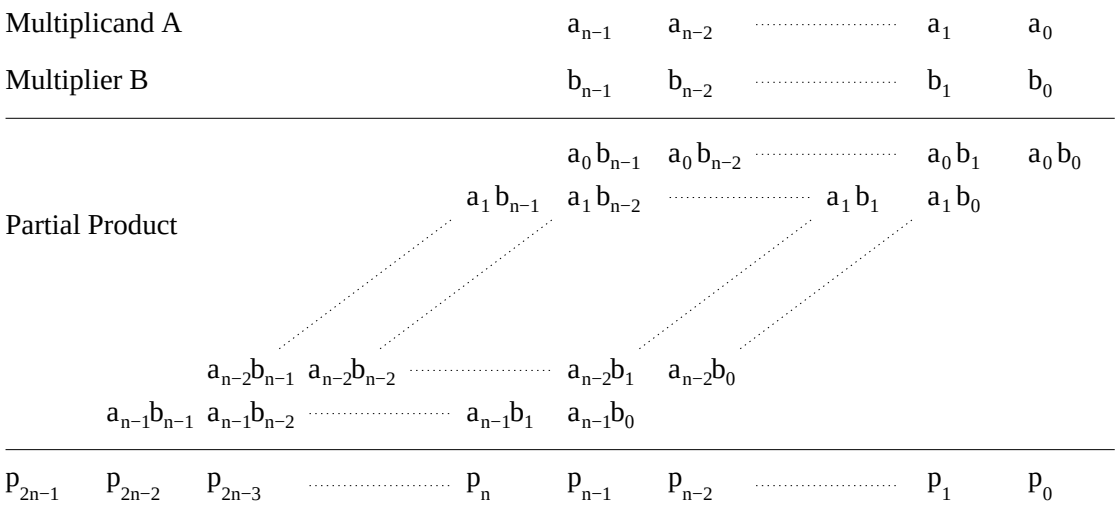

Figure 2.1: Partial Products Matrix of Multiplication.

and then summing the appropriately shifted partial products to produce $(m+n)$-bit result P. Theoretically, multiplication of two n bits unsigned fraction A and B will yield $2n$ bits products P, as shown below:

$$\begin{cases} A = \sum_{i=0}^{n-1} a_i \cdot 2^{-n+i} \\ B = \sum_{i=0}^{n-1} b_i \cdot 2^{-n+i} \\ P = \sum_{i=0}^{2n-1} p_i \cdot 2^{-2n+i} \end{cases} \tag{2.1}$$

Digital architectures for multiplication usually involve three separate steps:

1. Partial Product Generation (PPG) to utilize a collection of gates to generate the partial product bits.

2. Partial Product Reduction (PPR) to utilize adders to reduce the partial products to sum and carry vectors.

3. Final Carry Propagate Addition (CPA) to add the sum and carry arrays to produce the product.

Figure 2.1 illustrates the generation and summing of partial products in $n \times n$ bits multiplication. When A and B become larger, dot diagram is used to represent partial products more conveniently. For example, Figure 2.2 shows a dot diagram for an 8-bit

Partial Product Selection Table	
Multiplier Bit	Selection
0	0
1	Multiplicand

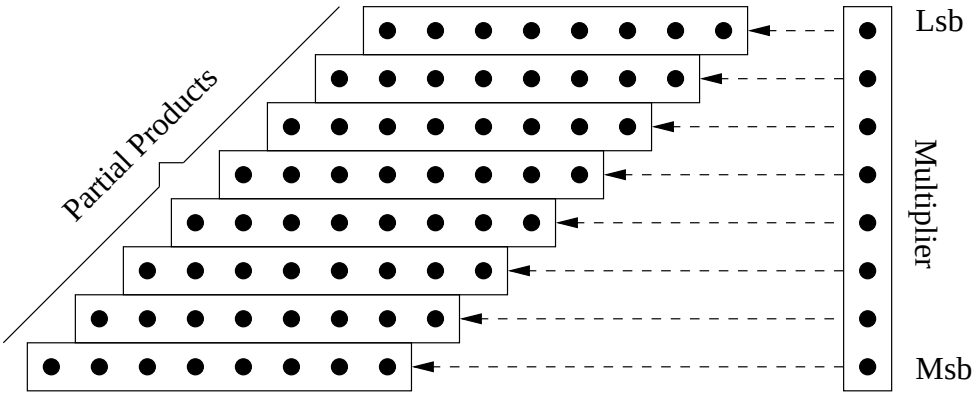

Figure 2.2: $8x8$-bit Dot Matrix Diagram of Multiplication.

multiplier. Each dot is equal to each bit in partial product matrix. It can be either "0" or "1" depends on the value of multiplicand and multiplier.

There are several techniques that can be used to implement multiplication. In general, the choice is base upon factors such as latency, throughput, area, and design complexity. The latency factor is related to the height of the dot diagram. This relationship can vary from logarithmily to linearly [30]. The simplest way to implement a multiplier is used an $(n+1)$-bit CPA to add the first two partial product arrays, then another CPA to add the third one to the running sum, and so forth. This technique will require $(n-1)$ $CPAs$, so it is slow even if a fast CPA is being used. More efficient parallel approaches use some sort of array or a tree of full adders to sum the partial product bits and using pipeline to reduce the cycle time and increase the throughput. In addition, this technique can save a large amount of area.

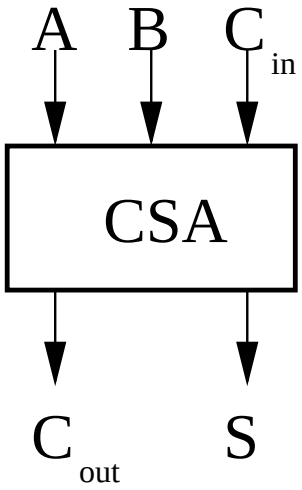

Figure 2.3: A Carry-Save Adder or (3, 2) Counter.

2.1 Carry Save Concept

After finishing first step PPG, all bits in this matrix will be added to get the final result by utilizing adders to reduce the execution time. Because the major delay of adders is come from the carrier chain, therefore, it is important to reduce the total time is involved in summing carries [31]. There are two principles to help save time arriving at a product:

- Carry-Save Addition (CSA): idea of utilizing addition without the carriers connected in series, but just to count.
- Carry-Propagate Addition (CPA): idea of the carries connected in series to produce a result in either conventional or redundant notation.

For these methods, each adder above acts the same as the full adder, however, the view in which each connection is made from adder to adder is where the main difference lies. Because both adders compute both carry and save information, sometimes $VLSI$ designers refer to it as a Cary-Save Adder (CSA) [31]. The summation of the partial product bits is done by using a CSA, each adder attempts to count the number of inputs that are "1", so it also referred as a counter. These counters can

be connected by several different topologies. These topologies are referred as regular and irregular ones. In the former, the counters are connected in a normal pattern that is replicated so that the design of partial product array can be connected as a hierarchical design. On the other hand, in the latter, the counter are connected in order to minimize the delay. The different of these topologies can be seen in array or tree multipliers.

A (c, d) counter is a special adder, in which c is the number of columns to be added and d is the number of outputs. For example, a $(3, 2)$ counter counts its inputs with the same weight and has 2 outputs. Fig 2.3 shows a typical $(3, 2)$ counter. Therefore, a n-bit CSA has three n-bit input operands and two n-bit outputs, one for sum and the other for carry. Large operand sizes would require more $CSAs$ to produce a result. However, a CPA would be required to produce the correct result [31]. For example, Figure 2.4 shows how to add four operands together $A + B + D + E$ with the value $5 + 12 + 7 + 10$. The implementation utilizing carry-save concept for this example is shown in Figure 2.5. Higher order counters can be created by putting together various sized counters. A higher counter (p, q) takes p input bits and produces q output bits. Because q is a number between 0 and $2^q - 1$ so the value of p must be smaller than 2^q.

2.2 Carry-Save Array Multipliers (CSAM)

A Carry-Save Array Multipliers is the simplest multiplier in which each partial product bit is being added [32] similar to paper and pencil multiplication is performed. Its operation is based on paper and pencil style multiplication where each partial product bit is added in array. The topology of a $CSAM$ is a regular one, so it is easy be built by a hierarchical design. Figure 2.6 shows an implementation of a 8×8-bit unsigned multiplier. Each diagonal in $CSAM$ corresponds to a column in the multiplication matrix shown by the bold line in Figure 2.6. Because of implementing in

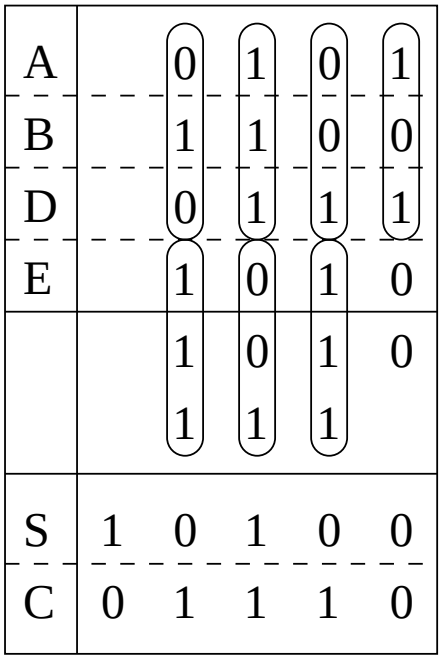

Figure 2.4: An Example of Carry-Save Concept Adder.

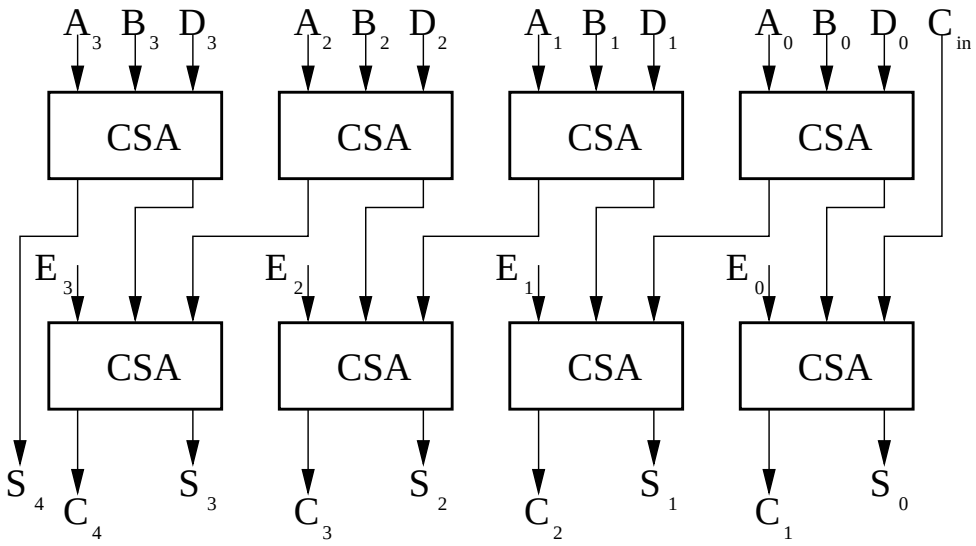

Figure 2.5: A Four Operands 4-bit Multi-Operand Adder.

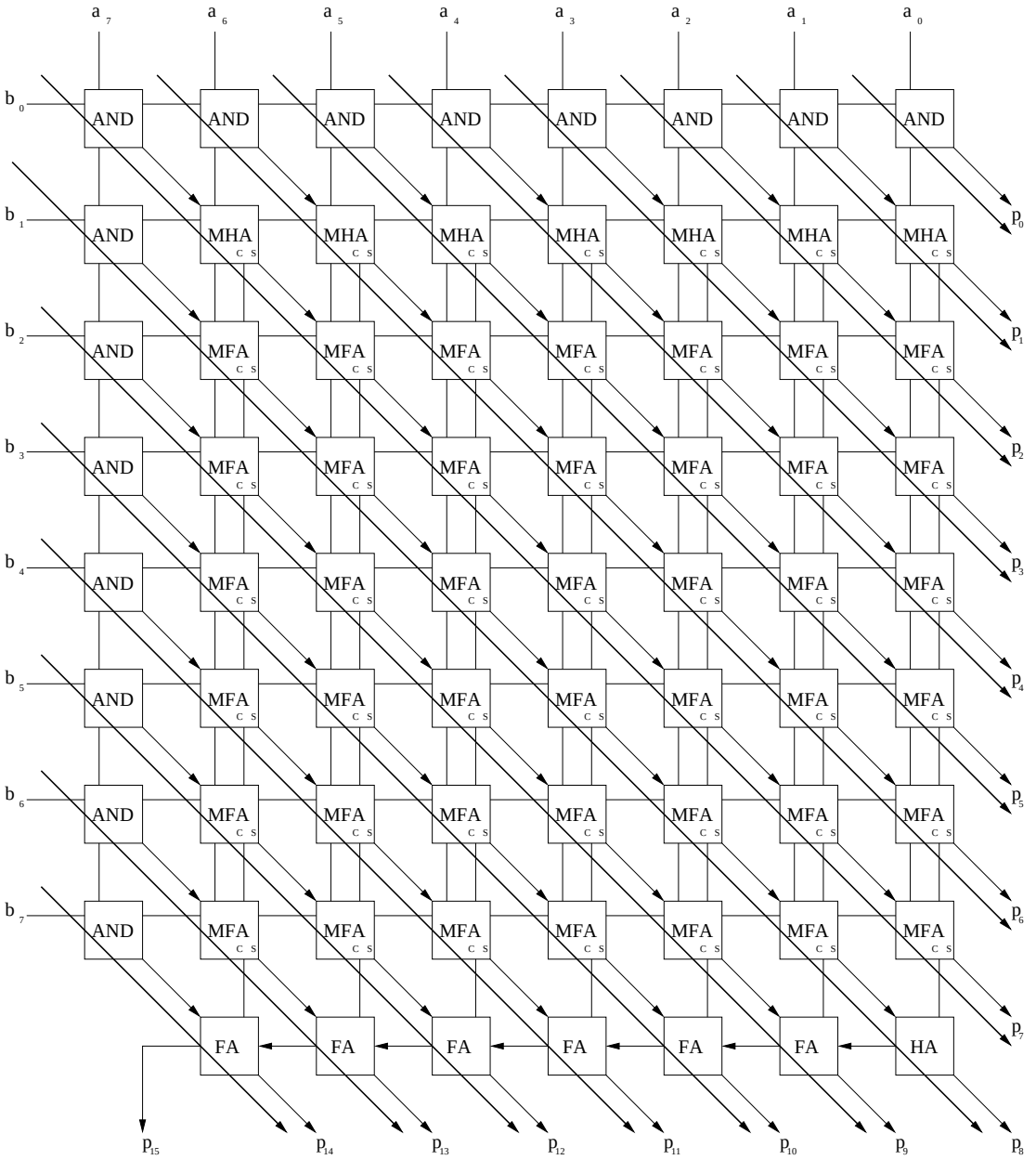

Figure 2.6: 8×8-bit Carry-Save Array Multiplier.

a square shape, $CSAMs$ allow metal tracks or interconnect to have less congestion, thus, reducing capacitance as well as it is easier to organize [31].

A $CSAM$ generates partial product bits by utilizing AND gates and uses an array of $CSA's$ to perform reduction. These AND gates form the partial product's and the $CSA's$ will be used to sum these products together or reduce them. Because half the lower part of the product is computed by the reduction, therefore, only

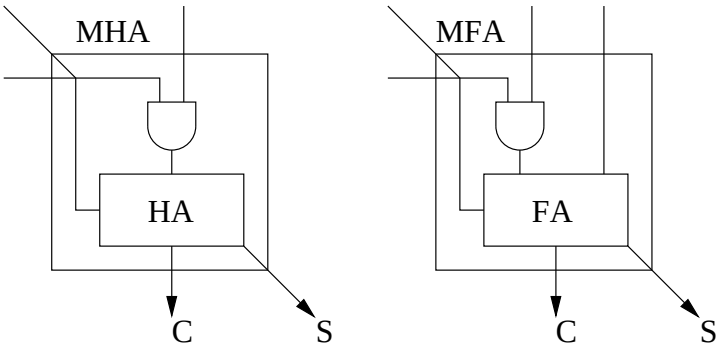

Figure 2.7: Modified Half Adder and Full Adder.

the upper half of the product need to be added in the final CPA [31]. To perform $CSAM$, each adder is modified so that it can perform partial product generation and addition. Two type of adders are used for $CSAM$ and called the modified half adder (MHA), and the modified full adder (MFA). A MHA consists of an AND gate to generate the partial product bit and one half adder (HA) to add this partial product bit with another one from the previous row. A MFA consists of an AND gate for generating partial product bit, and a full adder FA. This FA is used to add this partial product bit with the sum and carry bits from previous row. Figure 2.7 shows the block diagram of the MHA and the MFA.

2.3 Tree Multipliers

Tree topologies are a fast way for summing partial products. In a tree, counters are connected in different ways for each slice [30]. Therefore, to reduce the delay of array multipliers, tree reduction are often employed. A tree multiplier, which have $O(log(n))$ delay [31], uses an idea of reduction to reduce their partial product height down until it is equal *2*, then a high speed CPA is applied to get the final result. Tree multipliers vary in a way that each CSA performs partial product reduction. The main objective is to reduce the its height by utilizing the carry-save concept. Each partial product bit is reorganized, so that it can achieve an efficient reduction

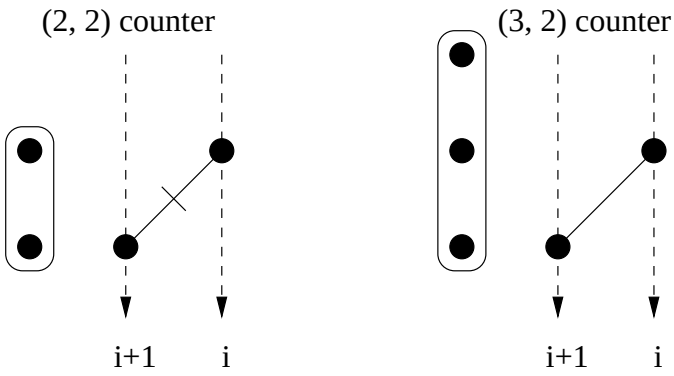

Figure 2.8: (2, 2) and (3, 2) Counters in Tree Multiplier.

in the column. This is possible for multiplication, because each bit in the matrix is commutative and associative with respect to addition.

The first trees were introduced in [4],called a Wallace tree, by utilizing a reduction using $(3, 2)$ and $(2, 2)$ counters in parallel. A Dadda tree [5] is an improvement of Wallace tree in which the number of reduction stages are minimized. Another tree presented in [33] improved multiplier's area compared with both Wallace and Dadda trees. In a tree multiplier, $(3, 2)$ and $(2, 2)$ counters are defined and used as in Figure 2.8.

2.3.1 Wallace Tree Multipliers

In 1964, a fast multiplier is defined using a column compression technique [4]. Partial product bits in the matrix of multiplier are reduced to two arrays carry and sum. In the final stage, a fast CPA is utilized by adding these two arrays together to get a final product. This technique has a total delay which is proportional to the logarithm of the operand word length, so these multipliers are faster than array multipliers [35].

Because all bits in the partial product matrix can be moved and interchanged as long as they are kept in the same column. Therefore, they can be reorganized in a proper way. Following are steps to implement a Wallace tree multiplier:

1. Reorganize dot matrix into inverted pyramid.

2. Group rows into a group of *3*.

3. Starting at rightmost column using $(2, 2)$ and $(3, 2)$ counters as long as each subset of *3* has at least 3 rows.

4. Repeat Step *2* and *3* until the final height of the reorganized matrix is *2*.

As can be seen in the Figure 2.9, each row are grouped in to set of three by using $(3, 2)$ counter as maximum as possible, the rest is group into set of two by using $(2, 2)$ counter. Rows that are not part of a three row set or two row set is transferred to the next reduction stage. The height of the matrix in the j^{th} reduction stage is w_j will be defined by the following recursive equation:

$$\begin{cases} w_0 = n \\ w_{j+1} = 2 \times \lfloor \frac{w_j}{3} \rfloor + (w_j \ mod \ 3) \end{cases} \tag{2.2}$$

For example, apply Equation 2.2 to the case of $n = 8$, four stages with their height $w_1 = 8$, $w_2 = 6$, $w_3 = 4$, and $w_4 = 3$ can be computed, respectively. Figure 2.9 is an implementation of 8×8-bit Wallace tree multiplier. As can be seen in this figure, four steps of reduction are processed step by step. In the first stage, *16* FAs and *5* HAs will be used to reduce matrix to the new one with the height is equal to *6*. In the second stage, *10* FAs and *6* HAs are employed to get a new matrix with its height is *4*. In the third stage, *7* FAs and *5* HAs will be used. In the last stage, *3* FAs and *8* $HAs$are employed to reduce the matrix height to *2*. The final product is generated by using an *11*-bit length CPA. Hence, this multiplier requires *64* AND gates, *36* FAs, *24* HAs and an *11*-bit CPA. The total delay is the sum of one AND gate delay, *4* FA delay (through 4 stages) and the delay of the *11*-bit length CPA. In general, the total HA, FA, and the length of CPA can be expressed as in [26].

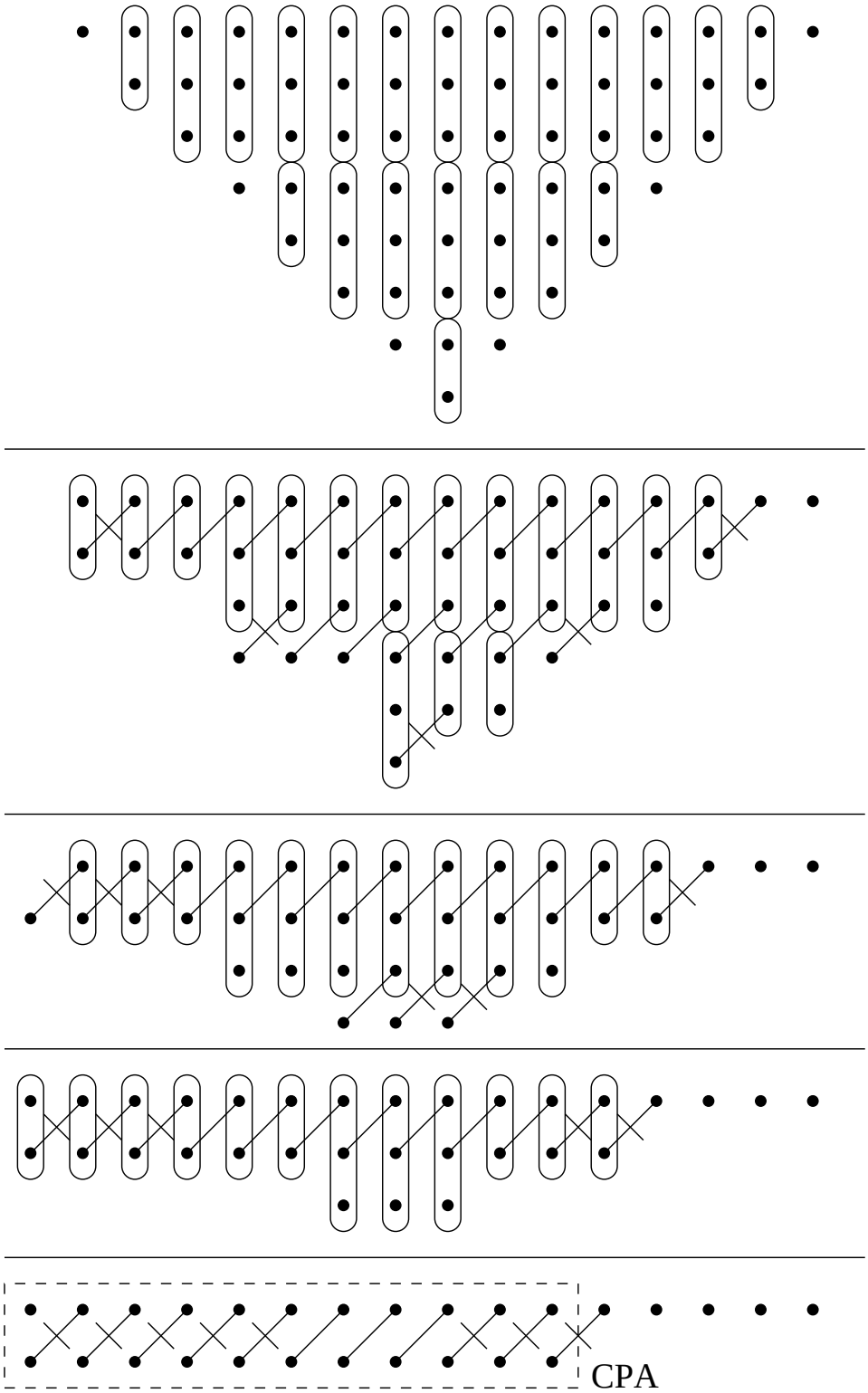

Figure 2.9: Wallace Tree Multiplier.

2.3.2 Dadda Tree Multipliers

In 1965, Wallace tree was modified in [5], in which, a unique placement strategy for the stages reduction was proposed. This technique minimizes the number of $(3,2)$ and $(2,2)$ counters [34], but increases the CPA length. Therefor the number of intermediate stages is set in term of lower bound. According to [31], the height of partial product matrix (Dadda sequence) can be defined as:

$$2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 9 \rightarrow 13 \rightarrow 19 \rightarrow 28 \rightarrow 42 \rightarrow 63 \cdots$$

Following are stages to implement Dadda tree multipliers:

1. Reorganize dot matrix into inverted pyramid (optional).
2. Figure out Dadda sequence and where the height of the dot matrix falls within this sequence:
$$\begin{cases} h_0 = 2 \\ h_{i+1} = \left\lfloor h_i \times \frac{3}{2} \right\rfloor . \end{cases} \tag{2.3}$$
Where h_i is the height for the i^{th} stage from the end
3. Draw a dotted line within Dadda multiplier representing next Dadda sequence.
4. Starting at right most column, reduce stage until Dadda sequence is meet from Step *3*.
5. Repeat from Step *3* until final height is *2*.

For example, when $n = 8$, Equation (2.3) is applied, the height sequence $h_0 = 2$, $h_1 = 3$, $h_2 = 4$, $h_3 = 6$, and $h_4 = 8$, consecutively is established. Four stages are needed for this tree reduction. Figure 2.10 represents an 8×8-bit Dadda tree multiplier. In the first stage, a dotted line divides the matrix into *2* sections with the heights are $h_3 = 6$. As can be seen in this figure, starting form the right most to the 5^{th} column, each of them has a height smaller or equal to *6*, but from the 6^{th}

column, its height is *7*, so one $(2,2)$counter must be employed to reduce its height to *6*. This counter will bring one carry to the 7^{th} column. Hence, one $(3,2)$ and one $(2,2)$ counter should be used to reduce the height of 7^{th} column to *6*. The process continues to the next until the left most column, and hence, a new matrix with the maximum height is *6* is established.Further steps are continued until the height of the final matrix is *2*. In the final stage, a *14*-bit CPA is used for generating final product. Figure 2.10 shows that this unit requires *64* AND gates, *33* FAs, *9* HAs and one *14*-bit length CPA. The total delay of this multiplier is the sum of one AND gate, *4* FA (though four stage reduction) and the delay through the *14*-bit length CPA.

In general, the number of $(3,2)$ and $(2,2)$ counters and the length of CPA in a Dadda tree is computed in [35]:

$$\begin{aligned} N_{(3,2)counters} &= n^2 - 4n + 3 \\ N_{(2,2)counters} &= n - 1 \\ CPA_{length} &= 2n - 2 \end{aligned} \tag{2.4}$$

2.3.3 Reduced Area (RA) Multipliers

The RA multiplier is another way to improve both Wallace and Dadda reduction techniques. The difference among RA and Wallace, Dadda tree multiplier is that the maximum number of FAs are utilized as early as possible, and HAs are carefully placed to reduce the word size of the CPA. This is an algorithm that is typically classified as a greedy algorithm. The following is the algorithm for a RA multiplier:

1. Reorganize dot matrix into inverted pyramid.
2. For each stage, the number of FAs used in column i^{th} is equal $\lfloor \frac{b_i}{3} \rfloor$, where b_i is the number of bits in column i^{th}
3. HAs are only used as follows:

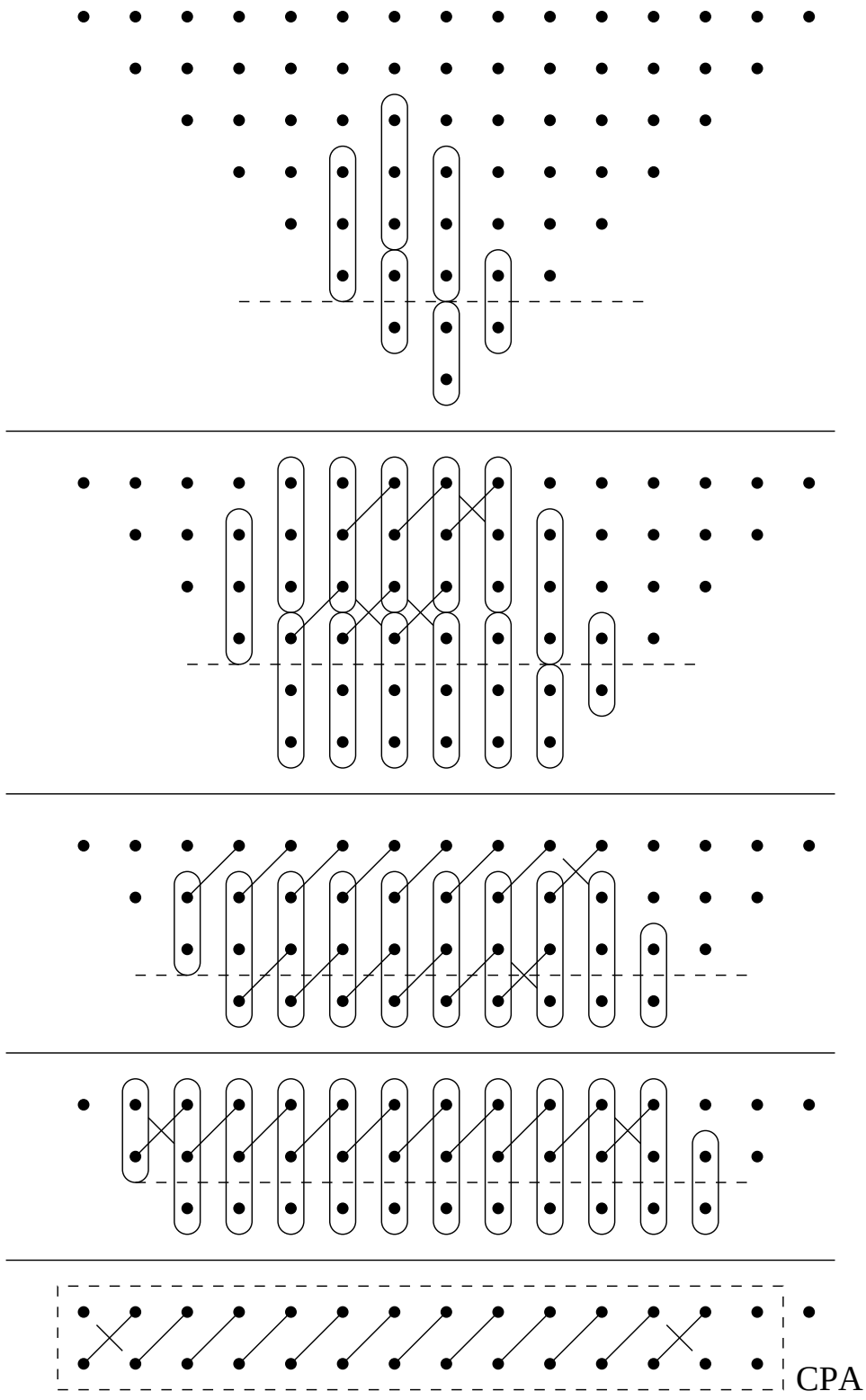

Figure 2.10: Dadda Tree Multiplier.

(a) In the right most column.

(b) When required according to Dadda.

Figure 2.11 shows a PPM for 8×8-bit RA tree multiplier. In the first stage, step *2* is applied to through column 2^{th} to 12^{th}, and only step $3(a)$ is applied to the column 1^{th}. In the second stage, step *2* is applied to through column 3^{th} to 12^{th}, step *3(a)* is applied to the column 2^{th}, and step *3(b)* is applied to the column 8^{th} because of violating Dadda sequence. Third and fourth stage will be continue doing in the same way to get carry/sum arrays. The last stage is implemented by using a *10*-bit length CPA to get the final result.

As can be seen in Figure 2.11, a 8×8-bit RA tree multiplier requires *64* AND gates, *39* FAs, *7* HAs, and one *10*-bit CPA. The total delay for generating final product is the sum of the one AND gate, *4* FA (through *4* stages reduction), and the delay through the *10*-bit CPA.

2.4 Baugh and Wooley Multiplication

The previous section gave a general idea for unsigned multiplication. For signed ones, $Baugh$ and $Wooley$ [36] proposed a straightforward way for calculating the output product. Using n-bit length multiplier A and multiplicand B, presented in two's

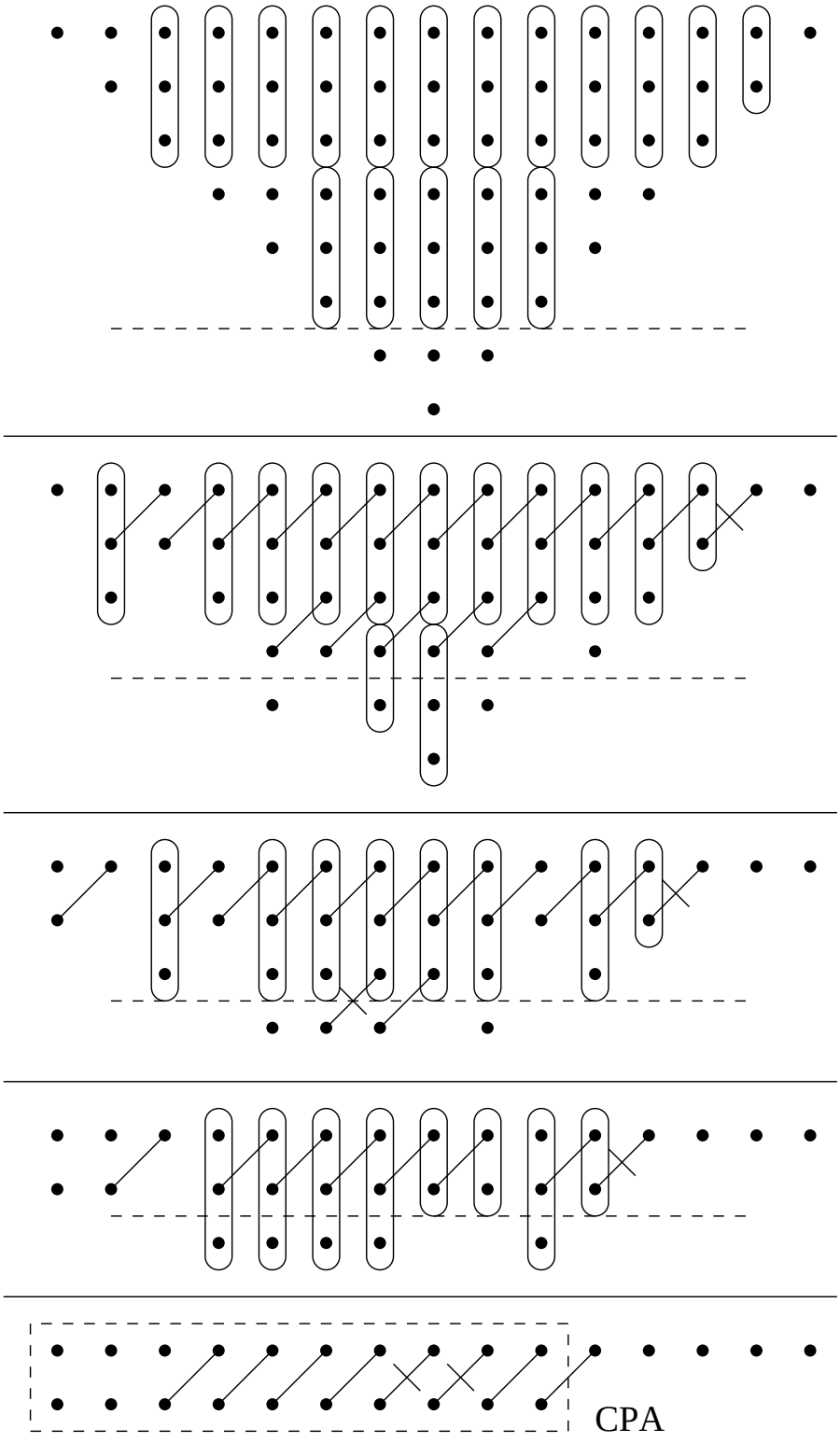

Figure 2.11: Reduce Area Multiplier.

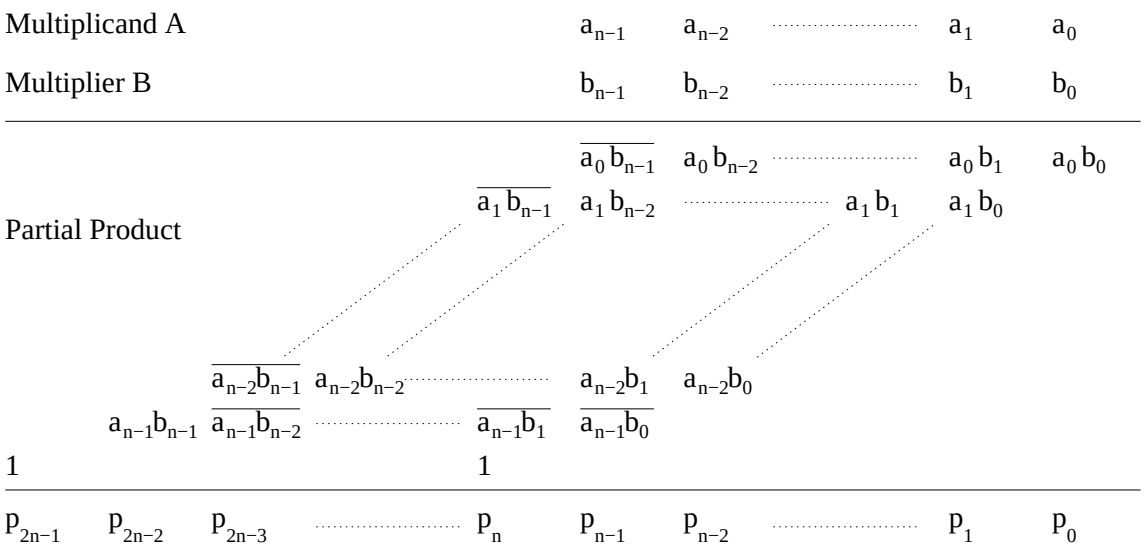

Figure 2.12: PPM of Baugh and Wooley Multiplier.

complement number system, and P is an output product, they can be expressed as:

$$
\begin{aligned}
A &= -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \\
B &= -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i \\
P &= a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j} \\
&\quad -[\sum_{j=0}^{n-2} a_{n-1} \cdot b_j 2^{n-1+j} + \sum_{i=0}^{n-2} a_{n-1} \cdot b_i \cdot 2^{n-1+i}] \\
&= a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j} \\
&\quad +[\sum_{j=0}^{n-2} \overline{a_{n-1} \cdot b_j} \cdot 2^{n-1+j} + \sum_{i=0}^{n-2} \overline{a_{n-1} \cdot b_i} \cdot 2^{n-1+i}] \\
&\quad +2^{2n-1} + 2^n
\end{aligned}
\tag{2.5}
$$

The PPM of this multiplier can be established as in Figure 2.12. To implement this matrix, some of $NAND$ gates are used instead of AND gates, two bits "1" are added in proper columns for signed adjustment. Figure 2.13 is an implementation of 8×8-bit $Baugh$ and $Wooley$ multiplier in hardware.

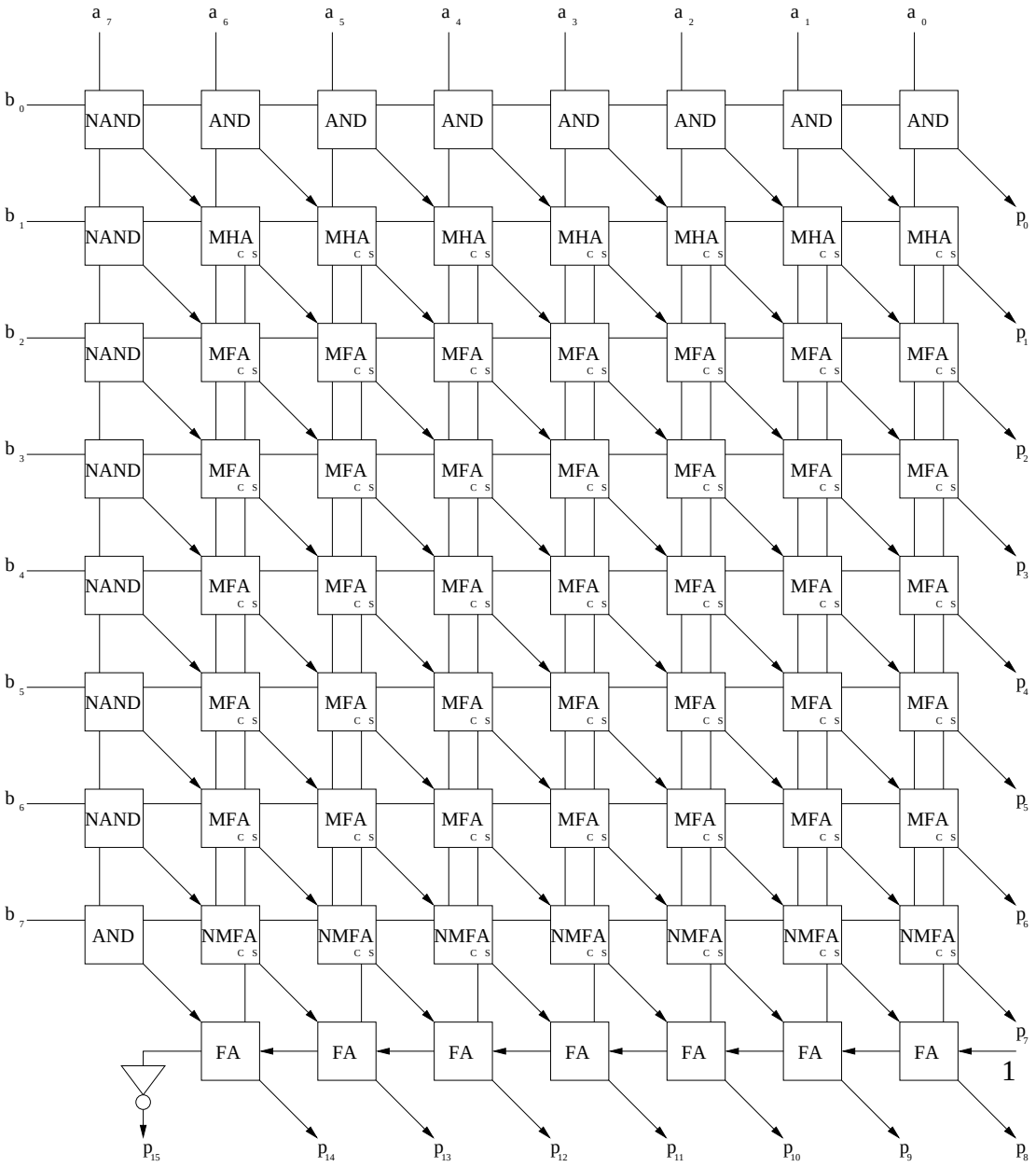

Figure 2.13: Implementation of Baugh and Wooley Multiplier.

2.5 Modified Booth Multipliers with Radix 4

A recoding scheme introduced by Booth [3] reduces the number of partial product bits and the height of its matrix nearly by half, hence, reduces hardware requirement and improve multiplier's performance. Straightforward extensions of the Booth recoding scheme [37, 38] called Modified Booth Multiplier (MBP) can further reduce the number of partial product bits. A digit i^{th} of a MBP with radix *4* (sometimes called Booth *2*) can be recoded as

$$\begin{aligned} B_i &= x_{i+1}x_ix_{i-1} \\ &= -2 \cdot x_{i+1} + x_i + x_{i-1} \end{aligned} \tag{2.6}$$

A signed number X presented in two's complement can be expressed as

$$\begin{aligned} X &= x_{2n-1}x_{2n-2}\cdots x_1x_0 \\ &= -x_{2n-1} \cdot 2^{2n-1} + \sum_{i=0}^{2n-2} x_i \cdot 2^i \\ &= (-2 \cdot x_{2n-1} + x_{2n-2} + x_{2n-3}) \cdot 2^{2(n-1)} \\ &\quad +(-2 \cdot x_{2n-3} + x_{2n-4} + x_{2n-5}) \cdot 2^{2(n-2)} \\ &\quad + \cdots + (-2 \cdot x_3 + x_2 + x_1) \cdot 2^2 \\ &\quad +(-2x_1 + x_0 + 0) \cdot 2^0 \\ &= B_{n-1}B_{n-2}\cdots B_1B_0 \\ &= \sum_{i=0}^{n-1} B_i \cdot 2^{2i} \end{aligned} \tag{2.7}$$

The output of a multiplier unit can be expressed as

$$\begin{aligned} P &= X \cdot Y \\ &= \sum_{i=0}^{n-1} (B_i \cdot Y) \cdot 2^{2i} \end{aligned} \tag{2.8}$$

Because B_i is recoded as in Equation 2.6, so the value of B_i will be in the set $\{-2, -1, 0, 1, 2\}$, hence, each partial product row $B_i \cdot Y$ will come from the set

Table 2.1: Partial Product Selection

x_{2i+1}	x_{2i}	x_{2i-1}	$B_i \cdot Y$
0	0	0	0
0	0	1	+ Multiplicand
0	1	0	+ Multiplicand
0	1	1	+2 × Multiplicand
1	0	0	-2 × Multiplicand
1	0	1	- Multiplicand
1	1	0	- Multiplicand
1	1	1	- 0

$\{-2Y, -Y, 0, Y, 2Y\}$. Table 2.1 is selection table for partial product in Booth *2* multiplication.

Because it is signed multiplication, the most significant partial product bit of multiplicand operand is kept at the same, no more bits is added to guarantee a positive result . When ± Multiplicand (entries 1, 2, 5 and 6 from the partial product selection table) is selected, the $n+1$ bit section of the effected partial product is filled with a sign extended copy of the multiplicand. This sign extension occurs before any complementing that is necessary to obtain -multiplicand. The PPM of Booth *2* multiplier can be viewed as in the Figure 2.14, in which, S is signed bit of B_i, and E is signed extension bit. The leading "1" strings, created by assuming that all partial products were negative, are cleared in each partial product under a slightly different condition. The leading "1" string for a particular partial product is cleared when that one is positive. For signed multiplication this occurs when the multiplicand is positive and the multiplier select bits chooses a positive multiple, and also when the multiplicand is negative and the multiplier select bits choose a negative multiple. A simple $EXCLUSIVE - NOR$ between the sign bit of the multiplicand and the

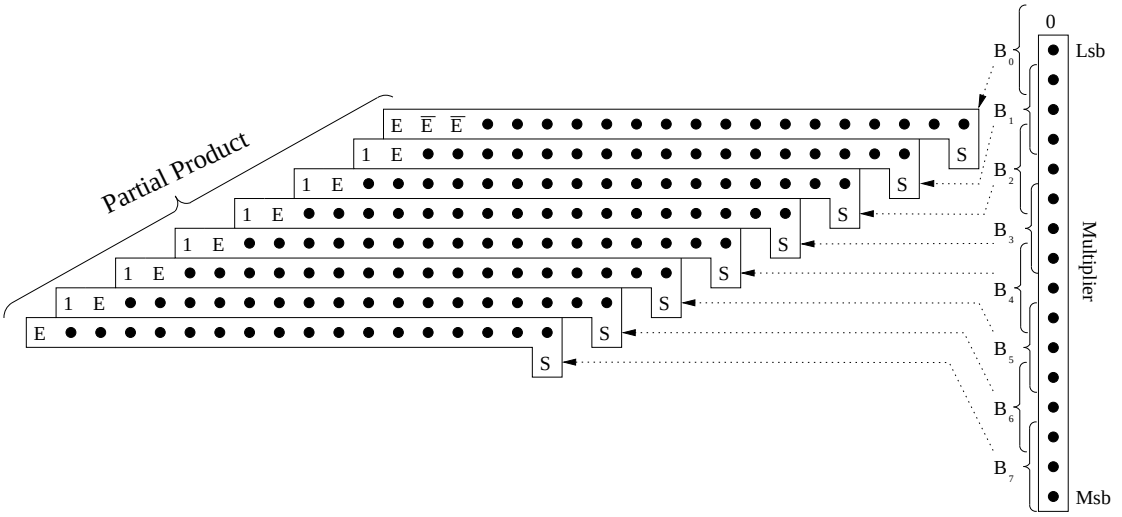

Figure 2.14: PPM of Booth 2 Multiplier.

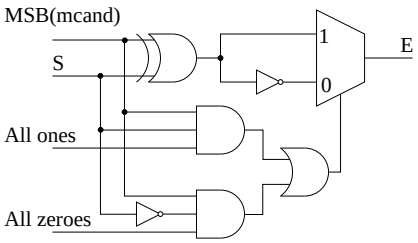

Figure 2.15: Sign Extension Bit of MBP.

high order bit of the partial product selection bits in the multiplier generates the one to be added to clear the leading "1"s correctly [38]. Although the sign extension presented in [38] is efficient at creating two's complement multipliers, it ultimately has a negative effect in that it does not compute positive results well. The reason for incorrect result occurs is that -0 and $+0$ is really not specified as a multiplicand partial product [39]. To fix that, sign extension bit is synthesized as in Figure 2.15. Each row of dots in the Figure 2.14 is a partial product row and can be generated by using logic functions as in the Figure 2.16.

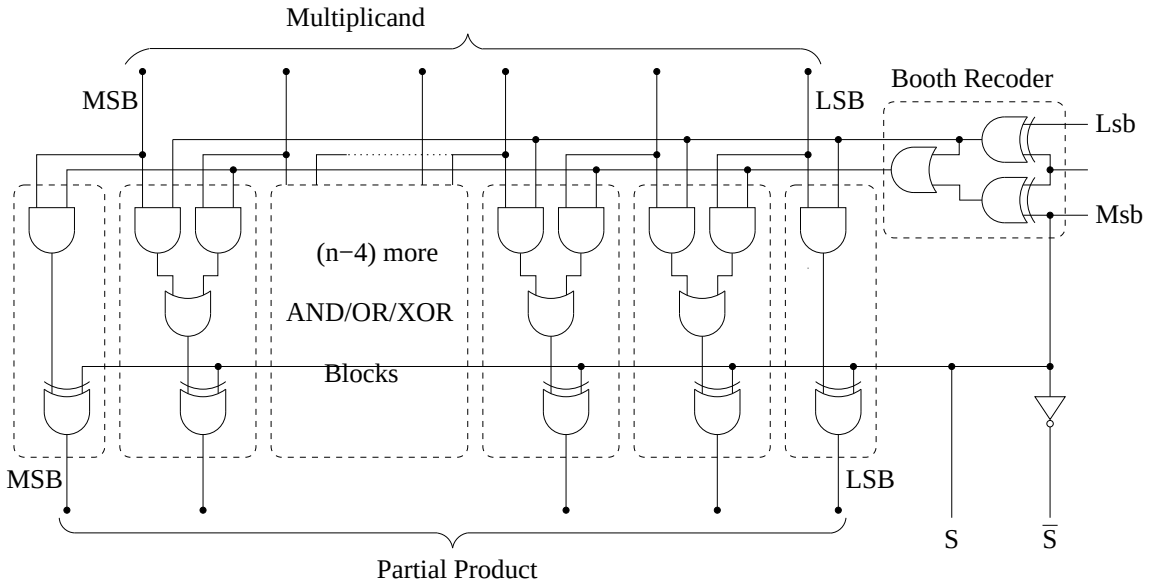

Figure 2.16: Partial Product Generation of Booth 2.

CHAPTER 3

Truncated Multipliers

In most DSP systems, the results of the basic operation are kept at a constant word length. This action is applied for multiplications that uses two n bits inputs and produces a product of $2n$ bits. The output then rounded to n bits. Consequently, the full addition of the partial product is executed before rounding to get accurate results. Is there any efficient way to do that, so the area and power consumption of a multiplier can be reduce is the goal of many designers. Truncated multipliers are the answer for this question.

Truncated multiplier can produce a fairly accurate result while significantly lowering area requirements. In addition, because of neglecting partial product bits in the least significant columns, truncated multipliers save a large amount of power consumption [28]. The first idea of truncated multiplier was introduced in [40], in which, there are two main methods for compensating the error caused by truncated the less significant part. The first one used a constant to estimate the weight of the truncated part. The second one, called conditional correction, used sum of the first r partial product terms in the $(n+k)^{th}$ column to estimate the sum of the first r partial product terms of the $(n+k+j)^{th}$ and $(n+k+j+1)^{th}$ column.

The constant correction method was refined by Schulte and Swartzlander [41]. Their idea is to use a similar method to [40], except that they used a rounding error to add to the constant. The value of the constant is total of rounding error and the mean of the truncated partial product terms. This idea developed further in [42] in which nearly half of the area of multiplier is truncated. This scheme saves significant

area, but produces a large error [43], especially when the width of the multiplicand and multiplier increase. The reason that the error in [42] is larger than that one in [41] is because only one bit $'1'$ is used for compensation. To get results with more accuracy, one more parameter k is used [28, 41]. When k increases, the error decreases, but the area and delay of the truncated multiplier increase. It is a trade-off between accuracy and area. Because of using a constant for compensating the error, all multipliers depend on this method are called correction constant truncated multiplier (CCT).

According to [44], the CCT method is easy for implementation, but has larger error compared with other methods. There are two main issues with error of CCT method. First, because of adding a constant for all data inputs, the resulting product will have non-zero DC component. Second, to limit the range of the maximum error to be less than an LSB of the data path, the area and power saving cannot be maximized. To reduce the error more, another technique was introduced in [21, 44–46], in which, the sum of all partial product bits in the most significant column of truncated part is used for compensating the reduction operation. One additional element of the rounding error in the CCT method is also used. Because the value of that sum is a variable that depends on the inputs of multiplier, hence, this is called variable correction truncated multiplier (VCT).

Many researchers extended the idea of VCT to develop their own variant truncated multipliers. In [43, 47, 48] proposed a scheme, in which half of columns in partial product matrix are truncated. To compensate the error, a circuit made by $AND-OR$ gates is used to generate carries to the next column. The inputs of this circuit are get from two most significant columns in the truncated part. A similar approach presented in [42] saves a lot of area but the circuit has a long critical delay part and no guarantee that the error can be compensated. To solve this problem, [49] proposed a fixed-width signed multiplier in which the circuit generates carries like in [43] are generalized by considering either the partial product terms in the $n + k + 1^{th}$ column

or their complements. This technique then is extended to the different value of h [50]. Another scheme presented in [51] attempts to explain the result obtained in [50], but the circuit is complicated for implementation and does not improve the performance compared to [50]. Others attempt in [52–54] represented a truncated scheme utilized a modified Booth encoding technique to reduce more area.

According to [45], the maximum error of VCT is smaller than that one of CCT, but it can be costly for the implementation. Another restriction of both CCT and VCT is that the error is not symmetrical. To overcome this restriction, another scheme in [55] was proposed. This architecture looks like VCT except that one more logic circuit is used to adjust VCT error.

Another technique called hybrid correction truncated multiplier (HCT) presented in [28] also tried to reduce error. This technique used a parameter that is the percentage of partial product bits in the most significant column of the truncated area to calculate a constant for compensation. HCT actually is the combination of CCT and VCT methods. The maximum error of the HCT is greater than that one of the VCT, but smaller than that one of the CCT method [28].

Recently, there was another approach [56–61], in which, mean square error of a truncated multiplier is minimized. This approach uses the idea of the VCT, but a function with inputs are the terms in $(n + k + 1)^{th}$ column are used instead of a variable. In [62] was additional method with mean error is reduced but mean square error is increased compared with that one in [56].

Another technique presented in [63] tried to predict and select a carry bit in a proper way. This technique is suitable when the width of multiplier is small, but not applicable when the width is high, due to the fast growing computational cost of the prediction process.

Truncated multiplier can be illustrated as in Figure 3.1. All $(n-k)$ less significant columns in the matrix are omitted, they are not participate in to form the product

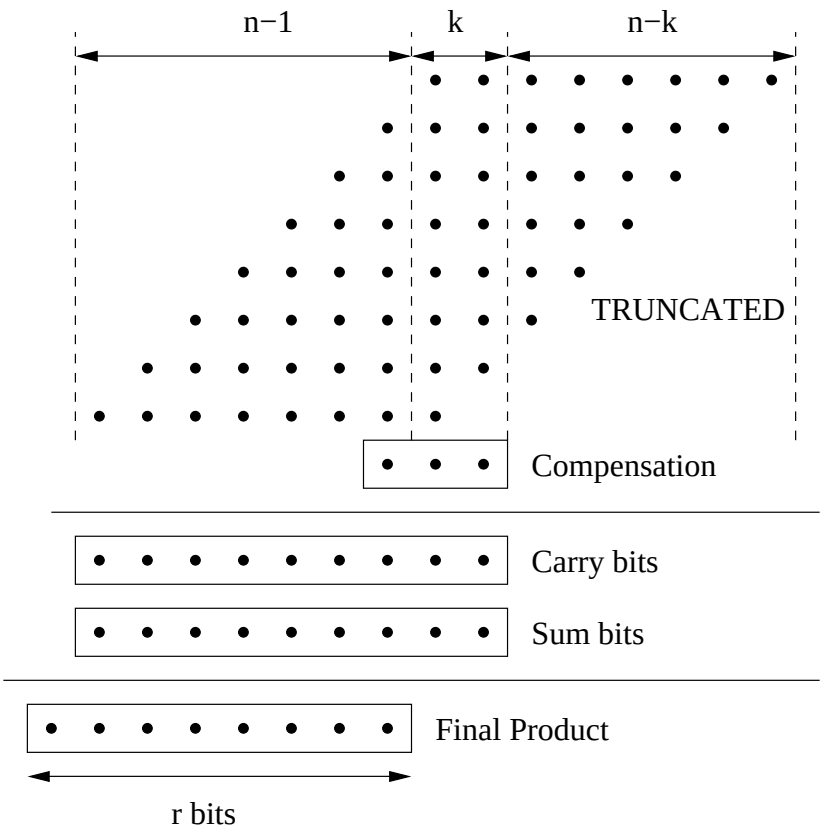

Figure 3.1: Product Matrix of Truncated Multiplier

result. As can be seen in this figure, only $(n + k - 1)$ most significant columns are used to compute the product. After that, the product is rounded to r-bit. Because of the truncation and rounding actions, there is a difference between the real result and the truncated result. To make the value of this error becomes smaller, an amount of correction is needed. Depending on the method of compensating for the error is the type of truncated multipliers.

3.1 Truncated Multiplication with Correction Constant (CCT)

Parallel multipliers are typically implemented as either carry-save array or tree multiplier [35]. An unsigned n-bit multiplicand A is multiplied by an unsigned n-bit multiplier B will produce an unsigned $2n$-bit product P. For fractional number, the value of A, B, P can be expressed as in the Equation 2.1.

In many systems, the $2n$-bit products are rounded to r-bit avoid growth in word size. This idea was introduced in [40] and then developed in [41]. In this method, a constant is added to truncated partial product matrix for compensating the value of the less significant columns that are truncated. As can be seen in Figure 3.1, define P is the true product, E_{reduct} is the error that caused by the truncation of the less significant columns, E_{round} is an error that caused by rounding to r-bit, C is the constant for compensating error, and $\hat{P}$ is the computed product, they can be expressed as:

$$\hat{P} = P + E_{reduct} + E_{round} + C \tag{3.1}$$

The error between computed $\hat{P}$ and real product P is

$$\epsilon = \hat{P} - P = E_{reduct} + E_{round} + C \tag{3.2}$$

To minimize this error, the correction constant C is selected as the inverse of E_{reduct} + E_{round} that depends on the value of input A and B. As can be seen, the probability of any input bit a_i or b_j being one is$\frac{1}{2}$, therefore the partial product bit $a_i b_j$ being one is $\frac{1}{4}$, hence, the expectation value of this partial product bit is $\frac{2^{-(m+n)+i+j}}{4}$. On the other hand, column k^{th} in the partial product matrix has $(k+1)$-bit, so the reduction error is the inverse of the sum of all partial product bits in the truncated area.

$$E_{reduct} = -\frac{1}{4}\sum_{i=0}^{n-k-1}(i+1)2^{-2n+i} \tag{3.3}$$

Figure 3.1 also shows that $n+k-1$ columns of the most significant part are used to compute $n+k$-bit product, that is rounded to r bits. Therefore, all bits from column $(n-k)^{th}$ to $(r-1)^{th}$ cause rounding error. Because the probability of p_i being one is $\frac{1}{2}$, the rounding error can be calculated as

$$E_{round} = -\frac{1}{2}\sum_{i=n-k}^{r-1}2^{-2n+i} \tag{3.4}$$

The total error that caused by truncation and rounding operations is

$$E_{total} = E_{reduct} + E_{round} \tag{3.5}$$

The constant C is obtained by rounding E_{total} to $r+k$ fractional bits such that

$$C_{CCT} = \frac{round(2^{r+k}E_{total})}{2^{r+k}} \tag{3.6}$$

where $round(x)$ indicates that x is rounded to the nearest integer. For example, if $n = r = 8$ and $k = 2$, errors and correction constant C can be calculated as:

$$\begin{aligned}
E_{reduct} &= -\frac{1}{4}\sum_{i=0}^{5}(i+1)2^{-16+i} \\
&= -0.0012245178222656 25 \\
E_{round} &= -\frac{1}{2}\sum_{i=6}^{7}2^{-16+i} \\
&= -0.00146484375 \\
E_{total} &= E_{reduct} + E_{round} \\
&= -0.002689361572265625 \\
C_{CCT} &= -\frac{round(2^{n+k}E_{total})}{2^{n+k}} \\
&= 0.001953125 \\
&= 2^{-9}
\end{aligned} \tag{3.7}$$

Figure 3.2 represents partial product matrix and Figure 3.3 is hardware implementation of this example. A specialized half adder (SHA) is used for adding the constant C to proper columns in the matrix. The architecture of a SHA is equivalent to a MFA, but it has an internal input which is set to "1" corresponded to the value of the constant C. Figure 3.4 is a hardware implementation of a SHA unit. As can be seen in Figure 3.3, to save more area, a reduced half adder (RHA) and a reduced full adder (RFA), which are the same as a HA and a FA, respectively, but produce only a carry output.

To improve performances of this multiplier, one of three methods in Section 2.3 is applied. Figure 3.5 is Dadda tree diagram for this truncated multiplier, in which, a bit $'1'$ in the column 1^{th} is represented for correction constant C.

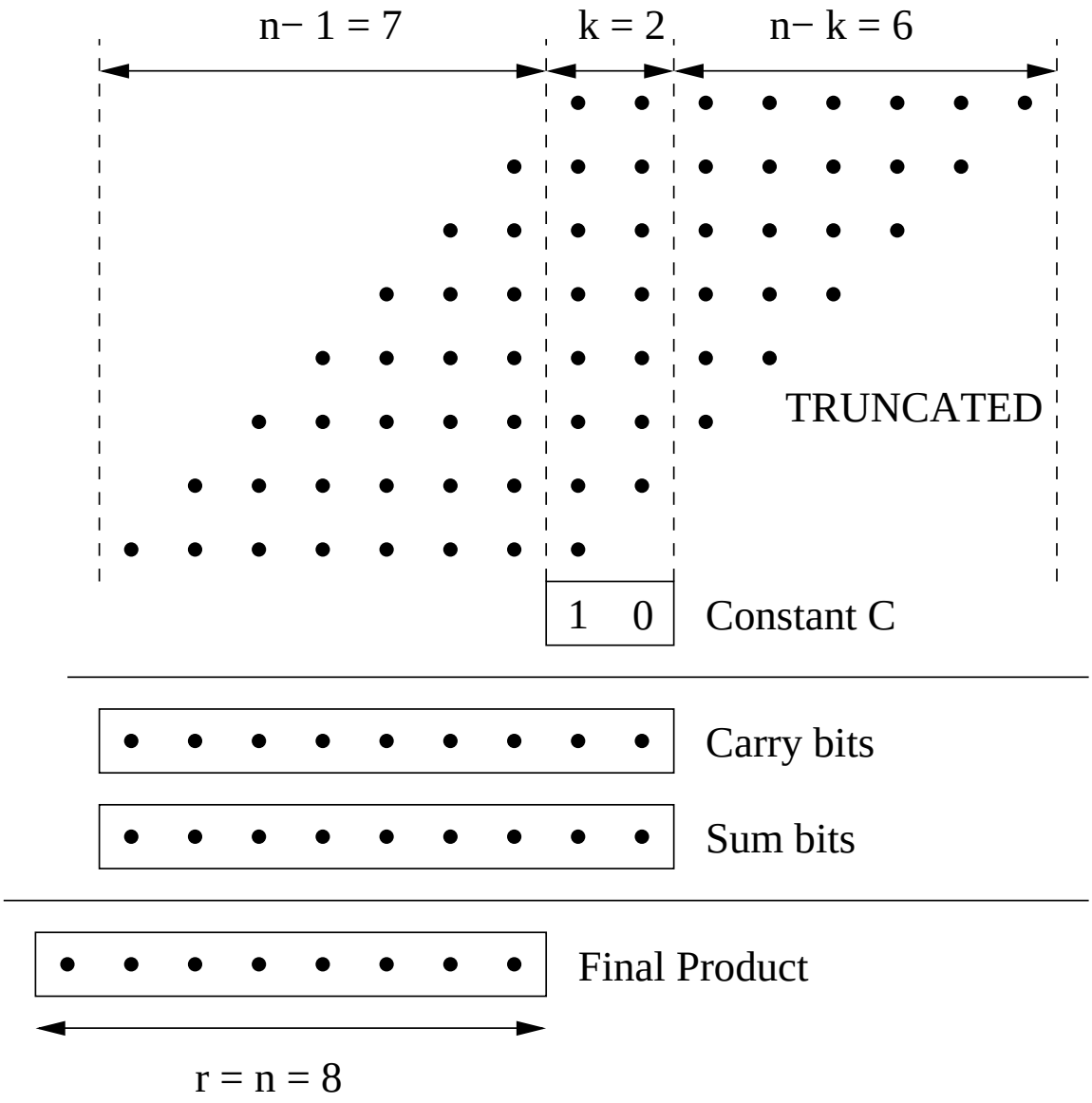

Figure 3.2: PPM of 8×8-bit Truncated Multiplier in CCT with $k = 2$.

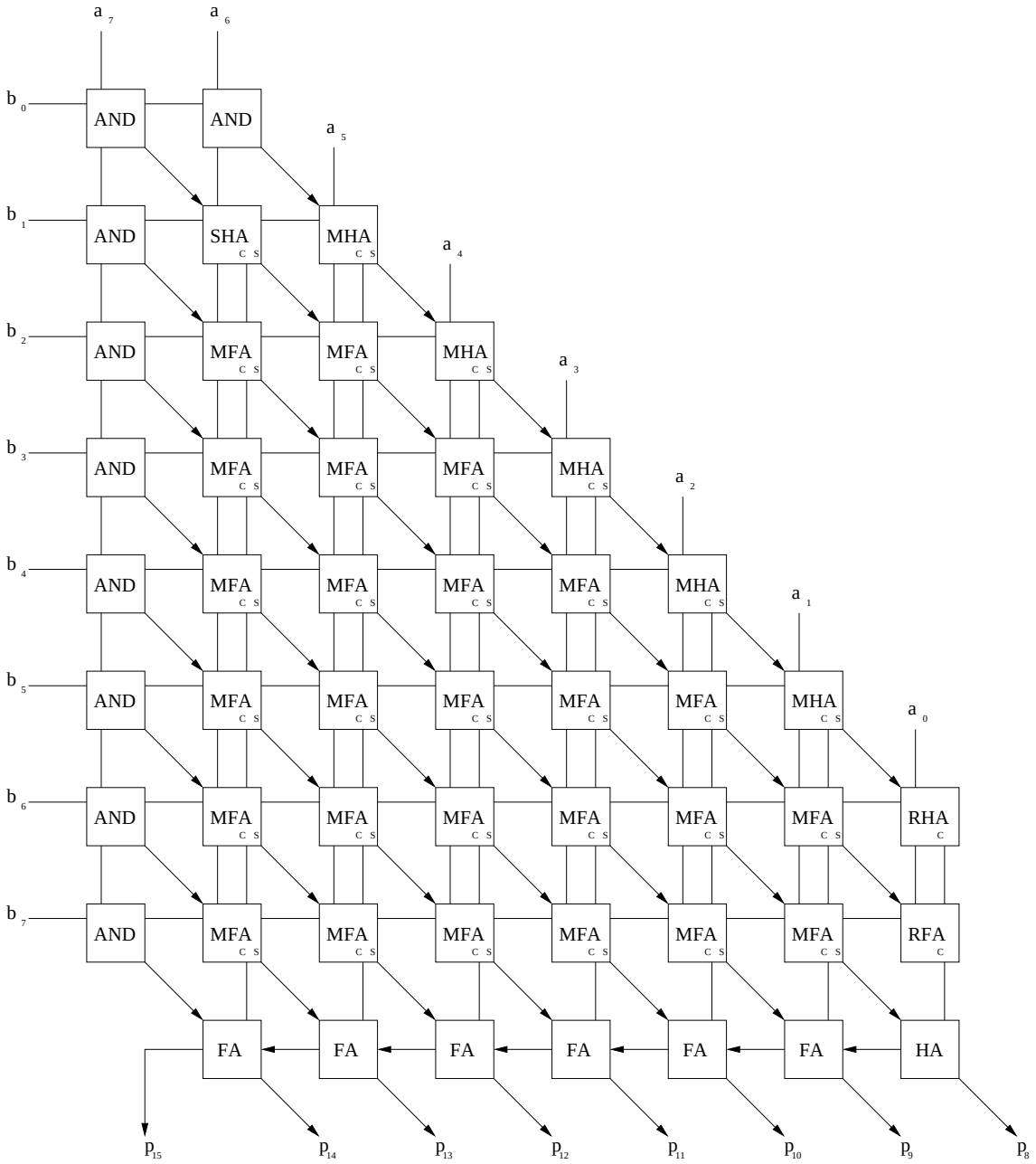

Figure 3.3: 8×8-bit CCT with $k = 2$ in CSAM Implementation.

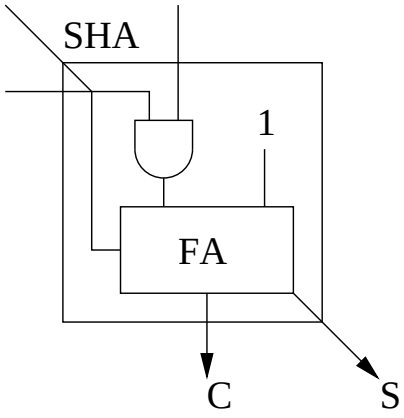

Figure 3.4: Special Half Adder.

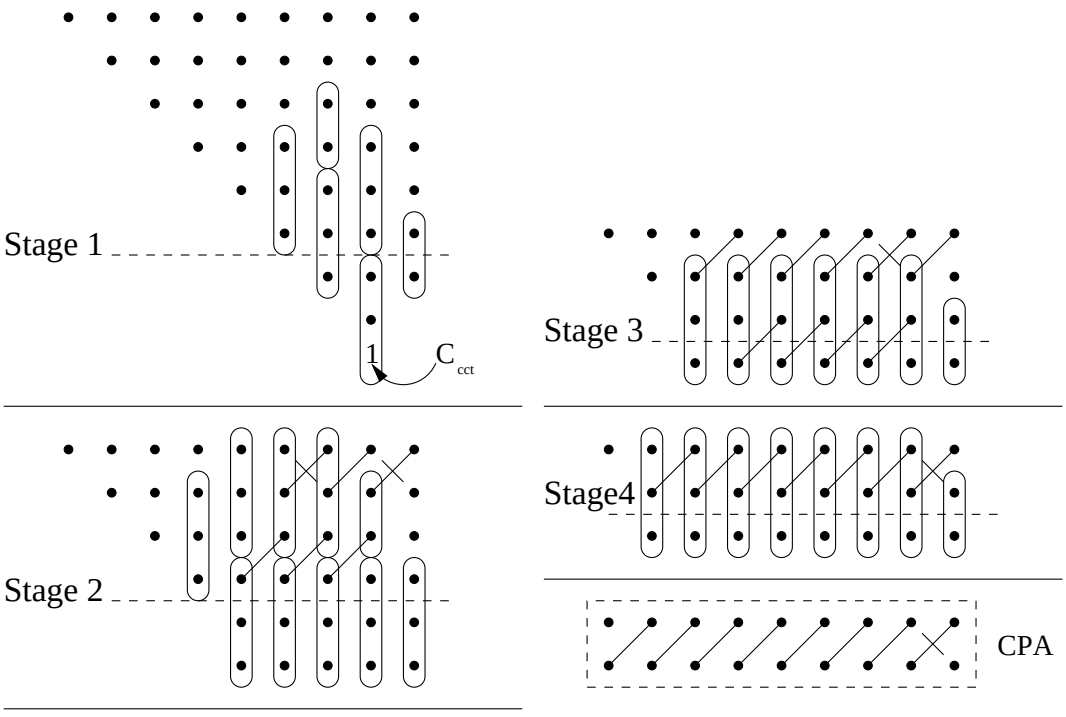

Figure 3.5: 8×8-bit CCT with $k = 2$ in Dadda Tree Reduction.

As can be seen, the maximum absolute error occurs when all partial product bits in truncated area and the bits in the column $2n-r-k$ to $2n-r-1$ are $'1'$ or they are all $'0'$. When they are all "0", the absolute error is C. When they are all "1" the absolute error is

$$\epsilon = \sum_{q=0}^{2n-r-k-1} (q+1) \times 2^{-2n+q} + 2^{-r}(1-2^{k}) \tag{3.8}$$

So the maximum absolute error is

$$E_{max} = max(C, \sum_{q=0}^{2n-r-k-1} (q+1) \times 2^{-2n+q} + 2^{-r}(1-2^{k})) \tag{3.9}$$

3.2 Truncated Multiplication with Variable Correction (VCT)

Truncated multiplier with variable correction was introduced in [44]. To compensate the error made by truncation, the partial product bits in the column $(n-k-1)^{th}$ are added to the column $(n-k)^{th}$. For rounding error, a constant is added through column $(n-2)^{th}$ to column $(n-k)^{th}$, so the value of C_{round} is

$$C_{round} = 2^{-n-1}(1-2^{-k+1}) \tag{3.10}$$

For example, 8×8-bit VCT with $k=2$, C_{round} can computed as

$$C_{round} = 2^{-8-1}(1-2^{-2+1}) = 2^{-10} \tag{3.11}$$

Figure 3.6 shows the partial product matrix for this example. Six terms in the column 5^{th} and one bit $'1'$ for C_{round} are added to the column 6^{th}. Like the CCT method, the VCT multiplier can also be implemented in $CSAM$, $Wallace$, Dadda, or RA tree. Figure 3.7 represents this multiplier in a $CSAM$ implementation, and Figure 3.8 is Dadda tree implementation. The maximum error of VCT can be calculated as

$$|E_{VCT}| = 2^{-1} + 2^{-r-k-1} + \sum_{q=1}^{\lfloor \frac{n-k}{2} \rfloor} (n-k+2-2q)2^{-r-k-2q-1} \tag{3.12}$$

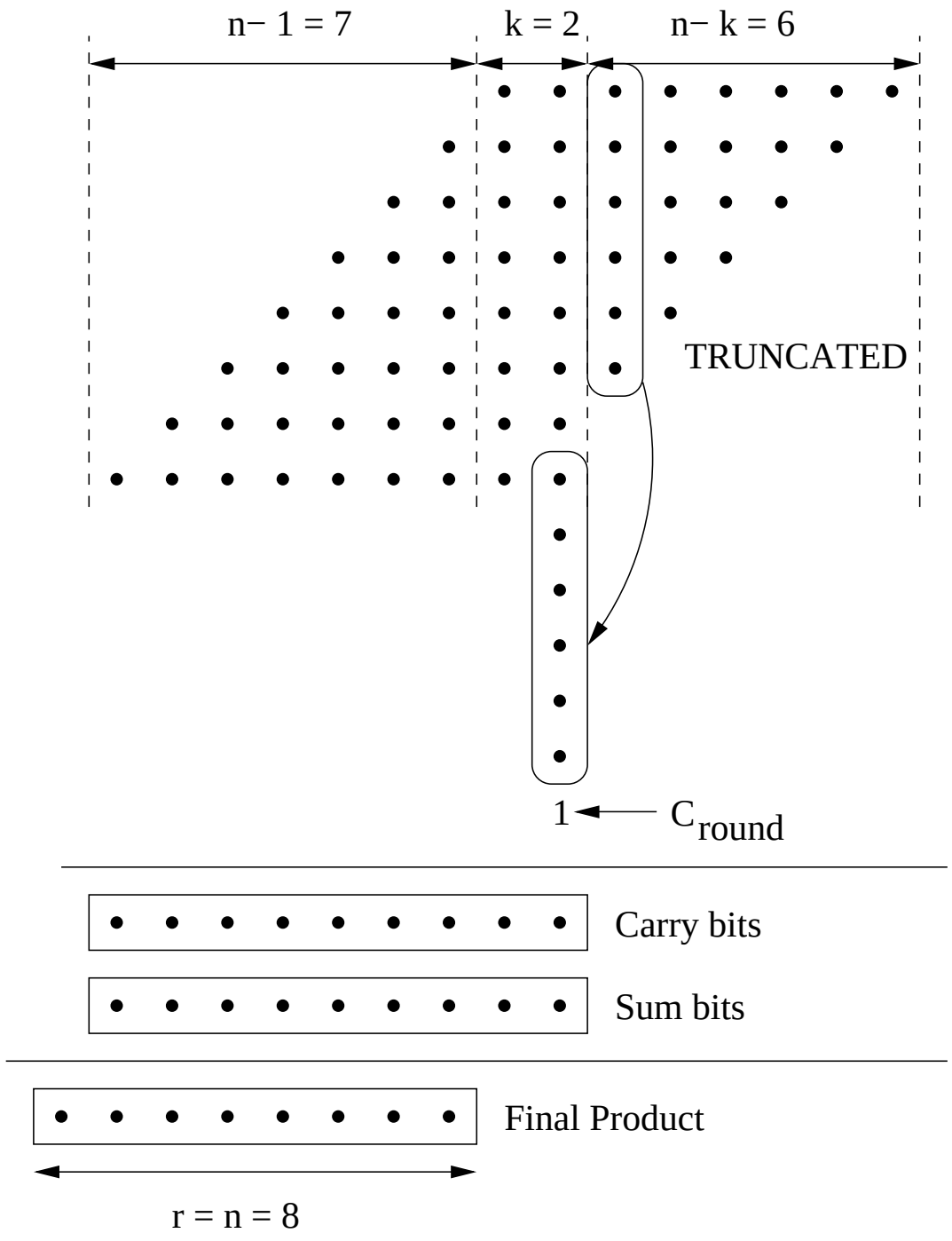

Figure 3.6: PPM of 8×8-bit VCT with $k = 2$.

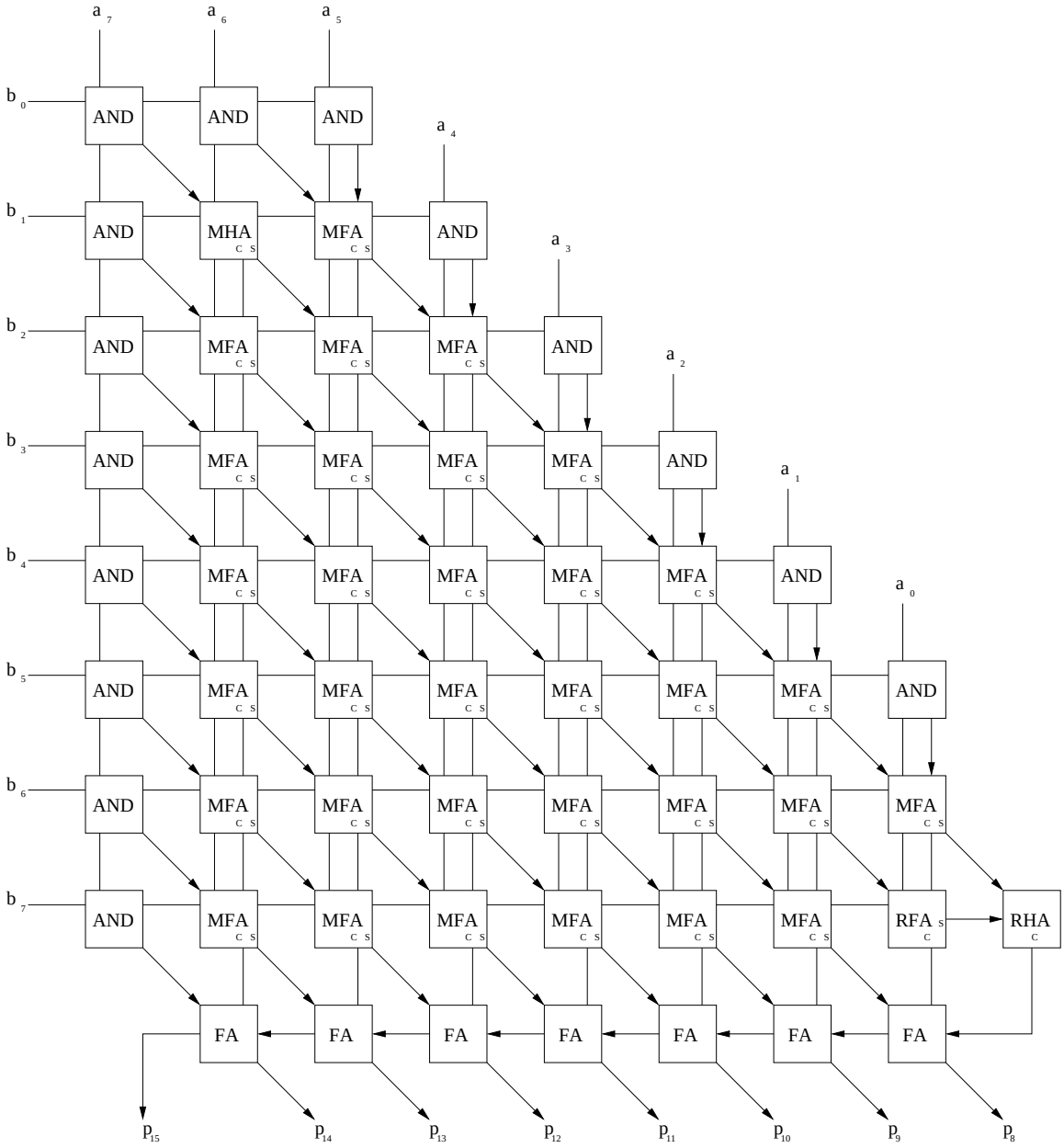

Figure 3.7: 8×8-bit VCT with $k = 2$ in CSAM Implementation.

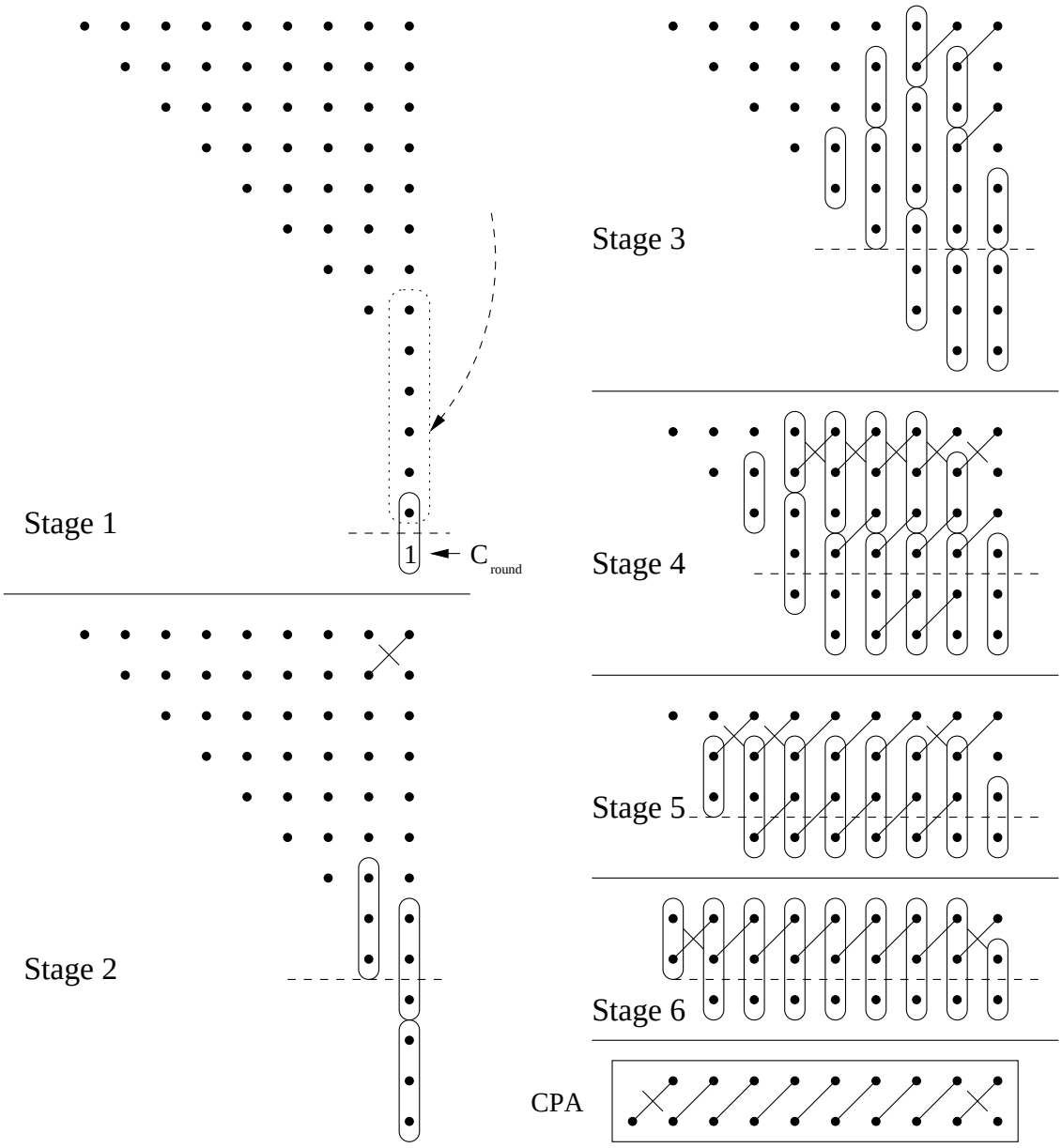

Figure 3.8: 8×8-bit VCT with $k = 2$ in Dadda Tree Reduction.

These errors occur when two operand and the k product bits from the column $(n-k)^{th}$ to $(n-1)^{th}$ are

$$\begin{aligned} a_{n-k-1}\cdots a_0, b_{n-k-1}\cdots b_0 &= \begin{cases} 0x\overline{x}\cdots x\overline{x}1, 0x\overline{x}\cdots x\overline{x}1 & \text{if } n-k \text{ is even} \\ 0x\overline{x}\cdots \overline{x}x1, 0x\overline{x}\cdots \overline{x}x1 & \text{if } n-k \text{ is odd} \end{cases} \\ p_{n-1}\cdots p_{n-k} &= 1\cdots 1 \end{aligned} \tag{3.13}$$

Where x is either "0" or "1". When these inputs fall in these patterns, all bits in the column $(n-k-1)^{th}$ are "0". Therefore, the partial product bits in truncated area make the maximum error occurs with the k-bit truncated product from $(n-1)^{th}$ to $(n-k)^{th}$. The maximum error of the k-bit final product from $(n-1)^{th}$ to $(n-k)^{th}$ is a half of unit in the last place of the n-bit result since the constant rounding C_{round} is added.

3.3 Truncated Multiplication with Hybrid Correction(HCT)

According to previous sections, the maximum error of VCT is less than that one of CCT for a given value of r and k, but the area of VCT is greater than that one of CCT because it uses more partial product bits in the PPM. To improve the VCT, a new method that use both constant and variable correction to make a compromise between both method was introduced in [28]. Since CCT has a maximum absolute error when all bit in truncated area is "1", and VCT has maximum absolute error when all that bits close to "0", HCT multiplier achieves a lower average and maximum absolute error compared to CCT and VCT multipliers.

To implement HCT, a new parameter p, that is the percentage of variable correction is used. This parameter shows that how many partial product bits in the column $(n-k-1)^{th}$ will be used to add into the next column. The number of that bits is computed as:

$$N_{HCT} = \lfloor (N_{VCT} \times p) \rfloor \tag{3.14}$$

The constant for HCT can be expressed as

$$C_{HCT} = \frac{1}{2} \times 2^{-r-k-1} \times N_{HCT} \tag{3.15}$$

A new constant is established based on the difference between the new variable and the constant in CCT method in Equation 3.6. Hence, rounding constant for this method can be expressed as

$$C_{round_{HCT}} = \frac{round((C_{CCT} - C_{HCT}) \times 2^{r+k})}{2^{r+k}} \tag{3.16}$$

Using the same example like in the previous section $n = r = 8$, $k = 2$ and $p = 0.5$, parameters of HCT can be calculated as

$$\begin{aligned} N_{HCT} &= \lfloor (6 \times 0.5) \rfloor \\ &= 3 \\ C_{HCT} &= \frac{1}{2} 2^{-11} \times 3 \\ &= 3^{-12} \\ C_{round_{HCT}} &= \frac{round((2^{-9} - 3 \times 2^{-12}) \times 2^{10})}{2^{10}} \\ &= 2^{-10} \end{aligned} \tag{3.17}$$

Because $N_{HCT} = 3$, it means that only 3 bits in the column 5^{th} are used to add to the next column. Not lose the generality, these terms a_4b_1, a_2b_3 and a_0b_5 are chosen. Figure 3.9 is $CSAM$ and Figure 3.10 is Dadda tree implementation of a HCT multiplier for this example.

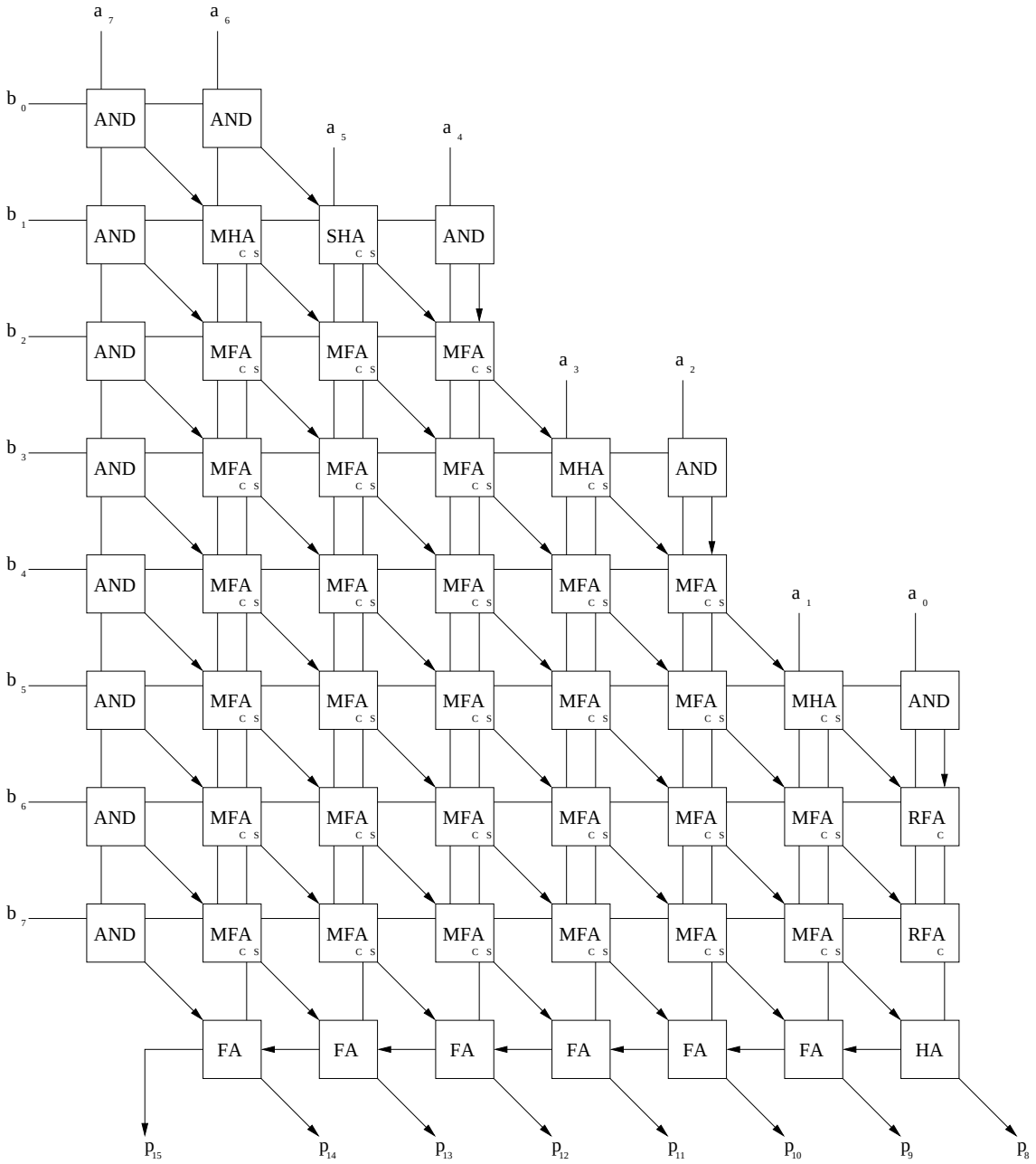

Figure 3.9: 8×8-bit HCT with $k = 2$, $p = 0.5$ in CSAM Implementation.

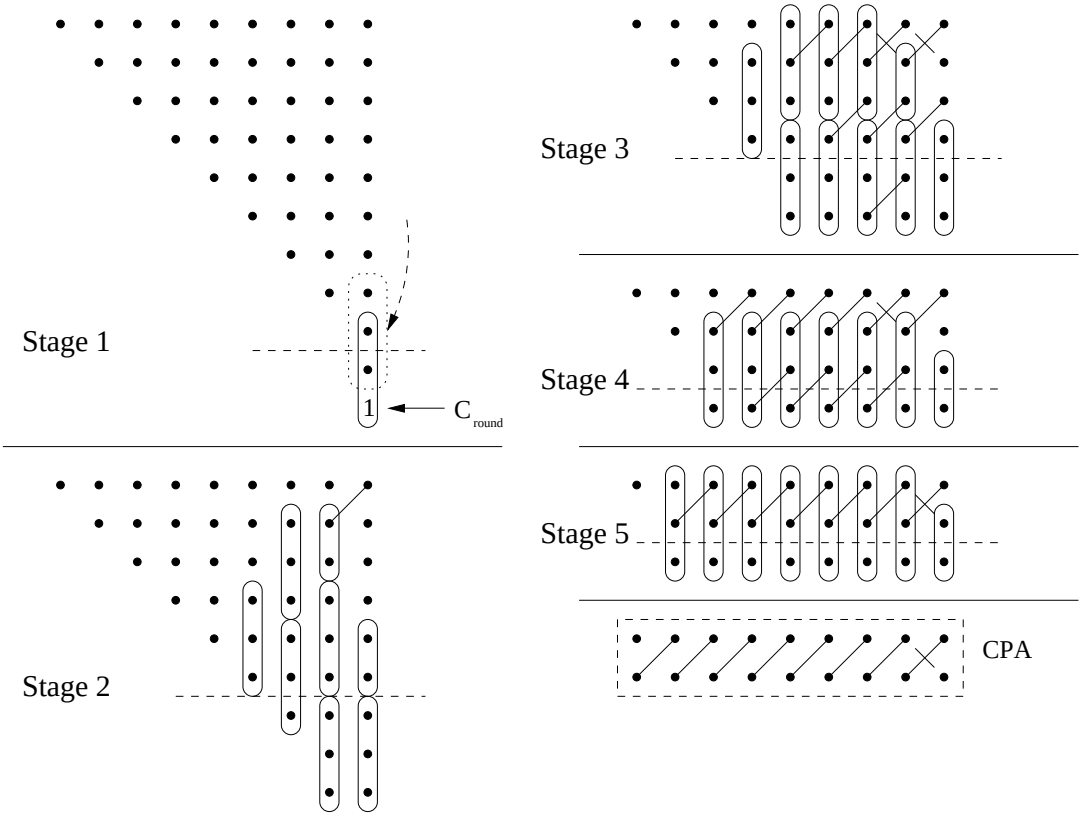

Figure 3.10: 8×8-bit HCT with $k = 2$, $p = 0.5$ in Dadda Tree Implementation.

CHAPTER 4

Squaring Unit Architectures

4.1 Unsigned Squaring Unit Architectures

Theoretically, a n-bit unsigned integer X is input operand and P is the output of a squaring unit, can be express as

$$\begin{aligned} X &= x_{n-1}x_{n-2}\dots x_1x_0 \\ &= \sum_{i=0}^{n-1} x_i \cdot 2^i \\ P &= X^2 \\ &= (\sum_{i=0}^{n-1} x_i \cdot 2^i)^2 \\ &= \sum_{i=0}^{n-1} x_i \cdot 2^{2i} + \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} x_i \cdot x_j \cdot 2^{i+j+1} \end{aligned} \tag{4.1}$$

Generally, to calculate an output of a squaring unit, the same number is supplied to the inputs of a multiplier as the multiplicand and the multiplier. This is simplest way, but a regular multiplier used as a squaring unit will be redundant [64], and also requires more area, power and delay [65]. To improve its performance, a dedicate squaring unit is necessary

4.1.1 Folded Squaring Units

The folding technique, first introduced in [66] and then developed in [67], utilized the symmetrical characteristic of squarer's PPM to reduce the number of its bits and height by nearly a half as in the Equation 4.1. The folding technique for squaring units employs the following steps:

1. Optimization of PPM

   (a) A pair of partial product bits $x_i \cdot x_j$ and $x_j \cdot x_i$ with $i \neq j$ in the column $(i+j)^{th}$ is reduced to one bit $x_i x_j$, then shifted to the left one position in the column $(i+j+1)^{th}$.

   (b) All bits $x_i \cdot x_i$ in the column $(2i)^{th}$ are reduced to x_i and kept in the same column.

2. The matrix height reduction is applied by column compression techniques, such as *Dadda* [5], to obtain the carry/sum arrays.

3. Applying a CPA to the carry and sum arrays to obtain the final result.

According to [68], the form of the folded partial product matrix varies slightly based on the value of n is whether odd or even. Figure 4.1 and 4.2 illustrate folded partial product matrix for $n = 7$ and $n = 8$, respectively. Due to the symmetry in the PPM of a squaring unit, specialized n-bit squaring circuits requires less hardware and delay than those ones in a regular $n \times n$ multiplier.

4.1.2 Merged Squaring Units

The merging technique presented in [69] and developed in [68, 70] augments the conventional folding technique, and reduces the critical path by shifting the partial products on the diagonal by one column to the left. Depending on whether the value of n is odd or even, the height of the PPM is reduced by one compared with that one in folded technique. The idea of this technique is combine to bits $x_i \cdot x_{i-1}$ and x_i in the row $(2i)^{th}$ of the folded PPM together. The equation of this technique can be represented as

$$(x_i \cdot x_{i-1} + x_i) \cdot 2^{2i} = x_i \cdot x_{i-1} \cdot 2^{2i+1} + x_i \cdot \overline{x_{i-1}} \cdot 2^{2i} \tag{4.2}$$

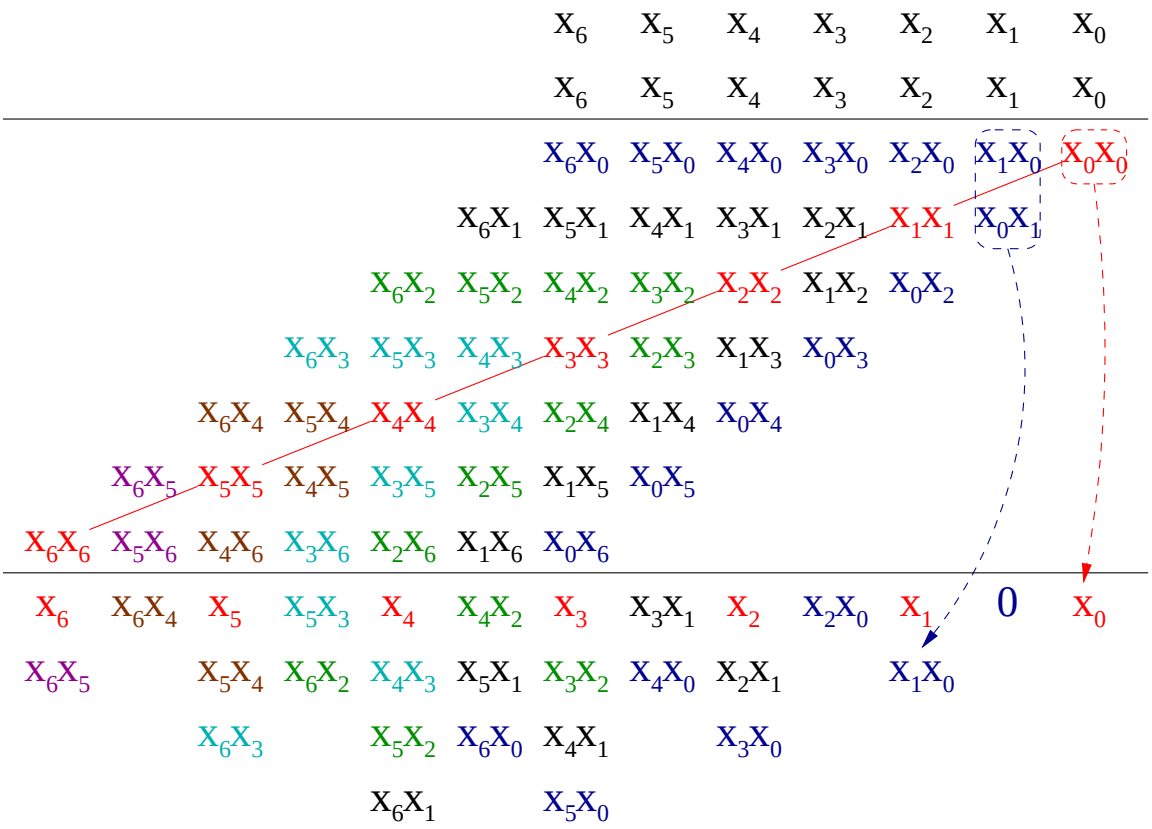

Figure 4.1: PPM of Folded Squaring Unit with $n = 7$

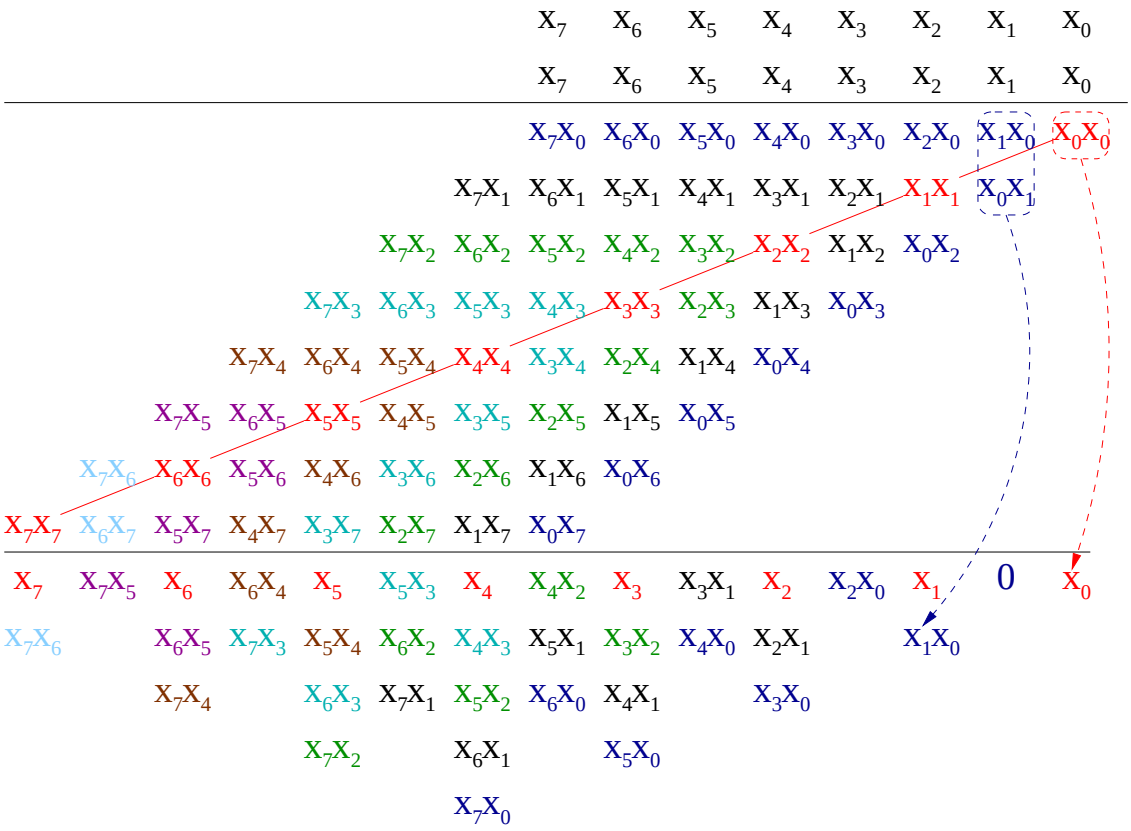

Figure 4.2: PPM of Folded Squaring Unit with $n = 8$

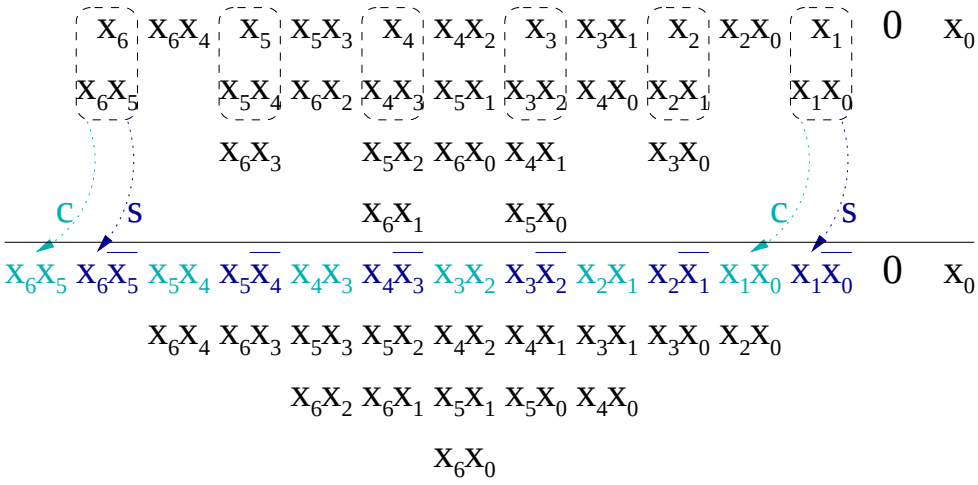

Figure 4.3: PPM of Merged Squaring Unit with $n = 7$

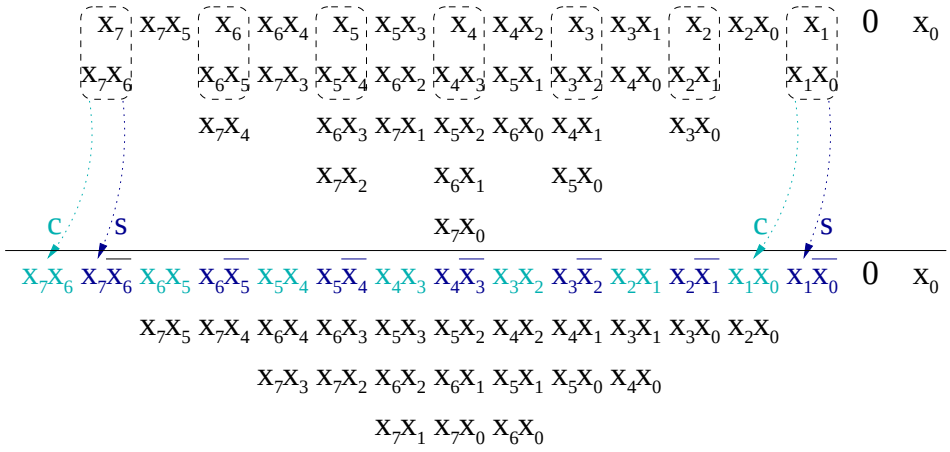

Figure 4.4: PPM of Merged Squaring Unit with $n = 8$

Applied this equation to the folded PPM, $n-1$ terms are moved one column to the left hand side. The same like in the folded method, the merged PPM is slightly changed when the length of the input n is odd or even. Figure 4.3 and 4.4 illustrate PPM for this method with $n = 7$ and $n = 8$. As observed in these figures, when n is even, the height of merged PPM is smaller than that one in the folded PPM.

4.1.3 Divide and Conquer Squaring Unit

This method was presented in [71, 72], bases on $Vedic$ mathematics. A squaring unit base on this method is implemented by the following steps

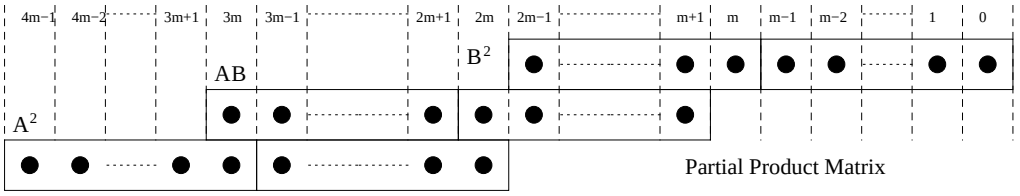

Figure 4.5: PPM of a Squaring Unit in Divide and Conquer Method

1. Step 1: The input operand X is divided into two parts $A = x_{2m-1} \dots x_m$ and $B = x_{m-1} \dots x_0$, each part contains m-bit, and the final result P can be represented as

$$\begin{aligned} X &= A \cdot 2^m + B \\ P &= X^2 \\ &= (A \cdot 2^m + B)^2 \\ &= A^2 \cdot 2^{2m} + A \cdot B \cdot 2^{m+1} + B^2 \end{aligned} \tag{4.3}$$

   Because each operand A and B has m-bit length, each term A^2, $A \cdot B$, B^2 will have $2m$-bit length. Therefore, when all the value of A^2, $A \cdot B$, B^2, are known, PPM of this squaring unit is established and the value of P can be computed. This means that this method can only be applied to input operands that are power of two and also each squaring operation is completed in repeated serial steps. Figure 4.5 is PPM for an $2m$-bit squaring unit.

2. Step 2: Repeat Step 1 until $n = 4$.

3. Step 3: Implement a squaring unit of 4-bit input. $X = x_3 \cdot x_2 \cdot x_1 \cdot x_0$, hence, $A = x_3 \cdot x_2$, $B = x_1 \cdot x_0$. All terms A^2, B^2, $A \cdot B$ will have only 4-bit length and have been expressed as in Figure 4.6

4.1.4 Proposed Squaring Unit

This section proposes a new method that utilizes more regularity of merged PPM to reduce more bits and its height. Following are stages for optimizing its PPM:

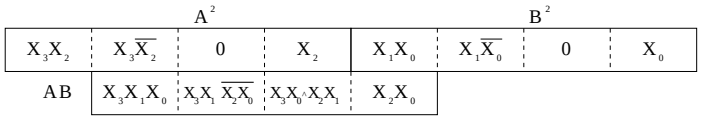

Figure 4.6: PPM of 4-bit Squaring Unit in Divide and Conquer Method

Table 4.1: Truth Table of Stage 1

x_i	x_{i-1}	x_{i-1}	$x_i x_{i-1}$	$x_{i-1}x_{i-2}$	C^1_{2i}	S^1_{2i-1}
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	1
1	0	0	0	0	0	0
1	0	1	1	0	0	1
1	1	0	0	0	0	0
1	1	1	1	1	1	0

- Stage 1: From the merged PPM, two bits $x_i \cdot x_{i-2}$ and $x_{i-1} \cdot x_{i-2}$ in the column $(2i-1)^{th}$ are added together. As can be seen in Table 4.1, the carry and sum bits of this adder can be calculated as:

$$\begin{aligned} S^1_{2i-1} &= x_i \cdot \overline{x_{i-1}} \cdot x_{i-2} + \overline{x_i} \cdot x_{i-1} \cdot x_{i-2} \\ C^1_{2i} &= x_i \cdot x_{i-1} \cdot x_{i-2} \end{aligned} \tag{4.4}$$

  Figure 4.7 is an illustration for this stage

- Stage 2: In this stage, carry bit from Stage 1 $C^1_{2i} = x_i \cdot x_{i-1} \cdot x_{i-2} = M$ is combined with 2 bits $x_i \cdot \overline{x_{i-1}} = N$ and $x_{i+1} \cdot x_{i-2} = Q$ in row $(2i)^{th}$ to form a special full adder. As can be seen in Table 4.1.4, carry and sum outputs of this special full adder are represented as:

$$\begin{aligned} S^2_{2i} &= x_{i+1} \cdot \overline{x_i} \cdot x_{i-2} + \overline{x_{i+1}} \cdot x_i \cdot x_{i-2} + x_i \cdot \overline{x_{i-1}} \cdot \overline{x_{i-2}} \\ C^2_{2i+1} &= x_{i+1} \cdot x_i \cdot x_{i-2} \end{aligned} \tag{4.5}$$

Table 4.2: Truth Table of A Special Adder in Stage 2

x_{i+1}	x_i	x_{i-1}	x_{i-2}	M	N	Q	C^2_{2i+1}	S^2_{2i}
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0
0	1	0	0	0	1	0	0	1
0	1	0	1	0	1	0	0	1
0	1	1	0	0	0	0	0	0
0	1	1	1	0	0	1	0	1
1	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	1
1	0	1	0	0	0	0	0	0
1	0	1	1	1	0	0	0	1
1	1	0	0	0	1	0	0	1
1	1	0	1	1	1	0	1	0
1	1	1	0	0	0	0	0	0
1	1	1	1	1	0	1	1	0

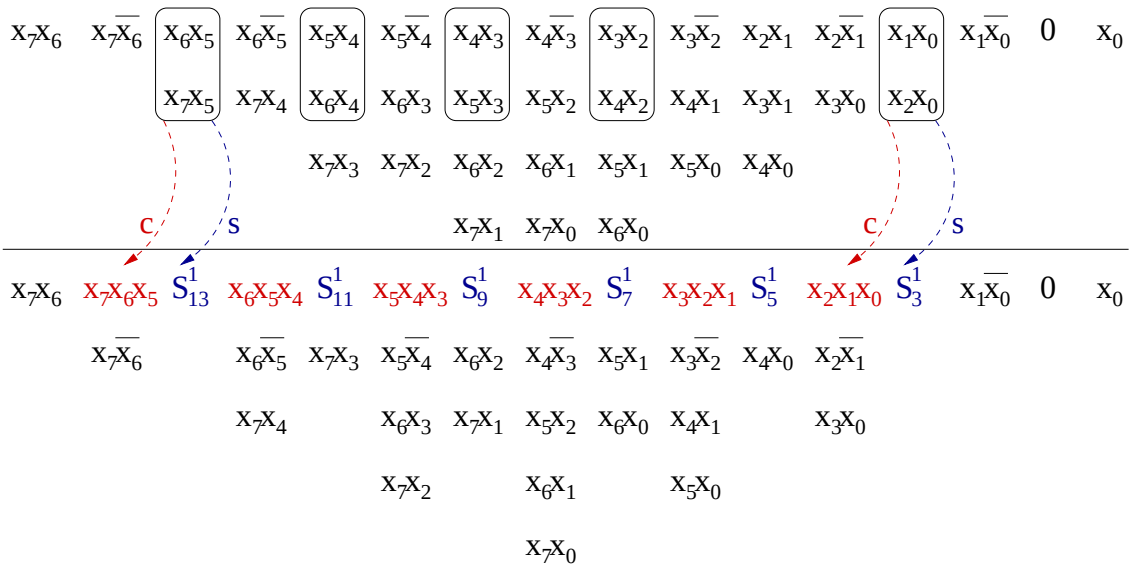

Figure 4.7: Optimization of a Squaring Unit at Stage 1

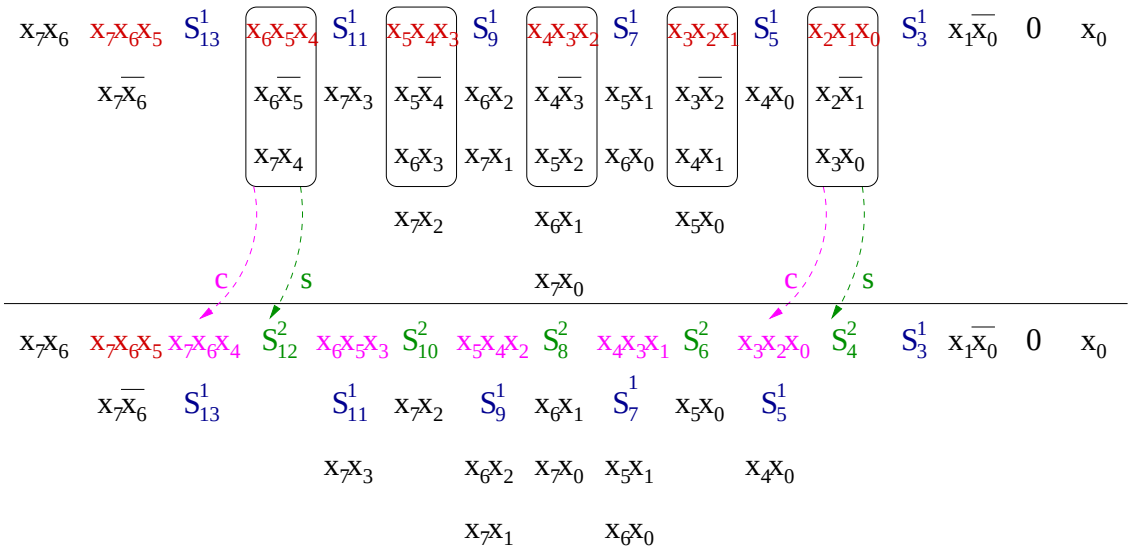

Figure 4.8: Optimization of a Squaring Unit at Stage 2

Figure 4.8 is an illustration for this stage

- Stage 3: In this stage, carry and sum bits from previous stages $C^2_{2i+1} = x_{i+1} \cdot x_i \cdot x_{i-2}$ and $S^1_{2i+1} = x_{i+1} \cdot \overline{x_i} \cdot x_{i-1} + \overline{x_{i+1}} \cdot x_i \cdot x_{i-1}$ are combined together. Because these two bits are not equal to 1 at the same time, therefore, the result will be only 1 bit, which means that one bit in PPM can be saved. As can be seen in Table 4.1.4, the sum of these bits can be expressed as:

Table 4.3: Truth Table of Stage 3

x_{i+1}	x_i	x_{i-1}	x_{i-2}	C^2_{2i+1}	S^1_{2i+1}	S^3_{2i+1}
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	1	1
1	0	1	1	0	1	1
1	1	0	0	0	0	0
1	1	0	1	1	0	1
1	1	1	0	0	0	0
1	1	1	1	1	0	1

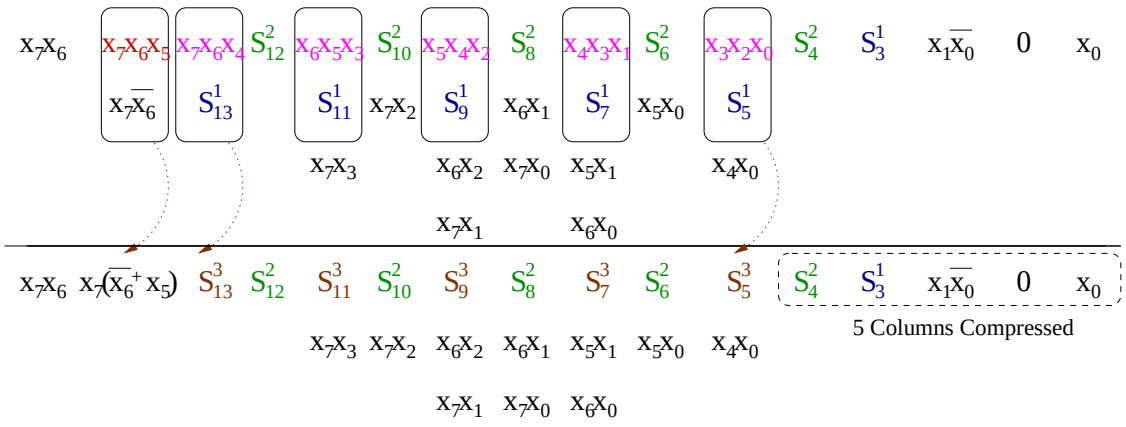

Figure 4.9: Optimization of a Squaring Unit at Stage 3

$$S^3_{2i+1} = x_{i+1} \cdot x_i \cdot x_{i-2} + x_{i+1} \cdot \overline{x_i} \cdot x_{i-1} + \overline{x_{i+1}} \cdot x_i \cdot x_{i-1} \tag{4.6}$$

Figure 4.9 is an illustration for this stage

4.2 Signed Squaring Unit Architectures

4.2.1 Booth Folding Squaring Unit

A squaring unit using Booth Folding technique was presented in [10,11], in which the symmetry property of a squaring unit and Booth 2 feature are combined to improve its performance. A $2m$-bit input X of a signed number represented using two's complement notation can be expressed as Equation 2.7. The partial product matrix of this unit can be seen in the Figure 4.10. Row i^{th} $(i = 0 \cdots m-2)$ in the folded matrix can be expressed as:

$$\begin{aligned}
R_i &= (P_i \cdot 2^3 + C_i) \\
P_i &= 2 \cdot B_{m-1} \cdot B_i \cdot 2^{2(m-1+i)} + 2 \cdot B_{m-2} \cdot B_i \cdot 2^{2(m-2+i)} + \cdots \\
&\quad +2 \cdot B_{i+2} \cdot B_i \cdot 2^{2(i+2+i)} + 2 \cdot B_{i+1} \cdot B_i \cdot 2^{2(i+1+i)} \\
&= (B_{m-1} \cdot 2^{2m-2i-4} + B_{m-2} \cdot 2^{2m-2i-6} + \cdots \\
&\quad +B_{i+2} \cdot 2^2 + B_{i+1}) \cdot B_i \cdot 2^{4i+3} \\
C_i &= = B_i \cdot B_i \cdot 2^{4i}
\end{aligned} \tag{4.7}$$

Because B_i is encoded by using Booth 2 as in the Equation 2.6, so P_i can be viewed as

$$\begin{aligned} P_i &= [(-2 \cdot x_{2m-1} + x_{2m-2} + x_{2m-3}) \cdot 2^{2m-2i-4} \\ &\quad +(-2 \cdot x_{2m-3} + x_{2m-4} + x_{2m-5}) \cdot 2^{2m-2i-6} + \cdots \\ &\quad +(-2 \cdot x_{2i+5} + x_{2i+4} + x_{2i+3}) \cdot 2^2 + (-2 \cdot x_{2i+3} + x_{2i+2} \\ &\quad +x_{2i+1})] \cdot B_i \cdot 2^{4i+3} \\ &= [(-x_{2m-1} \cdot 2^{2m-2i-3} + x_{2m-2} \cdot 2^{2m-2i-4} + \cdots \\ &\quad +x_{2i+3} \cdot 2 + x_{2i+2}) + x_{2i+1}] \cdot (-2 \cdot x_{2i+1} + x_{2i} + x_{2i-1}) \cdot 2^{4i+3} \\ &= (W_i + x_{2i+1}) \cdot (-2 \cdot x_{2i+1} + x_{2i} + x_{2i-1}) \cdot 2^{4i+3} \end{aligned} \tag{4.8}$$

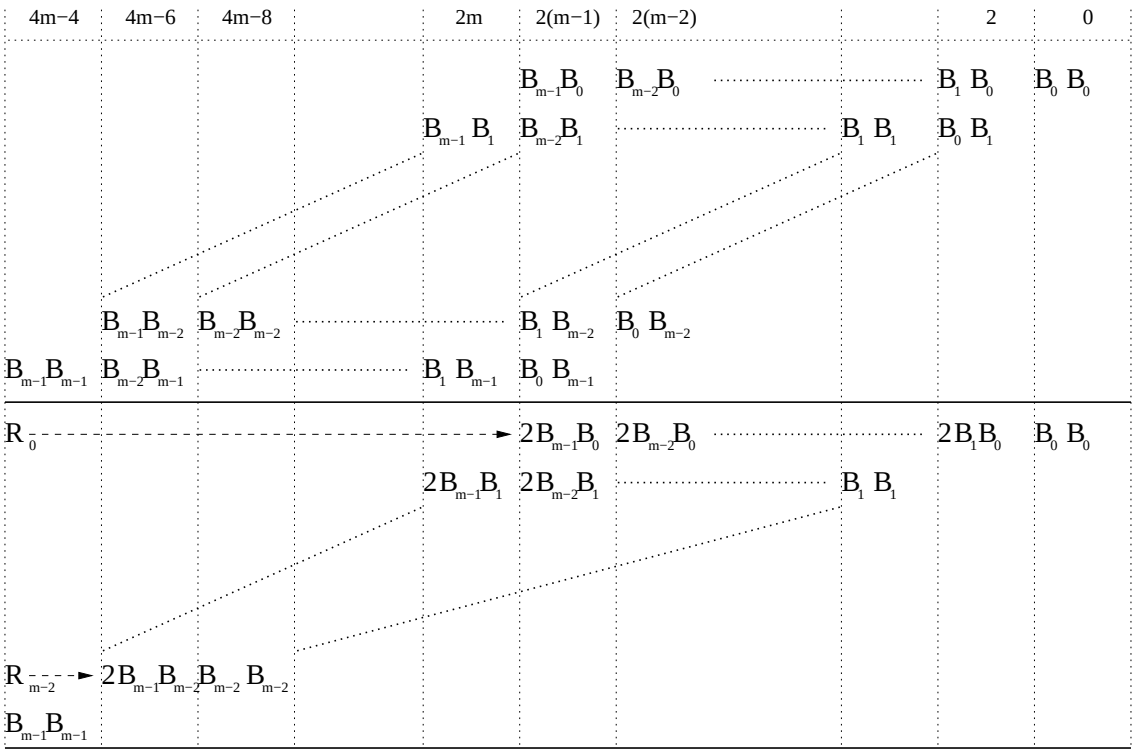

Figure 4.10: PPM of Folding Squaring Unit in Booth 2

where

$$\begin{aligned} W_i &= x_{2m-1}x_{2m-2}\cdots x_{2i+3}x_{2i+2} \\ &= -x_{2m-1}\cdot 2^{2m-2i-3} + x_{2m-2}\cdot 2^{2m-2i-4} + \cdots \\ &\quad +x_{2i+3}\cdot 2 + x_{2i+2} \end{aligned} \tag{4.9}$$

According to Equation 4.8, P_i can be calculates as following:

- When $x_{2i+1} = 0$, B_i is a positive number hence $P_i = W_i \cdot B_i \cdot 2^{4i+3}$.
- When $x_{2i+1} = 1$, B_i is a negative number, Equation 4.8 can be rewritten as

$$\begin{aligned} P_i &= (W_i + 1)\cdot B_i \cdot 2^{4i+3} \\ &= -(W_i + 1)\cdot |B_i| \cdot 2^{4i+3} \\ &= \overline{W_i}\cdot |B_i| \cdot 2^{4i+3} \end{aligned} \tag{4.10}$$

  where $\overline{W_i}$ obtained by complementing all bits of W_i:

$$\overline{W_i} = \overline{x_{2m-1}}\ \overline{x_{2m-2}}\cdots\overline{x_{2i+3}}\ \overline{x_{2i+2}} \tag{4.11}$$

Hence, Equation 4.8 can be expressed as

$$P_i = \begin{cases} W_i \cdot B_i \cdot 2^{4i+3}, & if \quad B_i \geq 0 \\ \overline{W_i}\cdot |B_i| \cdot 2^{4i+3}, & if \quad B_i < 0 \end{cases} \tag{4.12}$$

Because W_i is represented in two's complement, so P_i in the Equation 4.12 can be evaluated using a simple one's complement circuit [11]. As can be seen in the Equation 4.9, W_i has $(2m - 2i - 2)$-bit length, $-2 \leq B_i \leq 2$, hence, P_i should have $(2m - 2i - 1)$-bit length and runs from $lsb = (4i + 3)^{th}$ to $msb = (2m + 2i + 1)^{th}$. Truth Table 4.4 and logic equation for each bit of P_i can be established as a following equation:

Table 4.4: Truth Table for Partial Product P_i in Booth 2 Folding Technique

B_i	x_{2i+1}	x_{2i}	x_{2i-1}	P_i					
				msb	$msb-1$	$msb-2$	$\cdots\cdots$	$lsb+1$	lsb
0	0	0	0	1	0	0	$\cdots\cdots$	0	0
1	0	0	1	$\overline{x_{2m-1}}$	x_{2m-1}	x_{2m-2}	$\cdots\cdots$	x_{2i+3}	x_{2i+2}
1	0	1	0	$\overline{x_{2m-1}}$	x_{2m-1}	x_{2m-2}	$\cdots\cdots$	x_{2i+3}	x_{2i+2}
2	0	1	1	$\overline{x_{2m-1}}$	x_{2m-2}	x_{2m-3}	$\cdots\cdots$	x_{2i+2}	0
-2	1	0	0	x_{2m-1}	$\overline{x_{2m-2}}$	$\overline{x_{2m-3}}$	$\cdots\cdots$	$\overline{x_{2i+2}}$	0
-1	1	0	1	x_{2m-1}	$\overline{x_{2m-1}}$	$\overline{x_{2m-2}}$	$\cdots\cdots$	$\overline{x_{2i+3}}$	$\overline{x_{2i+2}}$
-1	1	1	0	x_{2m-1}	$\overline{x_{2m-1}}$	$\overline{x_{2m-2}}$	$\cdots\cdots$	$\overline{x_{2i+3}}$	$\overline{x_{2i+2}}$
-0	1	1	1	1	0	0	$\cdots\cdots$	0	0

$$\begin{cases} P_i[4i+3] &= (x_{2i+2} \oplus x_{2i+1}) \cdot (x_{2i} \oplus x_{2i-1}) \\ P_i[4i+4] &= (x_{2i+3} \oplus x_{2i+1}) \cdot (x_{2i} \oplus x_{2i-1}) \\ & \quad +(x_{2i+2} \oplus x_{2i+1}) \cdot (x_{2i+1} \oplus x_{2i}) \cdot \overline{(x_{2i} \oplus x_{2i-1})} \\ \vdots & \\ P_i[2m+2i] &= (x_{2m-1} \oplus x_{2i+1}) \cdot (x_{2i} \oplus x_{2i-1}) \\ & \quad +(x_{2m-2} \oplus x_{2i+1}) \cdot (x_{2i+1} \oplus x_{2i}) \cdot \overline{(x_{2i} \oplus x_{2i-1})} \\ P_i[2m+2i+1] &= (x_{2m-1} \oplus x_{2i+1}) \cdot (x_{2i+1} \oplus x_{2i}) \cdot \overline{(x_{2i} \oplus x_{2i-1})} \end{cases} \tag{4.13}$$

The most significant bit of P_i is inverted and hence, a constant is added to adjust the correction. On the other hand, C_i in the Equation 4.7 has three bits length and can be expressed as:

$$\begin{cases} C_i[2] = & (x_{2i+1} \oplus x_{2i}) \cdot \overline{(x_{2i} \oplus x_{2i-1})} \\ C_i[1] = & 0 \\ C_i[0] = & (x_{2i} \oplus x_{2i-1}) \end{cases} \tag{4.14}$$

So the partial product matrix of this unit can be illustrated as in Figure 4.11 and *Dadda* tree is applied to get the final result.

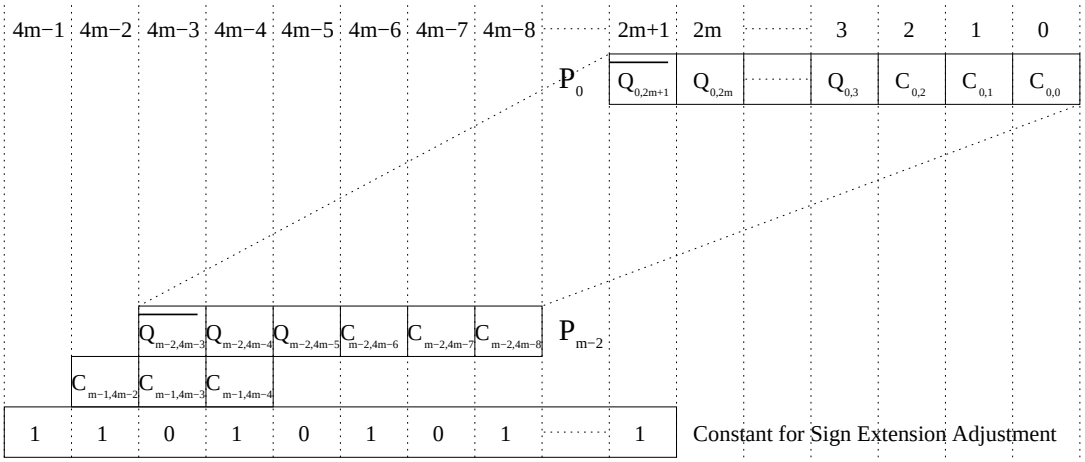

Figure 4.11: PPM of a Squaring Unit in Booth 2 Folding Technique

4.2.2 Booth 2 Left-to-Right Dual Recoding Squaring Unit

This method was presented in [73, 74]. A $2m$-bit input X of a signed number represented using two's complement notation can be expressed as in the Equation 2.7. Define X_i as follow

$$\begin{aligned} X_i &= x_{2i+1}x_{2i}\cdots x_1x_0 \\ &= -x_{2i+1}\cdot 2^{2i+1} + \sum_{k=0}^{2i} x_k \cdot 2^k \\ &= \overline{x_{2i+1}}x_{2i}\cdots x_1x_0 \end{aligned} \tag{4.15}$$

Following conclusions can be drawn:

$$\begin{aligned} X_{i+1} + X_i &= -x_{2i+3}\cdot 2^{2i+3} + \sum_{k=0}^{2i+2} x_k \cdot 2^k - x_{2i+1}\cdot 2^{2i+1} + \sum_{k=0}^{2i} x_k \cdot 2^k \\ &= -x_{2i+3}\cdot 2^{2i+3} + x_{2i+2}\cdot 2^{2i+2} + 2\cdot\sum_{k=0}^{2i} x_k \cdot 2^k \\ &= \overline{x_{2i+3}}x_{2i+2}x_{2i}\cdots x_1x_0 0 \\ &= Q_{i+1} \end{aligned} \tag{4.16}$$

$$\begin{aligned} X_{i+1} - X_i &= -x_{2i+3}\cdot 2^{2i+3} + \sum_{k=0}^{2i+2} x_k \cdot 2^k + x_{2i+1}\cdot 2^{2i+1} - \sum_{k=0}^{2i} x_k \cdot 2^k \\ &= (-2\cdot x_{2i+3} + x_{2i+2} + x_{2i+1})\cdot 2^{2i+2} \\ &= B_{i+1}\cdot 2^{2(i+1)} \end{aligned} \tag{4.17}$$

Table 4.5: Truth Table of P_i in Left-to-Right Dual Recoding Method

B_i	x_{2i+1}	x_{2i}	x_{2i-1}	P_i					
				$4i+1$	$4i$	$4i-1$	$\cdots\cdots$	$2i+2$	$2i+1$
0	0	0	0	0	0	0	$\cdots\cdots$	0	0
1	0	0	1	0	0	x_{2i-2}	$\cdots\cdots$	x_1	x_0
1	0	1	0	0	1	x_{2i-2}	$\cdots\cdots$	x_1	x_0
2	0	1	1	1	x_{2i-2}	x_{2i-3}	$\cdots\cdots$	x_0	0
-2	1	0	0	1	$\overline{x_{2i-2}}$	$\overline{x_{2i-3}}$	$\cdots\cdots$	$\overline{x_0}$	$1+1$
-1	1	0	1	0	1	$\overline{x_{2i-2}}$	$\cdots\cdots$	$\overline{x_1}$	$\overline{x_0}+1$
-1	1	1	0	0	0	$\overline{x_{2i-2}}$	$\cdots\cdots$	$\overline{x_1}$	$\overline{x_0}+1$
-0	1	1	1	0	0	0	$\cdots\cdots$	0	0

Equation 4.16 and 4.17 ensure that

$$\begin{aligned} X_{i+1}^2 - X_i^2 &= (X_{i+1} + X_i) \cdot (X_{i+1} - X_i) \\ &= Q_{i+1} \cdot B_{i+1} \cdot 2^{2(i+1)} \end{aligned} \tag{4.18}$$

From iteration Equation 4.18 , the output of a n-bit squaring unit can be expressed as

$$\begin{aligned} X^2 &= X_{m-1}^2 \\ &= \sum_{k=1}^{m-1} Q_k \cdot B_k \cdot 2^{2k} + B_0^2 \end{aligned} \tag{4.19}$$

Assuming $P_i = Q_i \cdot B_i \cdot 2^{2i}$, the truth table for P_i can be expressed as in the Table 4.5. According to this Table each bit of P_i can be expressed as

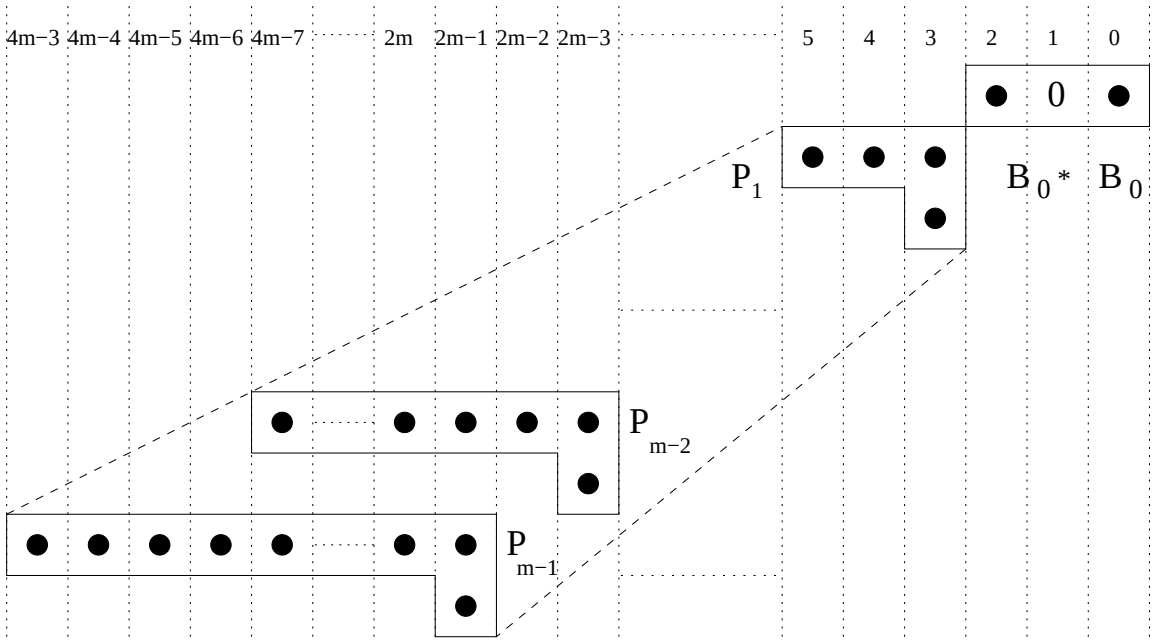

Figure 4.12: PPM of Squaring Unit in Booth 2 Left-to-Right Dual Recoding

$$
\begin{cases}
P_{i,2i+1}[0] &= (x_0 \oplus x_{2i+1}) \cdot (x_{2i} \oplus x_{2i-1}) + x_{2i+1} \cdot \overline{x_{2i}} \cdot \overline{x_{2i-1}} \\
P_{i,2i+1}[1] &= x_{2i+1} \cdot \overline{x_{2i}} + x_{2i+1} \cdot \overline{x_{2i-1}} \\
P_{i,2i+2}[0] &= (x_1 \oplus x_{2i+1}) \cdot (x_{2i} \oplus x_{2i-1}) + \overline{(x_0 \oplus x_{2i+1})} \cdot (x_{2i+1} \oplus x_{2i}) \cdot \\
&\quad \cdot \overline{(x_{2i} \oplus x_{2i-1})} \\
\vdots & \\
P_{i,4i-1}[0] &= (x_{2i-2} \oplus x_{2i+1}) \cdot (x_{2i} \oplus x_{2i-1}) + \overline{(x_{2i-3} \oplus x_{2i+1})} \cdot (x_{2i+1} \oplus x_{2i}) \cdot \\
&\quad \cdot \overline{(x_{2i} \oplus x_{2i-1})} \\
P_{i,4i}[0] &= \overline{(x_{2i-2} \oplus x_{2i+1})} \cdot (x_{2i+1} \oplus x_{2i}) + (x_{2i+1} \oplus x_{2i}) \cdot (x_{2i} \oplus x_{2i-1}) \\
P_{i,4i+1}[0] &= (x_{2i+1} \oplus x_{2i}) \cdot \overline{(x_{2i} \oplus x_{2i-1})}
\end{cases}
\tag{4.20}
$$

PPM of this unit is generated as in Figure 4.12, then *Dadda* tree is applied to get final result.

CHAPTER 5

Cubing Units

The simplest way to perform cubing operation is a using of traditional multipliers in serial fashion. That is, in the first multiplier, the input operand is multiplied by itself and the result of this multiplier is stored in a temporary register. The final result is obtained by multiplying the original operand by the stored result from the second multiplier. A disadvantage of this method is that latency of cubing operation will approximately be twice the amount of the multiplier. And, hence, when operand lengths increase the latency of the operation grows exponentially [25]. To improve this method, squaring unit [66, 68] is utilized for the intermediate result, and then a non-rectangular multiplier will generate the final result.

For parallel cubing computation, several methods were previously proposed. The first one in [67] suggested a reduced parallel cubing technique in which not only calculates the cube of an operand in parallel, but employs three reduction techniques that greatly reduce the height of the partial product generation. By doing that, the latency of the resulting CSA tree is reduced. The second one in [25] presents an extension to this technique by enhancing the method of [67] using a different ordering of the calculation as well as new partial product reduction technique.

Another recent technique presented in [71, 75] that using $Vedic$ mathematics to provide an efficient way of constructing a straight cubing system without using conventional multiplication methods. Although this method presents an easy way to compute the cube of a function using pen and paper, it does not lend itself to larger operand sizes. In this section, a new way for calculating cubing unit is presented by

dividing the input into small pieces, similar to the method that uses $Vedic$ mathematics, and then uses the three partial product reduction techniques in [67] to reduce the height of its PPM.

Define a $2m$-bit unsigned integer X is input operand and P is the output of cubing unit, they can be expressed as

$$\begin{aligned} X &= x_{2m-1}x_{2m-2}\ldots x_1x_0 \\ &= \sum_{i=0}^{2m-1} x_i \cdot 2^i \\ P &= X^3 \\ &= (\sum_{i=0}^{2m-1} x_i \cdot 2^i)^3 \\ &= \sum_{i=0}^{2m-1} x_i \cdot 2^{3i} + 3 \cdot \sum_{i=0}^{2m-2} \sum_{j=i+1}^{2m-1} x_i \cdot x_j \cdot (2^{2i+j} + 2^{i+2j}) \\ &\quad +6 \cdot \sum_{i=0}^{2m-3} \sum_{j=i+1}^{2m-2} \sum_{k=j+1}^{2m-1} x_i \cdot x_j \cdot x_k \cdot 2^{i+j+k} \end{aligned} \tag{5.1}$$

In traditional way, these bits in the Equation 5.1 are organized as in a PPM as in Figure 5.1. The problem with expressing this equation as a PPM, as in multiplication, is that the size of the partial product grows exponentially. Simplifying the output expression of P calculated by adding each row in this PPM from the right to the left hand side can produce equations for the number of bits that must be assimilated together as well as the height of the PPM. Therefore, the complexity of reducing the PPM results in the following items

$$\begin{aligned} PPA_{bit} &= (2m)^3 \\ PPA_{height} &= 2m + 2(2m-1) + \cdots + 2(m+1) + m \\ &= 3m^2 \end{aligned} \tag{5.2}$$

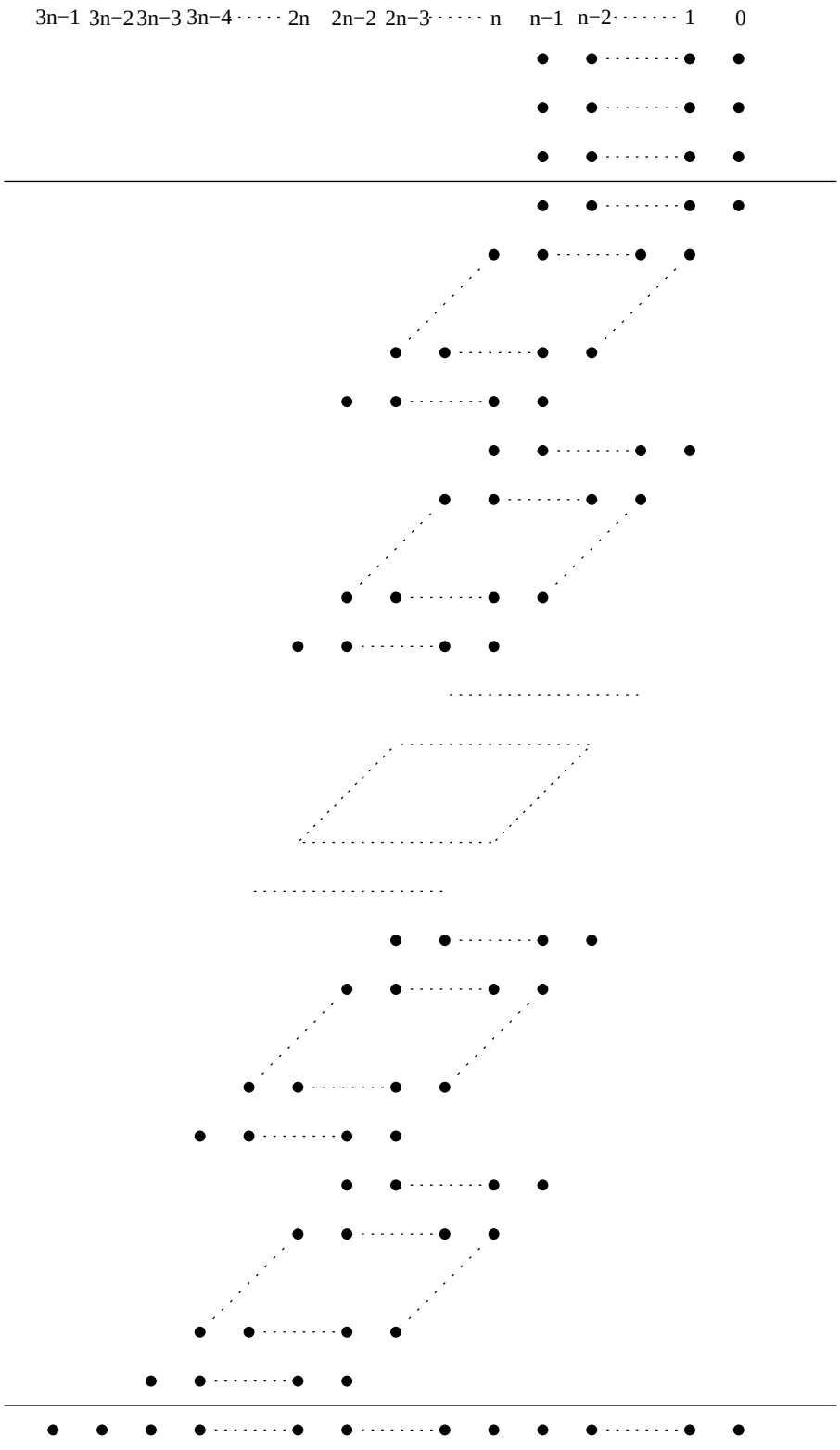

Figure 5.1: Partial Product Matrix of Cubing Unit

5.1 Liddicoat and Flynn Cubing Units

To overcome the restriction on the size of the PPM, [67] proposed a method, in which cubing unit can be implemented by following steps:

1. Step 1: Minimized PPM:

   (a) All terms $x_i \cdot x_i \cdot x_i$ in the row $(3i)^{th}$ are reduced to x_i and kept in the same row.

   (b) All three terms $x_i \cdot x_i \cdot x_j, x_i \cdot x_j \cdot x_i, x_j \cdot x_i \cdot x_i$ in the row $(2i + j)^{th}$ are reduced to one bit $x_i \cdot x_j$ and kept in the same row. Hence, the weight of these bits is 3.

   (c) All six terms $x_i \cdot x_j \cdot x_k$ in the row $(i + j + k)^{th}$ are reduced to one bit and shifted to the next row $(i + j + k + 1)^{th}$ on the left hand side, so the weight of these terms now is 3.

   Figure 5.2 is an illustration for Step 1 with 4-bit input.

2. Step 2: All the bits that have the weight of 3 are used to form an PPM. $Wallace$ tree [4] is applied to get carry and sum arrays. Figure 5.3 is an illustration of Step 2.

3. Step 3: The carry and sum arrays in the previous step are doubled and shifted on position to the left to form $2x$ in weight. These bits are combined with $2m$-bit x_i in the Step 1 and then using carry free $(5, 5, 4)$ counters to reduce the height of this matrix to get carry and sum arrays.

4. Step 3 CPA is utilized for carry and sum arrays to get the final result.

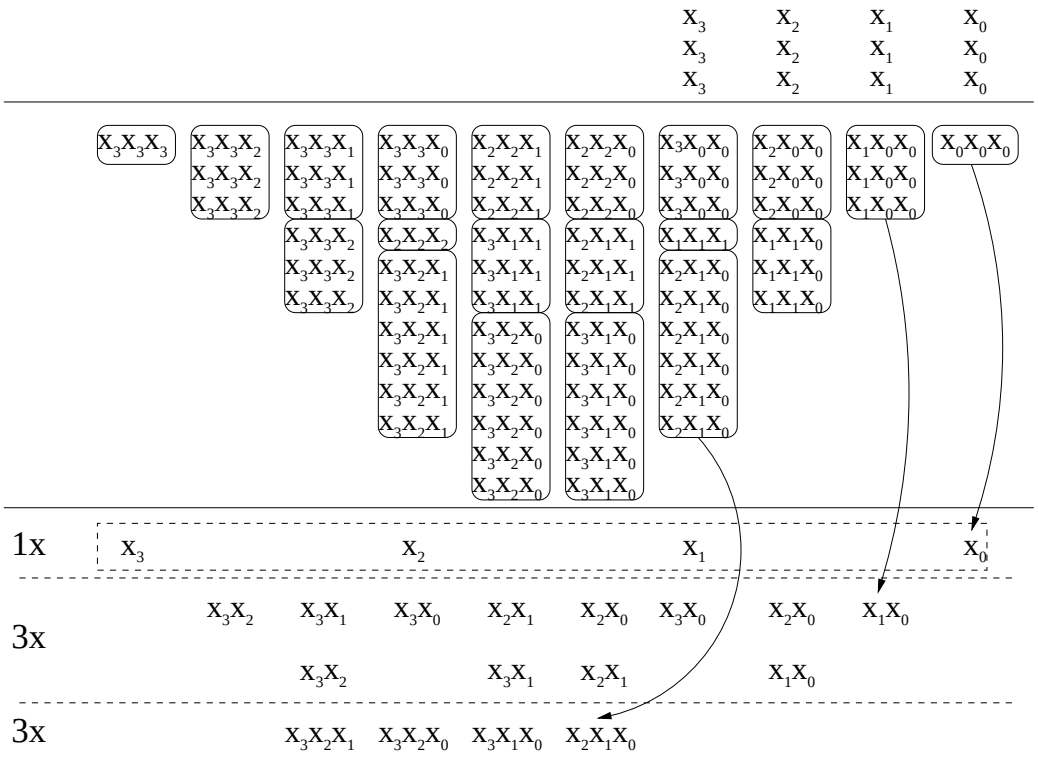

Figure 5.2: Liddicoat and Flynn Method for Step 1

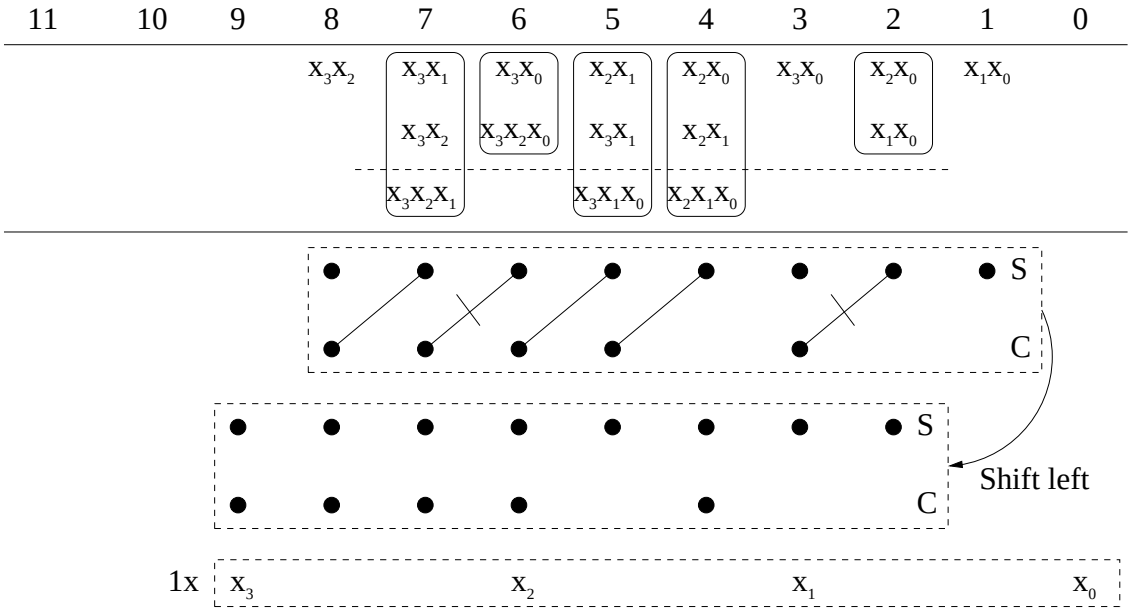

Figure 5.3: Liddicoat and Flynn Method for Step 2

5.2 Stine and Blank Cubing Units

Another method presented in [25], also focused on minimizing PPM by utilizing more regularity of PPM to reduce its height further. Following are steps of this method:

1. Step 1: Minimized PPM.

   (a) All terms $x_i \cdot x_i \cdot x_i$ in the row $(3i)^{th}$ are reduced to x_i and kept in the same row.

   (b) All three term $x_i \cdot x_i \cdot x_j, x_i \cdot x_j \cdot x_i, x_j \cdot x_i \cdot x_i$ in the row $(2i+j)^{th}$ are reduced as $x_i \cdot x_i \cdot x_j + x_i \cdot x_j \cdot x_i + x_j \cdot x_i \cdot x_i = 3x_i \cdot x_j = 2x_i \cdot x_j + x_i \cdot x_j$ so they are replaced by two bits $x_i x_j$, one in the same row $(2i+j)^{th}$ and the other in next row $(2i+j+1)^{th}$ on the left hand side.

   (c) All six terms in the row $(i+j+k)^{th}$ are calculated as $6 \cdot x_i \cdot x_j \cdot x_k = 2 \cdot (2 \cdot x_i \cdot x_j \cdot x_k + x_i \cdot x_j \cdot x_k)$, so they are also replaced by two bits $x_i \cdot x_j \cdot x_k$, one in the next row $(i+j+k+1)^{th}$, and the other is in next two row $(i+j+k+2)^{th}$ on the left hand side.

   Figure 5.4 is an implementation of a cubing unit for this method with 4-bit input for this step.

b) Step 2: Improving regularity.

   As can be seen in the previous step, three term $x_i \cdot x_{i-1} \cdot x_{i-1}$ are replaced by two bits $x_i \cdot x_{i-1}$, one is in the row $(3i-2)^{th}$ and the other is in the row $(3i-1)^{th}$. However, three terms $x_i \cdot x_i \cdot x_{i-1}$ are also replaced by two bits $x_i \cdot x_{i-1}$, one is in the row $(3i-1)^{th}$ and the other is in the row $(3i)^{th}$. So three bits $x_i \cdot x_{i-1}$ with two of them are in the row $(3i-1)^{th}$ and one is in the row $(3i)^{th}$ are reduced by equation $x_i \cdot x_{i-1} \cdot 2^{3i} + x_i \cdot x_{i-1} \cdot 2^{3i-1} + x_i \cdot x_{i-1} \cdot 2^{3i-1} = x_i \cdot x_{i-1} \cdot 2^{3i+1}$. In other word, it means that these bits are replaced by only one bit $x_i \cdot x_{i-1}$ in the row $(3i+1)^{th}$. An implementation for this step is shown in Figure 5.5

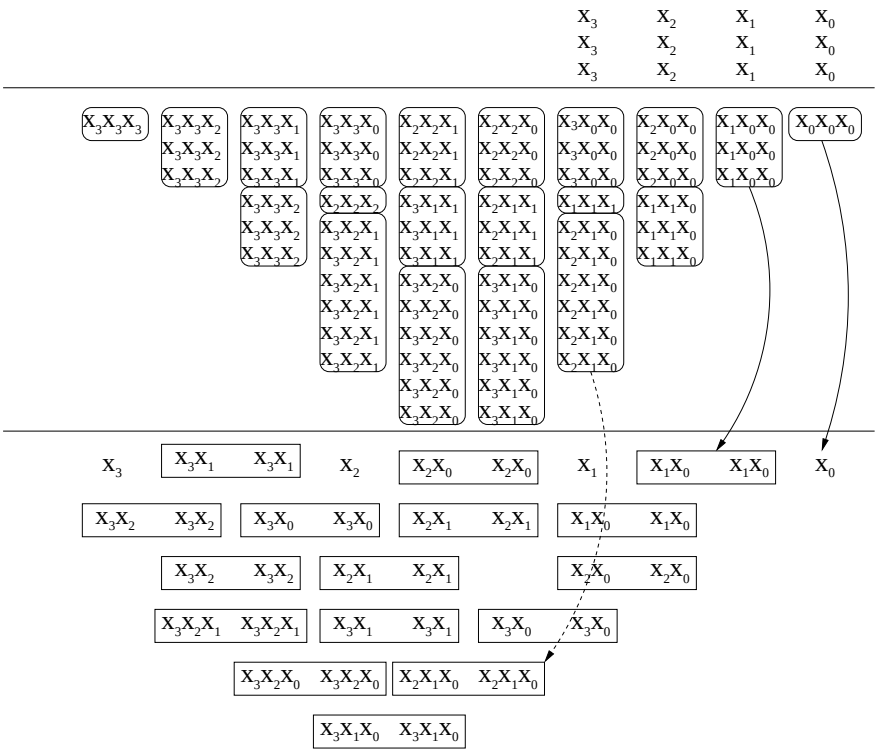

Figure 5.4: Stine and Blank Method Stage 1

c) Step 3: Calculate the final result.

   In this step, *Dadda* tree [5] is applied to reduce the height of this matrix, carry and sum arrays are generated.

d) Step 4: In the last stage a CPA is used to get the final result.

5.3 Divide and Conquer Cubing Units

Recently, in [75], another method was introduced by using $Vedic$ mathematics. $Vedic$ mathematics is a method that ancient civilizations utilized to help with computing large numbers. The implementation given in [75] is interesting is it tried to apply this method, but unfortunately these methods sometimes do not apply well to hardware. On the other hand, the method is interesting in that it attempts to compute the cube of an operation by breaking it down into simple and smaller cubing elements, hence

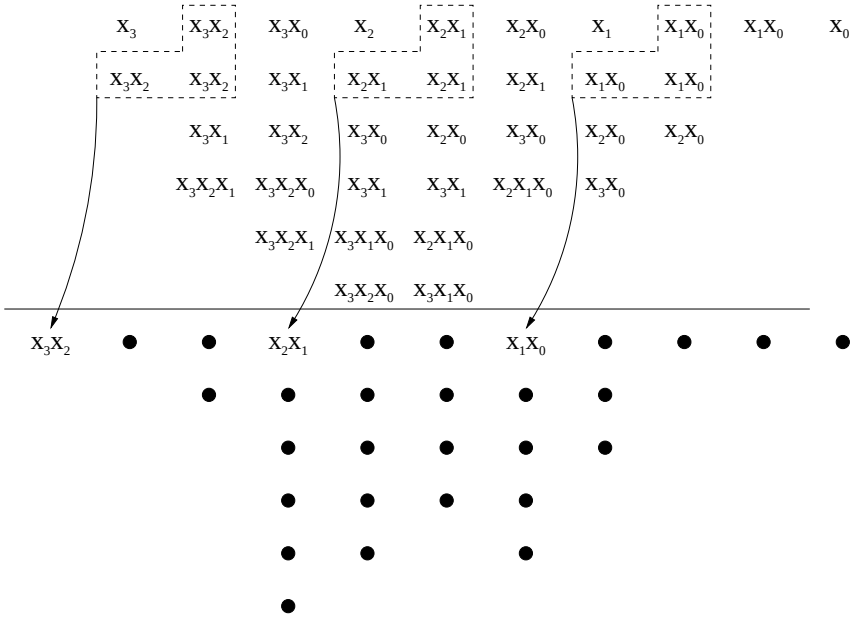

Figure 5.5: Stine and Blank Method Stage 2

it is called the *Divide & Conquer* technique. Following are steps to implement this unit:

1. Step 1: The input operand X is divided into two parts $A = x_{2m-1} \dots x_m$ and $B = x_{m-1} \dots x_0$, each part contains m-bit, and the final result P can be represented as

$$\begin{aligned} X &= A \cdot 2^m + B \\ P &= X^3 \\ &= (A \cdot 2^m + B)^3 \\ &= A^3 \cdot 2^{3m} + 3 \cdot A^2 \cdot B \cdot 2^{2m} + 3 \cdot A \cdot B^2 \cdot 2^m + B^3 \end{aligned} \tag{5.3}$$

Because each operand A and B has m-bit length, hence each term A^3, $A^2 \cdot B$, $A \cdot B^2$, B^3 will have $3m$-bit length. The $3\times$ terms $3 \cdot A^2 \cdot B$ and $3 \cdot A \cdot B^2$ can be expressed as $3 \cdot A^2 \cdot B = 2 \cdot A^2 \cdot B + A^2 \cdot B, 3 \cdot A \cdot B^2 = 2 \cdot A \cdot B^2 + A \cdot B^2$. Therefore, when all the value of A^3, $A^2 \cdot B$, $A \cdot B^2$, B^3 are known, PPM of this cubing unit is established and the value of P can be computed. This means

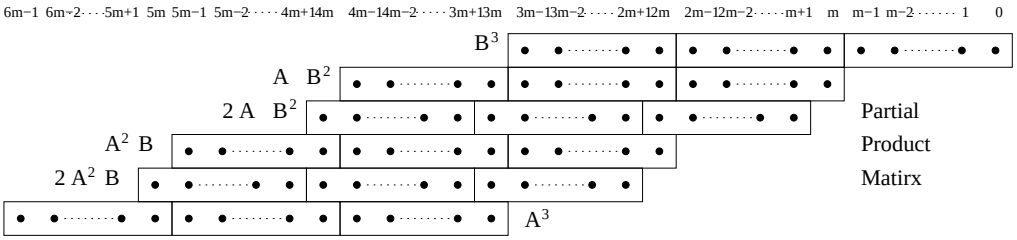

Figure 5.6: PPM of a Cubing Unit in Divide and Conquer Technique

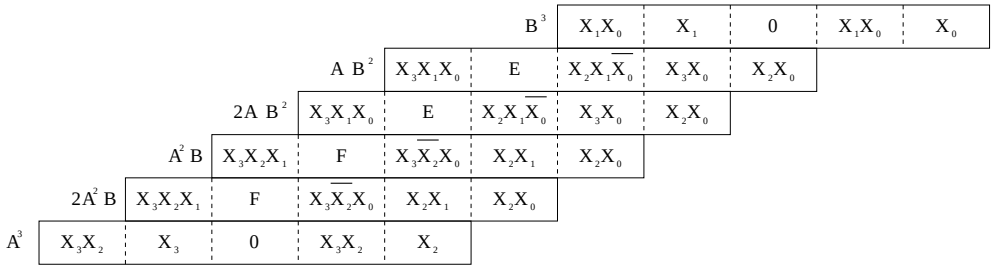

Figure 5.7: PPM of 4-bit Cubing Unit in Divide and Conquer Technique

that this method can only be applied to input operands that are power of two and also each cubing operation is completed in repeated serial steps. Figure 5.6 is PPM for a cubing unit of $2m$-bit input.

2. Step 2: Repeat Step 1 until $m = 4$.

3. Step 3: Implement a cubing unit of 4-bit input. $X = x_3x_2x_1x_0, A = x_3x_2, B = x_1x_0$. All terms $A^3, B^3, A^2 \cdot B, A \cdot B^2$ will have only 5-bit length and have been expressed as following, in which $E = x_3 \cdot x_1 \cdot \overline{x_0} + x_2 \cdot x_1 \cdot x_0$ and $F = x_3 \cdot x_2 \cdot x_0 + x_3 \cdot \overline{x_2} \cdot x_1$, partial product matrix of this unit can be generated as in Figure 5.7

As stated previously, the method proposed by [75] based on $Vedic$ mathematics, therefore it is rather slow as the input operand grows (as will be shown later). This is because this method has to compute the cube of $n = 4$ first, and then sequentially computes the cube of $n = 8$ and so on until it reaches its designed power of two operand size. Although this method is slow, the idea can be elaborate to combine all

three methods. The only difference is that, the intermediate steps within the method in [75] can be removed.

5.4 Proposed Cubing Unit.

In this section, a new way to implement a cubing unit will be presented [76]. The following are proposed steps

1. Step 1: Minimized PPM

   The $2m$-bit input operand X is divided into m parts:

   $$X_i = x_{2i+1}x_{2i} \tag{5.4}$$

   so the output of cubing unit can be represented as

   $$\begin{aligned} P &= X^3 \\ &= (\sum_{i=0}^{m-1} X_i \cdot 2^{2i})^3 \\ &= \sum_{i=0}^{m-1} X_i^3 \cdot 2^{6i} + 3 \cdot \sum_{i=0}^{m-2} \sum_{j=i+1}^{m-1} X_i^2 \cdot X_j \cdot 2^{2(2i+j)} \\ &\quad +3 \cdot \sum_{i=0}^{m-2} \sum_{j=i+1}^{m-1} X_i \cdot X_j^2 \cdot 2^{2(i+2j)} \\ &\quad +3 \cdot \sum_{i=0}^{m-3} \sum_{j=i+1}^{m-2} \sum_{k=j+1}^{m-1} X_i \cdot X_j \cdot X_k \cdot 2^{2(i+j+k)+1} \end{aligned} \tag{5.5}$$

   The PPM is formed for all $3x$ terms with the position of these terms depends on the weight of themselves. That is , the input operand is grouped into smaller elements, so that the equal terms (i.e.,weights) can be grouped together (e.g. $3 \times X$ in the equation above). For example, using an 8-bit input cubing unit, the input operand is grouped into following assuming a BNF format: $X[7:0] = X[7:6], X[5:4], X[3:2], X[1:0]$ or four sub-divisions.

2. Step 2: *Dadda* tree is applied for PPM reduction to get the carry and sum arrays. PPM for Step 1 and 2 is illustrated as in Figure 5.8

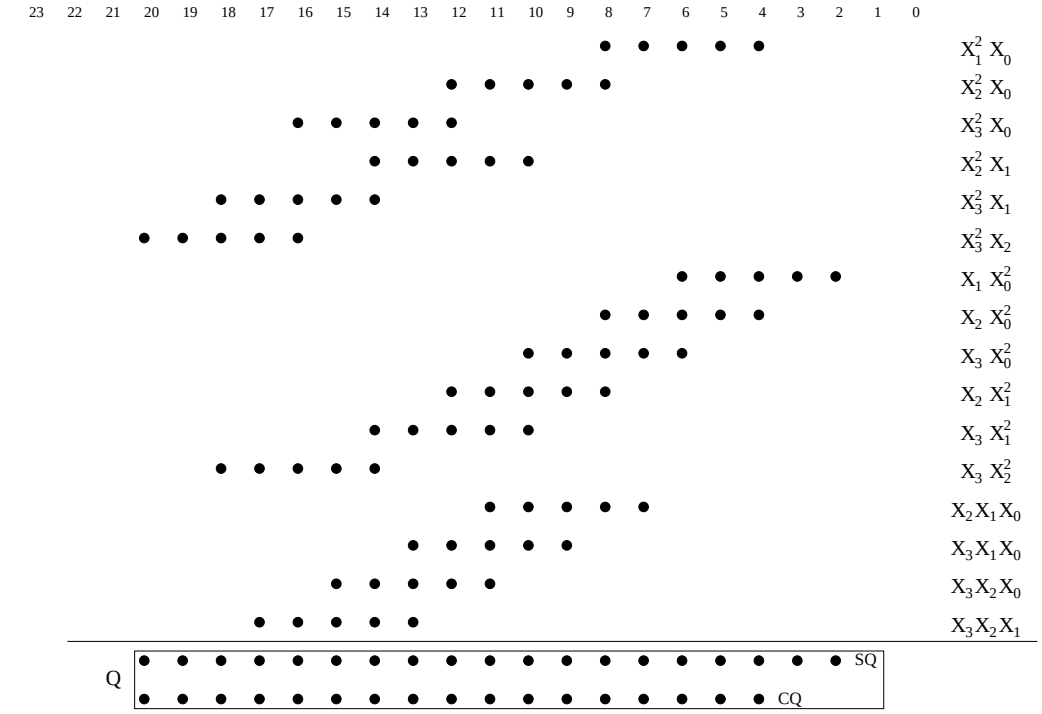

Figure 5.8: Partial Product Matrix for Proposed Cubing Unit at Step 1, 2

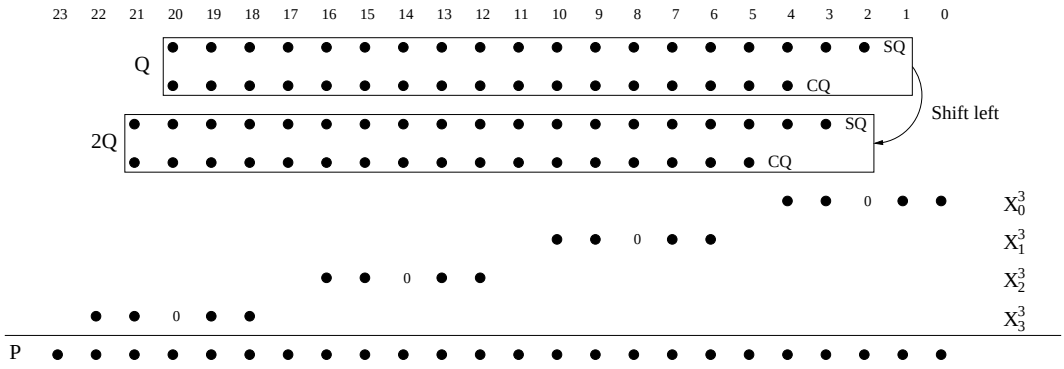

Figure 5.9: Partial Product Matrix for Proposed Cubing Unit at Step 3

3. Step 3: The carry and sum arrays in the previous step are doubled and shifted one position to the left hand side to form the $2x$ terms. These bits combined with X_i^3 to calculate the final result. This step is illustrated as in Figure 5.9

In this example, the smaller elements are just shifted to the left, depending on where the sub-division is. Consequently, and shown from equation above, these terms get accumulated in to a value which has prefix of 3. This mean they can be reduces

into one term and multiplied by the value of 3. The method in [67] has a similar 3 prefix, but in this method, the cubing unit is just reduced into smaller cubing units slightly similar to the method in [75] except that the proposed method is formed to aggregate all the columns via the Equation 5.5.

In summary, the new PPM is composed of two major elements: elements with prefix of 3 and the original X_i^3 elements. The idea in terms of interval is similar to [75] in term of breaking the operand into sub-intervals, but the different is that the new sub-intervals are utilized to form the PPM in power of two and then reduced and combined with the X_i^3. Then, the power of powers of two sub-intervals are easy to form into an array (e.g.$(11)^3 = 11011$). If Q is defined as

$$\begin{aligned} Q &= \sum_{i=0}^{m-2}\sum_{j=i+1}^{m-1} X_i^2 \cdot X_j \cdot 2^{2(2i+j)} \\ &\quad + \sum_{i=0}^{m-2}\sum_{j=i+1}^{m-1} X_i \cdot X_j^2 \cdot 2^{2(i+2j)} \\ &\quad + \sum_{i=0}^{m-3}\sum_{j=i+1}^{m-2}\sum_{k=j+1}^{m-1} X_i \cdot X_j \cdot X_k \cdot 2^{2(i+j+k)+1} \end{aligned} \tag{5.6}$$

This breaks the output P down into the following equation

$$\begin{aligned} P &= 3 \cdot Q + (\sum_{i=0}^{m-1} X_i \cdot 2^{2i})^3 \\ &= 2 \cdot Q + Q + (\sum_{i=0}^{m-1} X_i \cdot 2^{2i})^3 \end{aligned} \tag{5.7}$$

The two elements Q and X_i^3 are shown at the bottom of the Figure 5.9. Conversely, in [75] the grouping is done to form a smaller cube(e.g., $m = 4$) and then utilized to create a lager cuber after the final cube operation is formed. That is, the result for $n = 4$ is used to create $n = 8$ and onward, which can be in efficient in hardware. On the other hand, the method in [75] may be helpful for software routines. Moreover, the proposed method works provided the operand size in even, whereas, the method in [75] only works for power of two operand size (2, 4, 8, 16, 32, 64, ...).

CHAPTER 6

Truncated Squaring and Cubing Units

6.1 Truncated Squaring Units

This section will show how to implement a truncated squaring unit presented in [11], [74], [71] and [77] with CCT technique in Chapter 3. A constant C is calculated and PPM of each unit is established. *Dadda* tree is applied to get carry and sum arrays. After that, a fast adder CPA is utilized to get final result.

6.1.1 Truncated Squaring Units using Booth 2 Folding Technique

For Booth 2 folding method, truncated PPM of a squaring unit can be illustrated as in Figure 6.1. As explained in Chapter 3, there two types of errors for truncated

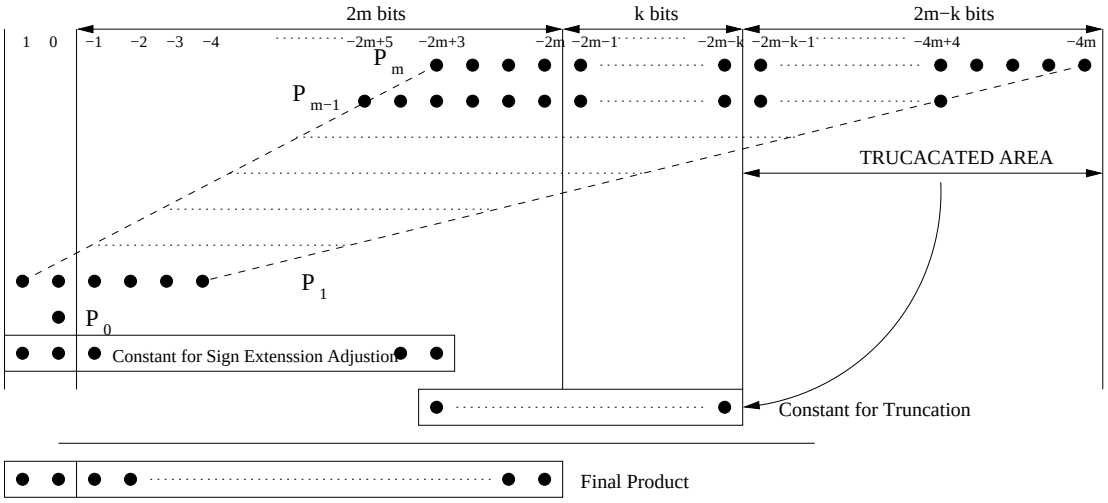

Figure 6.1: PPM of Truncated Squaring Unit in [56]

squaring units, reduction and rounding, that are expressed as

$$\begin{aligned} E_{reduct} &= \sum_{i=0}^{j}(E[P_{i-trunc}]) \\ E_{round} &= \frac{1}{2}\sum_{i=1}^{k} 2^{-2m-i} \end{aligned} \tag{6.1}$$

in which $P_{i-trunc}$ is partial product row i^{th} in truncated area and $E(P_{i-trunc})$ is its expectation. The number of row j that are truncated is depend of value of number of column to be kept k. Because P_i runs from $lsb = -4i$ to $msb = -2i + 3$, hence, if $-4i \geq -2m-k$, P_i is not truncated, otherwise a part of P_i is truncated. It means that, when i run from $a = \left\lceil \frac{2m+k}{4} \right\rceil$ to m, P_i will be truncated. According to Equation 4.13, the expectation of these bits in this row are calculated as

$$\begin{cases} E(P_{i,-4i}) &= \frac{1}{2} \\ E(P_{i,-4i+1}) &= 0 \\ E(P_{i,-4i+2}) &= \frac{1}{4} \\ E(P_{i,-4i+3}) &= \frac{1}{4} \\ E(P_{i,-4i+4}) &= \frac{3}{8} \\ \vdots & \\ E(P_{i,-2i}) &= \frac{3}{8} \\ E(P_{i,-2i+1}) &= \frac{3}{8} \\ E(P_{i,-2i+2}) &= \frac{3}{8} \\ E(P_{i,-2i+3}) &= \frac{1}{8} \end{cases} \tag{6.2}$$

The truncation of P_i and its expectation of each bit can be illustrated as in Figure 6.2 As can be seen in this figure, P_i is truncated when $i \geq a$, the value of this part can be estimated as

$$\begin{aligned} E(P_{i-trunc}) &= \frac{3}{8}\sum_{j=-2m-k-1}^{-4i+4} 2^j + \frac{1}{4}2^{-4i+3} + \frac{1}{4}2^{-4i+2}\frac{1}{2}2^{-4i} \\ &= 3 \times 2^{-2m-k-3} - \frac{5}{2} \times 2^{-4i} \end{aligned} \tag{6.3}$$

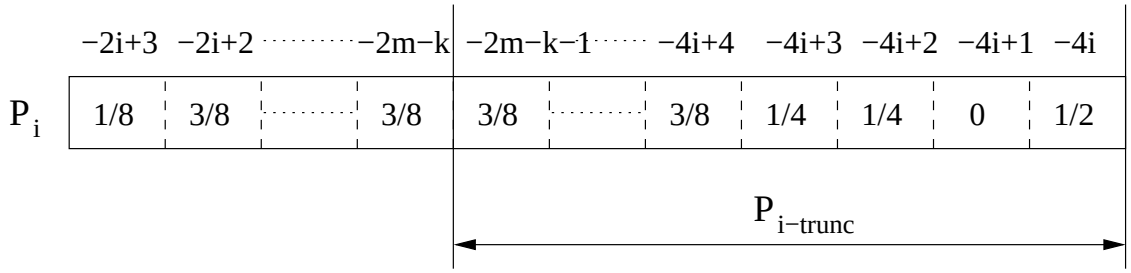

Figure 6.2: Truncation of P_i in [56]

.

The truncation error can be estimated as

$$\begin{aligned} E_{reduct} &= \sum_{i=a}^{m}(E[P_{i-trunc}]) \\ &= \sum_{i=a}^{m} 3 \times 2^{-2m-k-3} - 5 \times 2^{-4i-1} \\ &= 3(m-a)2^{-2m-k-3} - \frac{1}{6}2^{-4a+4} + \frac{1}{6}2^{-4m} \end{aligned} \tag{6.4}$$

The rounding error can be calculated as

$$\begin{aligned} E_{round} &= \frac{1}{2}\sum_{i=1}^{k} 2^{-2m-i} \\ &= (2^{-2m-1} - 2^{-2m-k-1}) \end{aligned} \tag{6.5}$$

The constant correction C can be calculated by replace Equation 6.4 and 6.5 to 3.2.

6.1.2 Truncated Squaring Unit using Booth 2 Left-to-Right Encoding Technique.

The partial product matrix of a truncated squaring unit with this method can be illustrated as in Figure 6.3. The error of reduction is calculated as same as Equation 6.5. The error of truncation depends on the expectation of bits of P_i in the

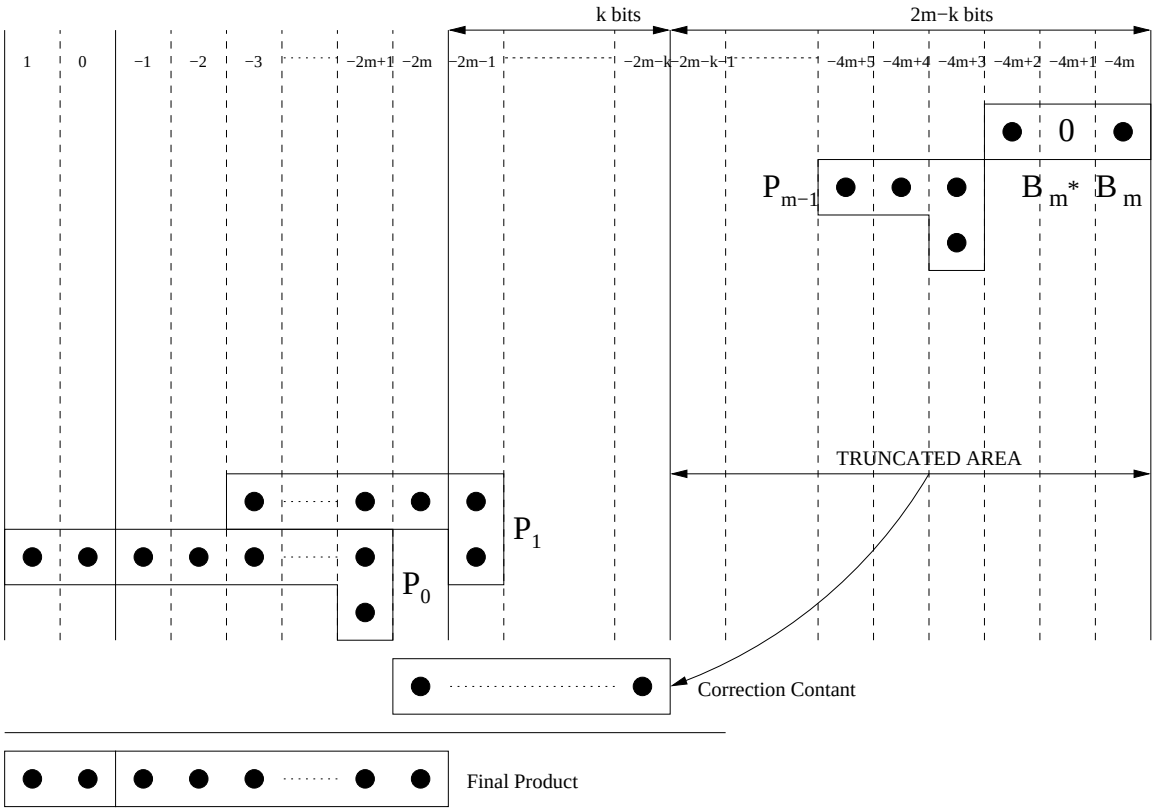

Figure 6.3: Partial Product Matrix of Truncated Squaring Unit In [74]

Equation 4.20, that are calculated as

$$\begin{cases} E(P_{i,-2m-2i+1}[0]) &= \frac{3}{8} \\ E(P_{i,-2m-2i+1}[1]) &= \frac{3}{8} \\ E(P_{i,-2m-2i+2}[0]) &= \frac{3}{8} \\ \vdots & \\ E(P_{i,-4i}[0]) &= \frac{3}{8} \\ E(P_{i,-4i+1}[0]) &= \frac{1}{8} \end{cases} \tag{6.6}$$

The truncation of P_i can be illustrated as in Figure 6.4. As can be seen in this figure, when $-2m-2i+1 \geq -2m-k$, P_i will not be in the truncated area. When $-4i+1 \leq -2m-k-1$, all bits of P_i will be in truncated area. Otherwise, part of P_i will be in truncated area. Define $a = \lfloor \frac{k+1}{4} \rfloor$, and $b = \lfloor \frac{2m+k+2}{4} \rfloor$, the estimation of

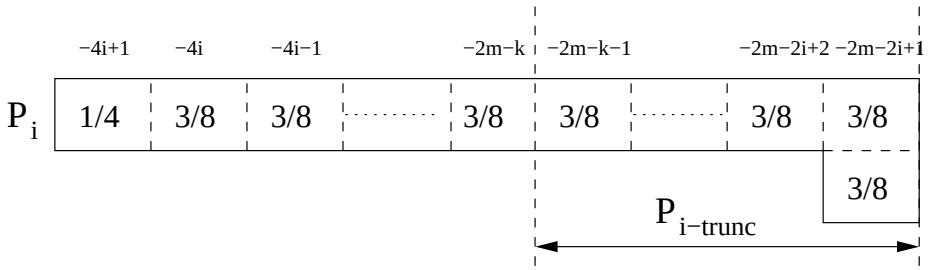

Figure 6.4: Truncation of P_i in Left-to-Right Encoding

P_i and $P_{i-trunc}$ will be computed as

$$\begin{aligned} E(P_i) &= \frac{1}{4}2^{-4i+1} + \frac{3}{8}\sum_{j=-4i}^{-2m-2i+1} 2^j + \frac{3}{8}2^{-2m-2i+1} \\ &= \frac{5}{8}2^{-4i+1} \\ E(P_{i-trunc}) &= \frac{3}{8}\sum_{j=-2m-k-1}^{-2m-2i+1} 2^j \frac{3}{8}2^{-2m-2i+1} \\ &= \frac{3}{8}2^{-2m-k} \end{aligned} \tag{6.7}$$

So truncation error can be calculated as

$$\begin{aligned} E_{reduct} &= -\sum_{i=a+1}^{b} E(P_{i-trunc}) - \sum_{b+1}^{m-1} E(P_i) - E(B_m^2) \\ &= -\frac{3(b-a-1)}{8}2^{-2m-k} - (\frac{1}{12}(2^{-4b} - 2^{-4m+4})) - \frac{3}{2}2^{-4m} \end{aligned} \tag{6.8}$$

The rounding error can be calculated as Equation 6.5, hence, the correction constant C can be calculated by replace Equation 6.5 and 6.8 into 3.2.

6.1.3 Truncated Squaring Unit using Divide and Conquer Technique.

The partial product matrix of a truncated squaring unit using Divide and Conquer technique can be illustrated as in Figure 6.5. Rounding error can be expressed as

$$E_{round} = -\frac{1}{2}\sum_{i=2m-k}^{2m-1} 2^{-4m+i} \tag{6.9}$$

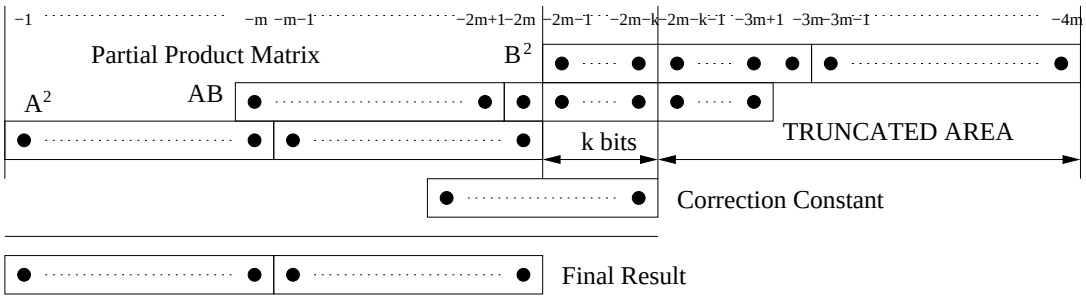

Figure 6.5: PPM of Truncated Squaring Unit in [71]

The truncation error is hard to calculate but we can see that, the expectation of each bit in truncated is grater than zero, so the constant correction C will be

$$\frac{round(2^{2m+k} \times E_{round})}{2^{2m+k}} \leq C \tag{6.10}$$

Replace Equation 6.9 to 6.10

$$2^{-2m-1} \leq C \tag{6.11}$$

As can be seen in the Figure 6.5, the expectation of each bit in the truncated area is smaller than $\frac{1}{2}$ so truncation error will be smaller than $E_{trunc-max}$, in which

$$E_{trunc-max} = \frac{1}{2}(\sum_{i=0}^{2m-k-1} 2^{-4m+i} + \sum_{i=m+1}^{2m-k-1} 2^{-4m+i}) \tag{6.12}$$

So the correction constant C is bounded by

$$C \leq \frac{round(2^{2m+k}(E_{round} + E_{trunc-max}))}{2^{2m+k}} \tag{6.13}$$

By replacing Equation 6.9, 6.12 into 6.13, ones can have

$$C \leq 2^{-2m-1} \tag{6.14}$$

Equation 6.11 and 6.14 ensure that $C = 2^{-2m-1}$.

6.1.4 Proposed Truncated Squaring Unit

The proposed truncated squaring unit is implemented by applying CCT method to squaring unit in [77]. PPM of this unit can be view as in Figure 6.6. The number

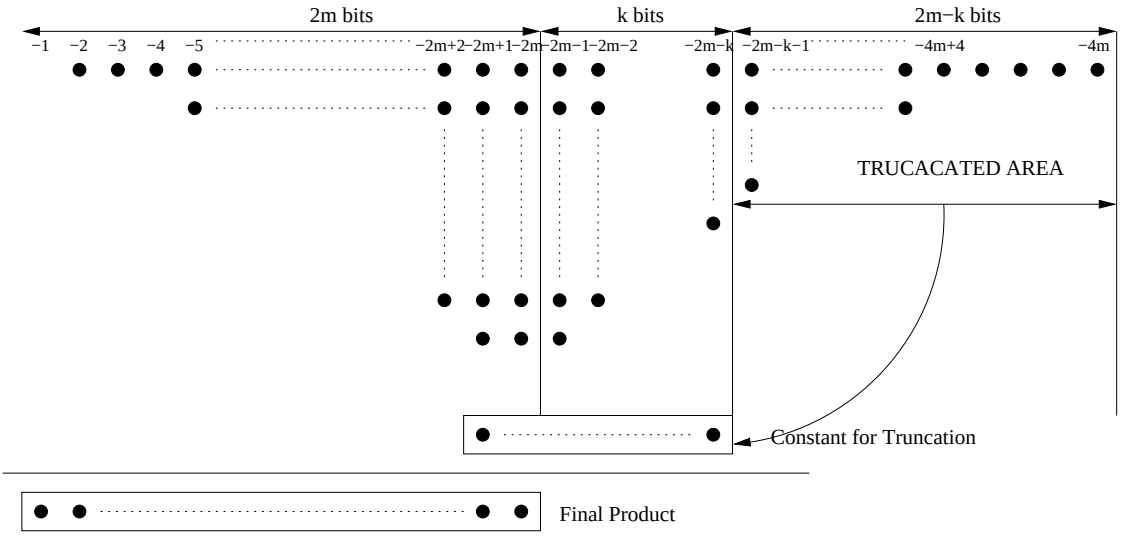

Figure 6.6: PPM of Proposed Truncated Squaring Unit

of partial product bits in the column $(-2m-k-1)^{th}$ is $\lfloor \frac{2m-k-1}{2} \rfloor$, in which one bit come from either Stage 2 or Stage 3, the others are in form $x_i \cdot x_j$. According to Equation 4.5, 4.6, the expectation of this bit is $\frac{3}{8}$, hence, the expectation of the column $(-2m-k-1)^{th}$ is:

$$\begin{aligned} E_{(-2m-k-1)^{th}} &= (\frac{3}{8} + \frac{1}{4}(\left\lfloor \frac{2m-k-1}{2} \right\rfloor - 1)) \cdot 2^{-2m-k-1} \\ &= (\left\lfloor \frac{2m-k-1}{2} \right\rfloor + \frac{1}{2}) \cdot 2^{-2m-k-3} \end{aligned} \tag{6.15}$$

So the reduction error caused by truncation operation is calculated as

$$\begin{aligned} E_{reduct} &= \sum_{j=5}^{2m-k-1} E_{i^{th}} + 41 \cdot 2^{-4m-2} \\ &= \sum_{j=5}^{2m-k-1} (\left\lfloor \frac{j}{2} \right\rfloor + \frac{1}{2}) \cdot 2^{-4m+j} + 41 \cdot 2^{-4m-2} \end{aligned} \tag{6.16}$$

The rounding error is calculated as Equation 6.5. Replace Equation 6.5, 6.16 into 3.2, correction constant C will be calculated.

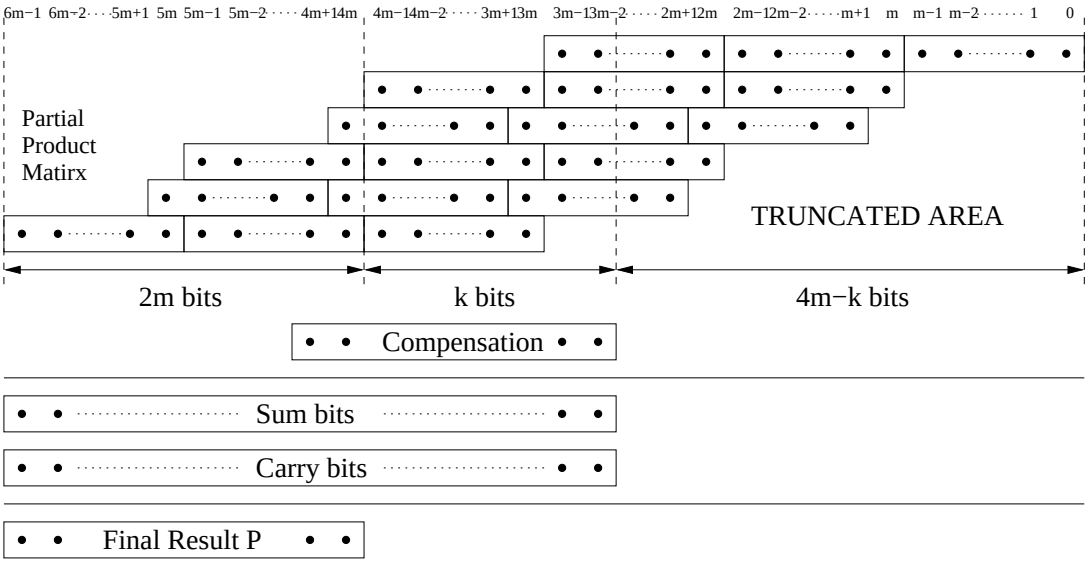

Figure 6.7: PPM of Truncated Cubing Unit in [75] .

6.2 Truncated Cubing Units

6.2.1 Truncated Cubing Unit using Divide and Conquer Technique.

Figure 6.7 illustrates the PPM of a truncated cubing unit base on [75] method, in which $(4m-k)$ less significant columns are truncated. Because each bit in this area has probability of being one is less than $\frac{1}{2}$, hence, the truncated error can calculated as

$$E_{reduct} \leq -\frac{1}{2}(\sum_{i=0}^{2m-1} 2^{-6m+i} + \sum_{i=m}^{2m-1} 2^{-6m+i} + \sum_{i=m+1}^{2m-1} 2^{-6m+i}) \tag{6.17}$$

As we can see in the Figure 6.7, all bits from column $(4m-k)^{th}$ to $(4m-1)^{th}$ cause rounding error. Because the probability of these bits being one is also less than $\frac{1}{2}$, rounding error can computed as

$$E_{round} = -\frac{1}{2}(\sum_{i=2m}^{4m-1} 2^{-6m+i}) \tag{6.18}$$

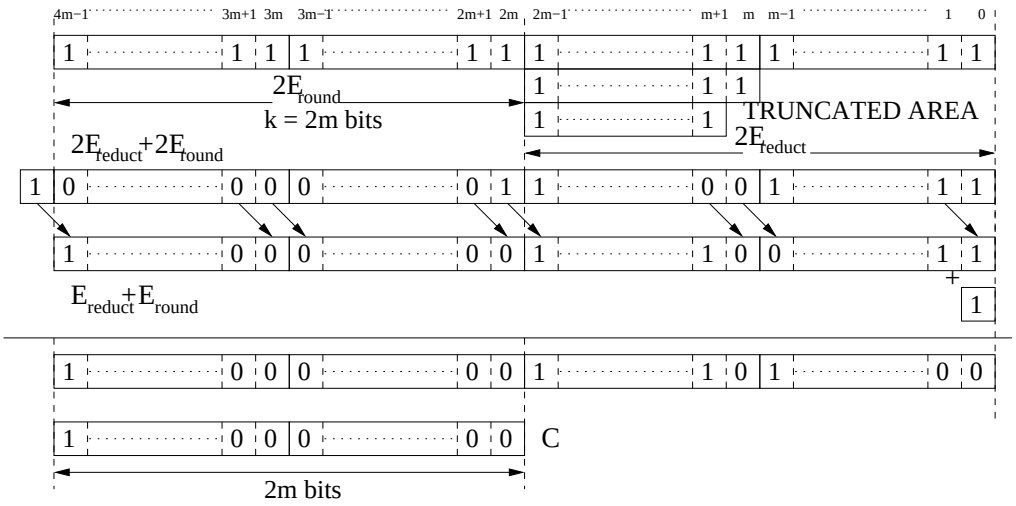

Figure 6.8: Correction Constant for Truncated Cubing Unit in [75] when $k = 2m$.

Hence, the total error is the sum of reduction and rounding error is bounded by

$$\begin{aligned}
E_{total} &= E_{reduct} + E_{round} \\
&\leq \frac{1}{2}\left(\sum_{i=0}^{2m-1} 2^{-6m+i} + \sum_{i=m}^{2m-1} 2^{-6m+i} + \sum_{i=m+1}^{2m-1} 2^{-6m+i}\right) + \frac{1}{2}\left(\sum_{i=2m}^{4m-1} 2^{-6m+i}\right) \\
E_{total} &\leq 2^{-4m-1} + 2^{-5m-1} - 2^{-6m} + 2^{-2m-1} - 2^{-4m-1} \\
E_{total} &\leq 2^{-2m-1} + 2^{-5m-1} - 2^{-6m} \qquad (6.19)
\end{aligned}$$

The constant C is obtained by rounding E_{total} to $r + k$ fractional bits such that

$$C = \frac{round(2^{2m+k} \times E_{total})}{2^{2m+k}} \qquad (6.20)$$

Where $round(x)$ indicates x is rounded to nearest integer. Replace Equation 6.19 into 6.20, C can computed as

$$C \leq 2^{-(2m+1)} \qquad (6.21)$$

Figure 6.8 and 6.9 illustrate how to compute C when $k = 2m$ and $0 < k < m$, respectively. Figure 6.10 and 6.11 show *Dadda* tree reduction for truncated cubing units, in which $k = 2m$ and $0 < k < m$. As can bee seen, when $k = 2m$, total $m + 1$ HAs, $5m$ FAs, and $4m$-bit CPA are needed; when $0 < k < m$ total $m + 2$ HAs, $m + 3k + 1$ FAs, and $2m + k$-bit CPA are needed for implementing this unit. It

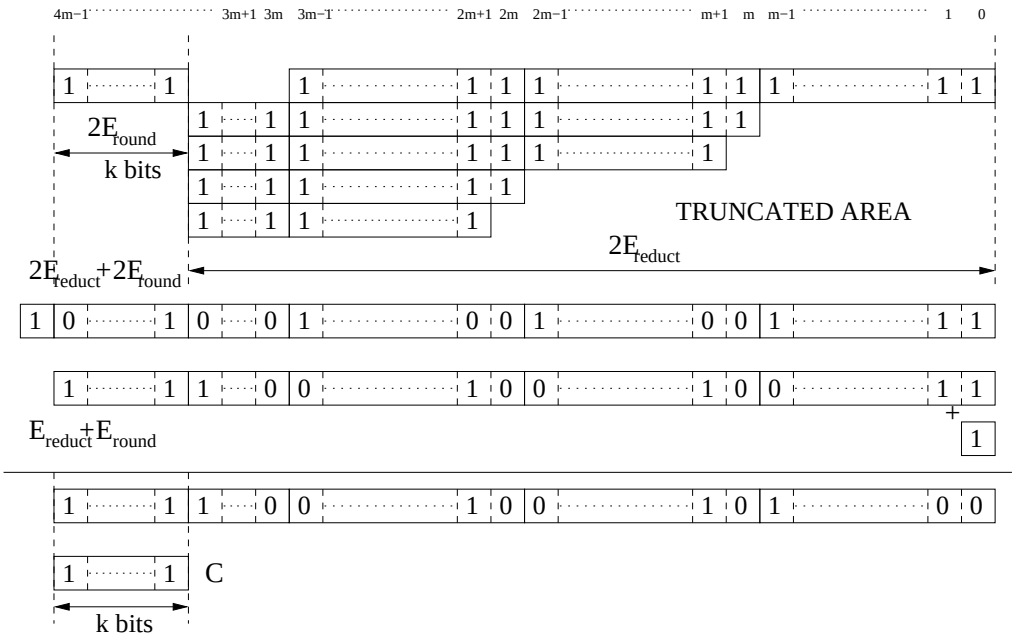

Figure 6.9: Correction Constant for Truncated Cubing Unit in [75] when $0 < k < m$.

means that when the parameter k is reduced, the area and delay of this unit can be reduced, but the error will be increased.

6.2.2 Proposed Truncated Cubing Unit

The proposed truncated cubing unit is base on technique presented in [76]. The same method CCT is applied for error compensation. Following are steps to implement this unit:

1. Step 1: Establish truncated PPM for all terms that have the weight of 3:

   Using PPM generated as in the Figure 5.8, truncated PPM of terms with the weight of 3 is established by omitting $(4m - k)$ columns in the lower part of that PPM, in which k is the number of columns to be kept.

2. Step 2: *Dadda* tree reduction

   Dadda tree is applied to get the carry and sum arrays. Figure 6.12 is an example of these steps for 8-bit truncated cubing unit with $k = 8$.

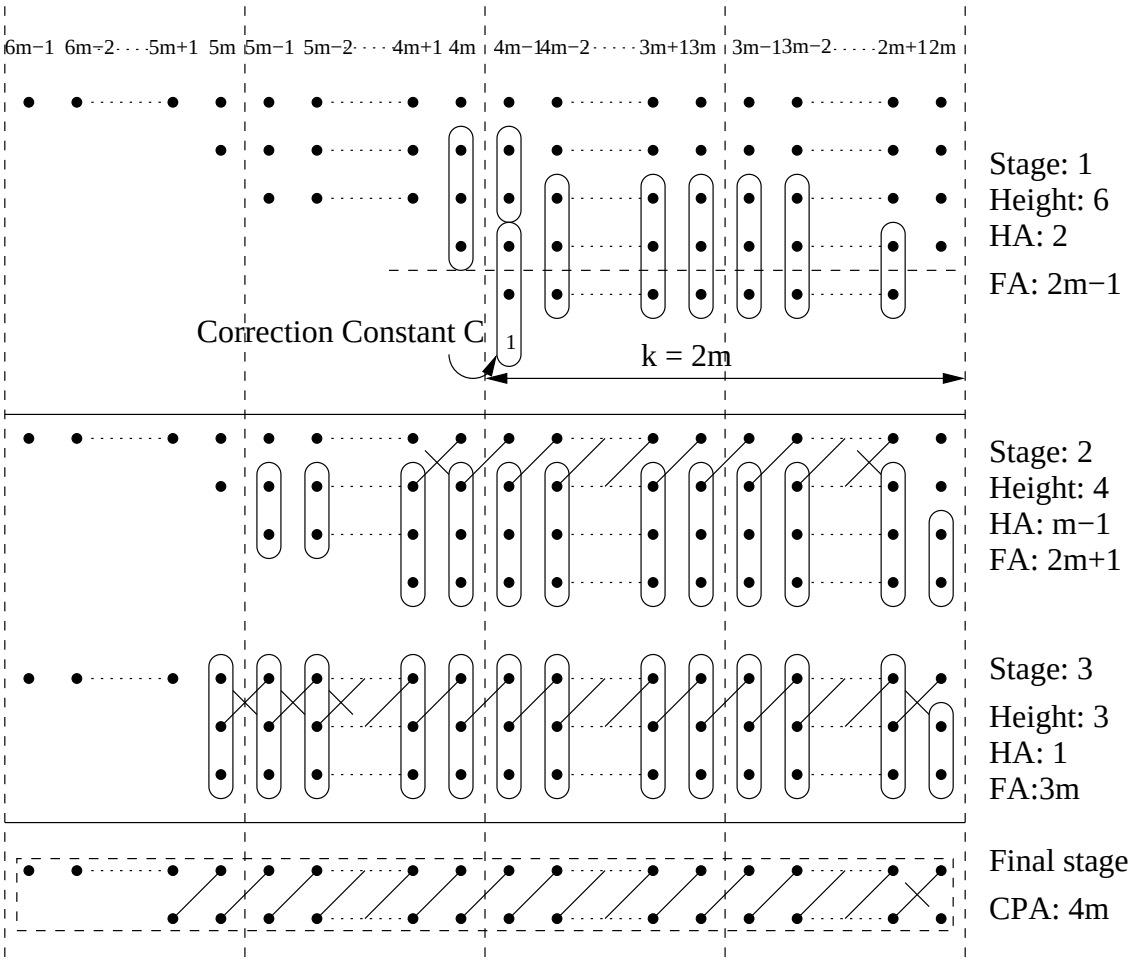

Figure 6.10: Dadda Tree of Truncated Cubing Unit in [75] when $k = 2m$.

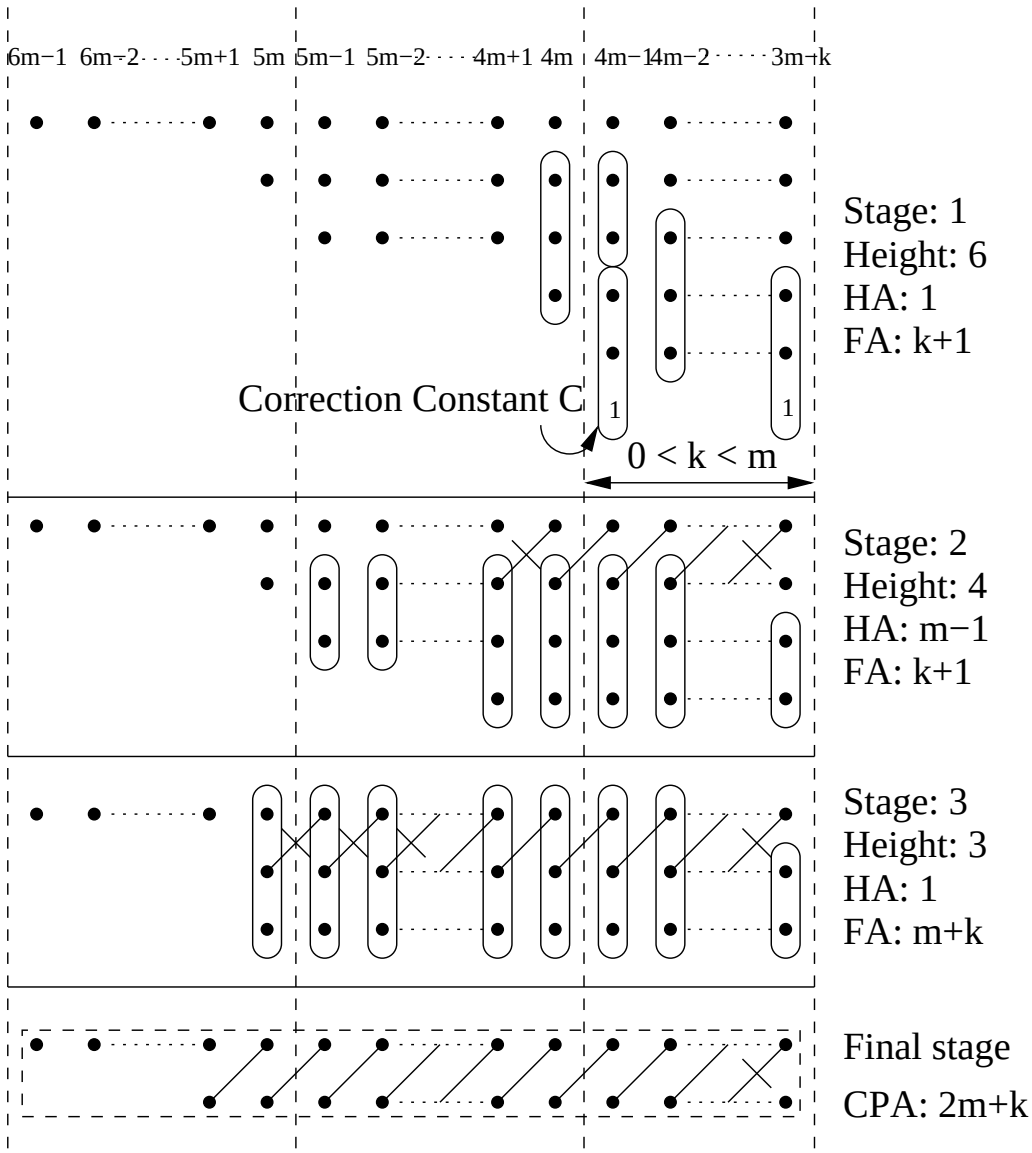

Figure 6.11: Dadda Tree of Truncated Cubing Unit in [75] when $0 < k < m$.

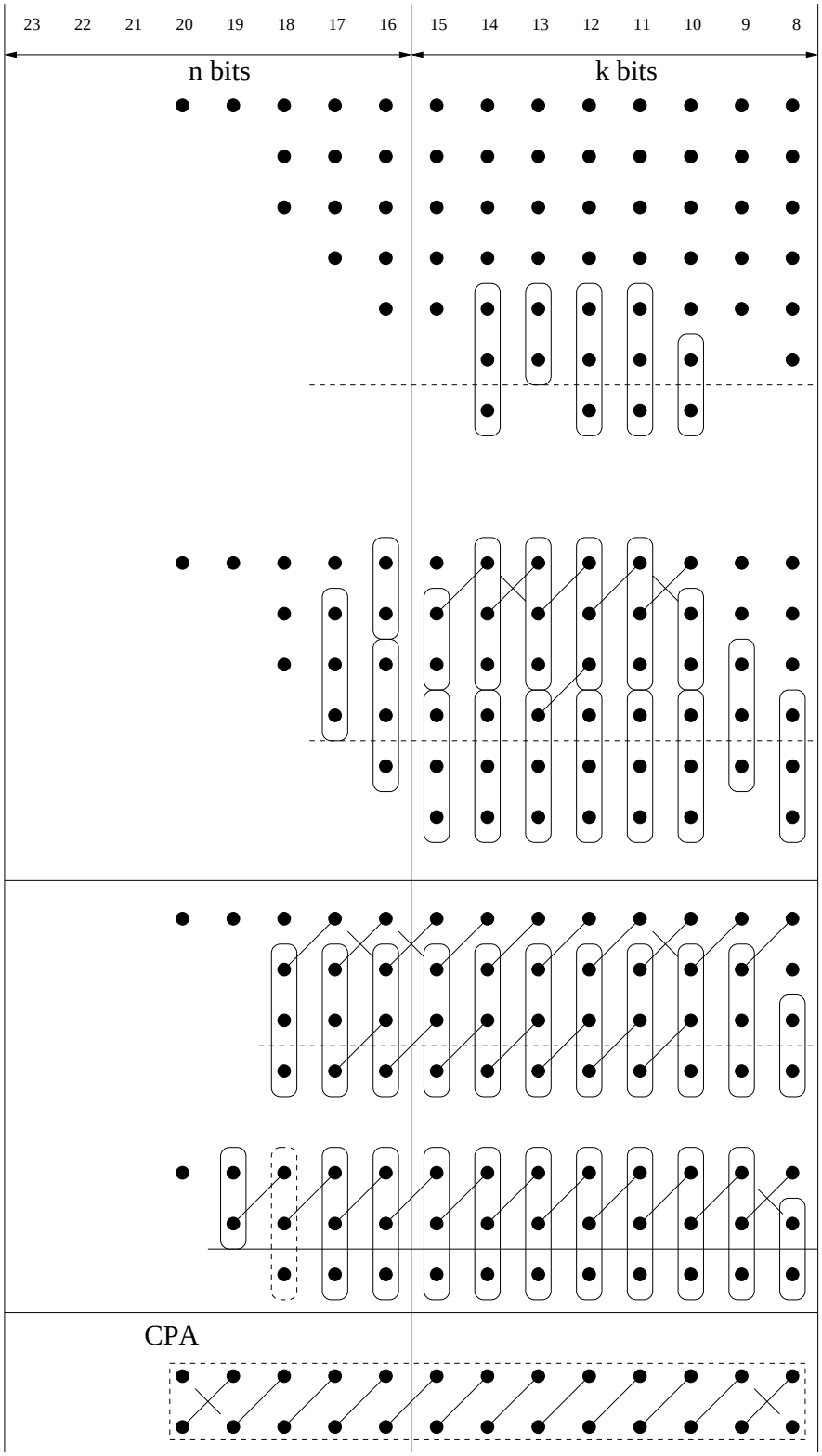

Figure 6.12: PPM for Truncated Proposed Cubing Unit at Step 1, 2

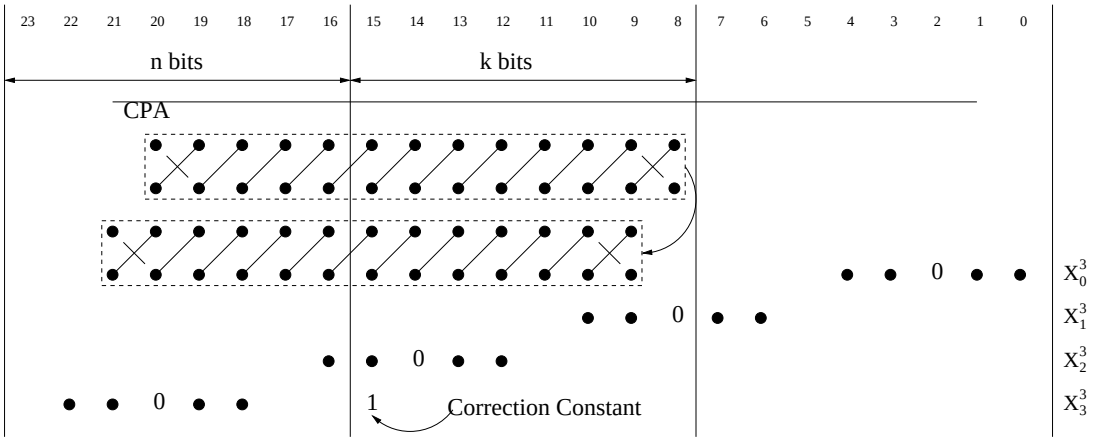

Figure 6.13: Partial Product Matrix for Proposed Cubing Unit at Step 3

3. Step 3: The carry and sum arrays in the previous step are doubled and shifted one position to the left hand side to form the $2x$ terms. These bits combined with X_i^3 , that are truncated too, and the correction constant C to calculate the final result. This step is illustrated as in Figure 6.13.

The correction constant C is estimated by calculating E_{reduct} and E_{round}. The rounding error E_{round} can be easily calculated as the same as in the Equation 6.18. The truncated error is hard to calculate because it depends on the probability of each bit in terms of $X_i^2 \cdot X_j$, $X_i \cdot X_j^2$, and $X_i \cdot X_j \cdot X_k$. First of all, truncated error for the PPM in the Figure 6.12 should be calculated. Then it is tripled and added with the rounding error E_{round} to establish the total error. As can be seen in the Figure 6.12, partial product term $X_i^2 \cdot X_j$ has 5-bit length, from $(4i + 2j)^{th}$ to $(4i + 2j + 4)^{th}$. Whenever $(4i + 2j) \leq (2n - k - 5)$ this term will be truncated in a whole. When $(2n - k - 4) \leq (4i + 2j) \leq (2n - k - 1)$ this term is truncated in a part. Otherwise, it is not in truncated area. The same thing is applied for the terms $X_i \cdot X_j^2$, and $X_i \cdot X_j \cdot X_k$. Hence, it is not practical for formulating the correction constant C, but we can get that value by running a C program in the section below.

Table 6.1: Errors Comparison for Truncated Squaring Units in [11]

n	k	C	E_{avg}	E_{sqr}	E_{max}
8	2	9.77E-04	9.38E-04	2.05E-06	3.19E-03
	4	1.71E-03	1.45E-04	1.24E-06	2.09E-03
16	2	7.63E-06	2.40E-06	2.91E-11	1.65E-05
	4	7.63E-06	3.55E-07	1.96E-11	9.13E-06
	6	7.63E-06	4.22E-08	1.94E-11	7.94E-06
	8	7.57E-06	2.29E-08	1.93E-11	7.66E-06
32	4	1.02E-10	4.18E-11	6.40E-21	2.12E-10
	8	1.16E-10	2.27E-12	4.52E-21	1.21E-10
	12	1.16E-10	1.19E-13	4.52E-21	1.17E-10
	14	1.16E-10	2.67E-14	4.52E-21	1.16E-10
	15	1.16E-10	1.16E-14	4.52E-21	1.16E-10

6.3 Errors Comparison

Define E_{avg} is average error, E_{sqr} is mean square error and E_{max} is maximum error. A C program was written for calculating these errors with various input length and number of column to be kept k.

For squaring units, methods in [11], [74], [71] and [77] are used for generating PPM. CCT technique is applied to each unit, a correction constant C is calculated. The output of each truncated unit is produced, and each type of error is generated. Table 6.1, 6.2, 6.3 and 6.4 are errors comparison of truncated squaring units for each method above. As can be seen in these tables, errors reduce when value of k is increased.

Table 6.5 represents RNE errors of a squaring unit. Table 6.6, 6.7 shows the average, mean square and maximum absolute error, and the correction constant for several truncated cubing units in both [75] and [76] methods. According to this Table,

Table 6.2: Errors Comparison for Truncated Squaring Units in [74]

n	k	C	E_{avg}	E_{sqr}	E_{max}
8	2	1.95E-03	3.81E-05	1.24E-06	2.21E-03
	4	1.71E-03	1.45E-04	1.24E-06	2.09E-03
16	2	1.14E-05	1.41E-06	2.55E-11	1.27E-05
	4	8.58E-06	5.69E-07	1.98E-11	8.85E-06
	6	7.87E-06	1.81E-07	1.94E-11	7.87E-06
	8	7.63E-06	2.29E-08	1.93E-11	7.66E-06
32	4	1.46E-10	1.82E-12	4.65E-21	1.69E-10
	8	1.18E-10	4.56E-13	4.52E-21	1.19E-10
	12	1.16E-10	5.13E-14	4.52E-21	1.17E-10
	14	1.16E-10	1.59E-14	4.52E-21	1.16E-10
	15	1.16E-10	1.68E-14	4.52E-21	1.16E-10

all of these errors will be reduced when the number of keeping column k is increased, but the trade off is the increasing in the number of column in partial product matrix, hence, their area and delay will be increase. Table 6.8 shows the average, mean square and maximum absolute error of rounding to nearest even, in which r is the length of cubing unit we want to keep. According to Table 6.6 and Table 6.7, all of these errors in Table 6.6 in which $k = n$ are smaller or equal these one in the Table 6.7. This indicates that truncated cubing unit with $k = n$ produces the output that is the same as that one of rounding to nearest even with $r = n$.

Table 6.3: Errors Comparison for Truncated Squaring Units in [71]

n	k	C	E_{avg}	E_{sqr}	E_{max}
8	2	1.95E-03	2.29E-05	1.25E-06	2.33E-03
	4	1.95E-03	9.91E-05	1.20E-06	1.94E-03
16	2	7.63E-06	1.35E-06	2.26E-11	1.13E-05
	4	7.63E-06	2.66E-07	1.95E-11	8.46E-06
	6	7.63E-06	3.49E-10	1.93E-11	7.75E-06
	8	7.63E-06	2.95E-08	1.93E-11	7.63E-06
32	4	1.16E-10	6.53E-12	4.58E-21	1.31E-10
	8	1.16E-10	4.25E-13	4.52E-21	1.17E-10
	12	1.16E-10	1.76E-14	4.52E-21	1.16E-10
	14	1.16E-10	9.19E-20	4.52E-21	1.16E-10
	15	1.16E-10	1.78E-15	4.52E-21	1.16E-10

Table 6.4: Errors Comparison for Truncated Squaring Units in [77]

n	k	C	E_{avg}	E_{sqr}	E_{max}
8	2	1.95E-03	2.28E-05	1.25E-06	2.33E-03
	4	1.95E-03	9.91E-05	1.20E-06	1.94E-03
16	2	7.63E-06	1.35E-06	2.26E-11	1.13E-05
	4	7.63E-06	2.66E-07	1.95E-11	8.46E-06
	6	7.63E-06	3.49E-10	1.93E-11	7.75E-06
	8	7.63E-06	2.95E-08	1.93E-11	7.63E-06
32	4	1.16E-10	6.53E-12	4.58E-21	1.31E-10
	8	1.16E-10	4.25E-13	4.52E-21	1.17E-10
	12	1.16E-10	1.76E-14	4.52E-21	1.16E-10
	14	1.16E-10	9.19E-20	4.52E-21	1.16E-10
	15	1.16E-10	1.78E-15	4.52E-21	1.16E-10

Table 6.5: Rounding to Nearest Even (RNE) Errors of Squaring Unit

n	r	E_{avg}	E_{sqr}	E_{max}
8	8	-9.92E-05	1.20E-06	1.94E-03
16	16	-2.95E-08	1.93E-11	7.63E-06
32	32	-1.77E-15	4.52E-21	1.164E-10

Table 6.6: Errors Comparison for Truncated Cubing Unit in [75]

n	k	C	E_{avg}	σ^2_{avg}	E_{max}
4	4	3.125E-02	9.766E-03	2.135E-04	3.052E-02
8	4	1.953E-03	1.106E-04	1.289E-06	2.127E-03
8	8	1.953E-03	1.907E-05	1.271E-06	1.953E-03
16	8	7.629E-06	6.083E-08	1.924E-11	7.725E-06
16	12	7.629E-06	1.799E-08	1.924E-11	7.632E-06
16	16	7.629E-06	1.543E-08	1.924E-11	7.629E-06
32	24	1.164E-10	9.252E-15	4.516E-21	1.164E-10
32	32	1.164E-10	9.252E-15	4.156E-21	1.164E-10

Table 6.7: Errors Comparison for Truncated Cubing Unit in [76]

n	k	C	E_{avg}	σ^2_{avg}	E_{max}
4	4	3.125E-02	9.766E-04	2.135E-04	3.052E-02
8	4	1.953E-03	2.785E-04	1.409E-06	2.632E-03
8	8	1.953E-03	1.907E-05	1.271E-06	1.953E-03
16	8	7.629E-06	3.326E-07	1.937E-11	8.603E-06
16	12	7.629E-06	3.568E-08	1.924E-11	7.670E-06
16	16	7.629E-06	1.636E-08	1.924E-11	7.630E-06
24	24	2.980E-08	9.727E-12	2.955E-16	2.980E-08
24	26	2.980E-08	9.695E-12	2.955E-16	2.980E-08

Table 6.8: Error Comparison of Rounding to Nearest Even(RNE)

n	r	E_{avg}	$\sigma_e^2 vg$	E_{max}
4	4	9.766E-04	2.135E-04	3.052E-02
8	8	1.907E-05	1.271E-06	1.953E-03
16	16	1.543E-08	1.924E-11	7.629E-06
32	32	9.130E-15	4.516E-21	1.164E-10

CHAPTER 7

Area and Delay Analyses

Previous chapters presented some types of squaring and cubing units and proposed a new model for implementing both squaring and cubing units. These units, then are used for truncation. This chapter will analyses to make a performance comparison in area, delay and power consumption of each units in previous chapters. The method is used to do that is *Linear-Delay* analyses. Although this approach is simplistic, it is helpful in establishing algorithmically the complexity of each algorithm. More elaborate and more detailed models can also be utilized, however, research has previously demonstrated that linear-delay analysis is a good method in assessing algorithmic complexity for a given design.

According to [78], a *Dadda* [5] is better than a Wallace [4] reduction scheme in both delay and area performance. Hence, in this dissertation, a *Dadda* tree will be used for all units instead of Wallace tree. To estimate the area, the total number of AND gates for generating the PPM and the total number of HA, FA using in *Dadda* tree reduction are counted equally regardless of the gate or fan-out utilized (e.g., AND gate). For example, Figure 7.1 shows a simple half-adder gate that consumes *4* gates and has a critical path of 3Δ.

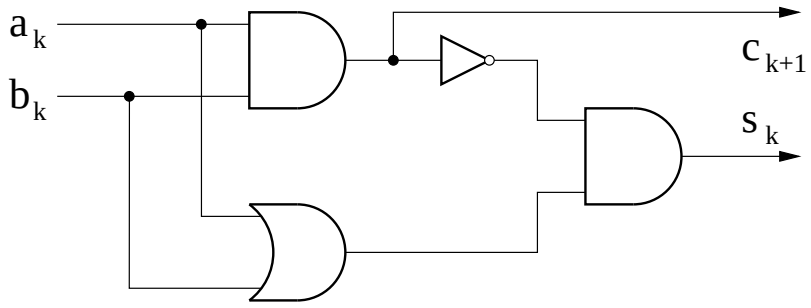

Figure 7.1: Half Adder Implementation

7.1 Area Estimation of Squaring Units.

7.1.1 Folded Squaring Unit.

According to Figure 4.7 , to form the folded matrix, the number of AND gates are used for generating $x_i \cdot x_j$ $(i \neq j)$ is

$$N_{AND} = \frac{n(n-1)}{2} \tag{7.1}$$

The folded matrix is established by n bits of x_i, and $\frac{n(n-1)}{2}$ bits of $x_i x_j$, so the total number of bits in this matrix can be calculates as

$$\begin{aligned} N_{PPB} &= n + \frac{n(n-1)}{2} \\ &= \frac{n^2+n}{2} \end{aligned} \tag{7.2}$$

Because $Dadda$ tree is used for column compression, the number of HAs is

$$N_{HA} = n - 1 \tag{7.3}$$

As can be seen, each FA takes the 3 inputs and produces 2 outputs, hence, the number of bits in the PPM is reduced by 1 after using 1 FA for reduction. Therefore, the total number of FA in all stages is the differential between the number of bits in original PPM and the length of final result, which is $2n$ bit, but the bit in the column

1^{th} is always equal 0, so the number of FAs can be calculated as

$$\begin{aligned} N_{FA} &= N_{PPB} - (2n-1) \\ &= \frac{n^2+n}{2} - (2n-1) \\ &= \frac{n^2-3n}{2} + 1 \end{aligned} \tag{7.4}$$

According to Figure 7.1 each HA contains 4 gates, ones can easy get that a FA will contain 9 gates. Hence, the total number of gates using in folded method is calculated as

$$\begin{aligned} N_{folded} &= N_{AND} + 4N_{HA} + 9N_{FA} \\ &= 5n^2 - 10n + 5 \end{aligned} \tag{7.5}$$

7.1.2 Merged Squaring Unit.

According to Figure 4.7, each term $x_{i+1}\overline{x_i}$ with $i = 0 \cdots n-2$ is established by 1 $Inverter$ and 1 AND gate. Hence, the number of $Inverters$ and AND gates are used to generate merged matrix are

$$\begin{aligned} N_{INV} &= n-1 \\ N_{AND} &= n-1+\frac{n(n-1)}{2} \\ &= \frac{(n+2)(n-1)}{2} \end{aligned} \tag{7.6}$$

Because the merged method groups some of two bits in folded matrix and generates two other bits, so the total number of partial product bits in merged matrix is the same as that one in folded matrix. According to Equation 7.2, this number is

$$N_{PPB} = \frac{n^2+n}{2} \tag{7.7}$$

The number of HAs for this matrix when applied $Dadda$ tree for column compression is

$$N_{HA} = n-2 \tag{7.8}$$

The number of FAs is the differential between the total bits in PPM and the length of final result can be calculated as

$$\begin{aligned} N_{FA} &= N_{PPB} - (2n-1) \\ &= \frac{n^2+n}{2} - (2n-1) \\ &= \frac{n^2-3n+2}{2} \end{aligned} \tag{7.9}$$

The total number of gates for merged method can be calculated as

$$\begin{aligned} N_{merged} &= N_{INV} + N_{AND} + 4N_{HA} + 9N_{FA} \\ &= 5n^2 - 8n - 1 \end{aligned} \tag{7.10}$$

7.1.3 Divide & Conquer Squaring Unit

For the squaring unit in [71], to calculate the output of $2m$-bit input, squaring of the m-bit and the multiplier of the m-bit are calculated first, then Equation 4.3 is applied to get the final result. These steps are repeated until the input is equal to 4. Hence, to apply this method, the length of input should be power of 2. Define $n = 2^k$, the total number of gates for a 2^k-bit squaring unit will be expressed as

$$N_{sqr(2^k)} = 2 \cdot N_{sqr(2^{k-1})} + N_{mult(2^{k-1})} + N_{Dadda(2^{k-1})} \tag{7.11}$$

The total gates of (2^{k-1})-bit squaring unit is

$$N_{sqr(2^{k-1})} = 2 \cdot N_{sqr(2^{k-2})} + N_{mult(2^{k-2})} + N_{Dadda(2^{k-2})} \tag{7.12}$$

As can be seen in Figure 4.6, $N_{sqr(2)} = 4$, $N_{mult(2)} = 7$, $N_{Dadda(2)} = 30$, total gates of 4-bit squaring unit is

$$\begin{aligned} N_{sqr(4)} &= 2 \cdot N_{sqr(2)} + N_{mult(2)} + N_{Dadda(2)} \\ &= 45 \end{aligned} \tag{7.13}$$

Replace Equation 7.12, 7.13 to 7.11, total bit of a 2^k-bit squaring unit is calculated as

$$N_{sqr(2^k)} = 2^{k-2} \cdot N_{sqr(4)} + \sum_{i=2}^{k-1} 2^{k-i-1} \cdot N_{mult(2^i)} + \sum_{i=2}^{k-1} 2^{k-i-1} \cdot N_{Dadda(2^i)} \quad (7.14)$$

According to Equation 2.4, total gates of 2^i-bit multiplier is

$$N_{mult(2^i)} = 10 \cdot (2^i)^2 - 14 \cdot (2^i) \quad (7.15)$$

According to Figure 4.5, total gates for 2^i-bit squaring unit in *Dadda* tree is

$$N_{Dadda(2^i)} = 11 \cdot (2^i) - 9 \quad (7.16)$$

Replace Equation 7.13, 7.15, 7.16 to 7.14, total gates for this unit is

$$N_{Divide\&Conquer} = 5n^2 - \frac{53n}{4} - \frac{3n}{2} \cdot (\log_2(m) - 2) + 9 \quad (7.17)$$

7.1.4 Proposed Squaring Unit.

According to Figure 4.7, in the Stage 1, $(n-2)$ special HAs are used. Each of these adders takes 2 inputs and produces 2 output. Hence, the number of bits in the matrix after this stage does not change. But in the Stage 2, $(n-3)$ special full adders are used, each of them takes 3 inputs and produces 2 outputs, so 1 bit is reduced. The same thing happen with the Stage 3, $(n-3)$ special half adders are used, each of them takes 2 inputs and produces 1 output, so it can reduce matrix by one bit. In the column $(2n-2)^{th}$, 2 bits $x_{n-1} \cdot \overline{x_{n-2}}$ and $x_{n-1} \cdot x_{n-2} \cdot x_{n-3}$ are combined to 1 bit, hence, the total partial product bits in proposed matrix is

$$\begin{aligned} N_{PPB} &= \frac{n^2+n}{2} - (n-3) - (n-3) - 1 \\ &= \frac{n^2-3n}{2} + 5 \end{aligned} \quad (7.18)$$

The number of HAs for *Dadda* tree reduction applied to this PPM

$$N_{HA} = n - 2 \quad (7.19)$$

The number of FAs is the differential between the partial product bits in the matrix and the length of final result, so it can be calculated as

$$\begin{aligned} N_{FA} &= N_{PPB} - (2n-1) \\ &= \frac{n^2-3n}{2} + 5 - (2n-1) \\ &= \frac{n^2-7n}{2} + 6 \end{aligned} \tag{7.20}$$

As in Equation 4.5, 4.6, to generate partial product bits S^2_{2i} and S^3_{2i+1}, 8 and 6 gates are needed, respectively. According to Figure 4.8, 4.9, each partial product bit in column 3^{th} and $(2n-2)^{th}$ requires 3 gates, and $[\frac{n^2-3n}{2} + 2(n-3)]$ AND gates are used to generate the rest of partial product bits in the proposed matrix. Hence total number of gates for this method can be calculated as

$$\begin{aligned} N_{proposed} &= 8(n-3) + 6(n-3) + 6 + \frac{n^2-7n}{2} + 11 \\ &\quad +4N_{HA_{squarer_{proposed}}} + 9N_{FA_{squarer_{proposed}}} \\ &= 5n^2 - 17n + 17 \end{aligned} \tag{7.21}$$

7.1.5 Booth 2 Folding Squaring Unit.

According to Figure 6.1, each array P_i will run from $(-2i+3)^{th}$ to $(-4i)^{th}$, but the bit in position $(-4i+1)^{th}$ is always zero, so the length of P_i is $2i+3$-bit. Hence the total partial product bits in this matrix is calculated as

$$\begin{aligned} N_{PPB} &= 1 + \sum_{i=1}^{m} (2i+3) \\ &= m^2 + 2m \\ &= \frac{n^2}{4} + n \end{aligned} \tag{7.22}$$

The number of HAs can be approximated as

$$N_{HA} \approx \frac{3n}{4} - 1 \tag{7.23}$$

The number of FAs is

$$\begin{aligned} N_{FA} &= N_{PPB} - (2n-1) \\ &= \frac{n^2}{4} - n + 1 \end{aligned} \tag{7.24}$$

According to Equation 4.13 and 4.14, the number of gates to generate P_i is

$$\begin{aligned} N_{P_i} &= 5 + 9(2i-2) + 3 + 3 + 1 \\ &= 18i - 6 \end{aligned} \tag{7.25}$$

The total gates to generate this matrix is

$$\begin{aligned} N_{gate_{matrix}} &= \sum_{i=1}^{m} (N_{P_i}) \\ &= \frac{9n^2}{4} - \frac{15n}{2} \end{aligned} \tag{7.26}$$

From Equation 7.23, 7.24 and 7.26, total gate for squaring unit in *Booth 2 Folding* method can be calculated as

$$\begin{aligned} N_{Strollo} &= 4N_{HA} + 9N_{FA} + N_{gate_{matrix}} \\ &= \frac{9n^2}{2} - \frac{27n}{2} + 5 \end{aligned} \tag{7.27}$$

7.1.6 Booth 2 Left-to-Right Recoding

Like *Booth 2 Folding* technique, the number of partial product bits in *Booth 2 Left-to-Right Recoding* can be calculated as

$$\begin{aligned} N_{PPB} &= 3 + \sum_{i=0}^{m-1} (2m - 2i + 2) \\ &= \frac{n^2}{4} + \frac{5n}{2} - 2 \end{aligned} \tag{7.28}$$

The number of *HAs* can be approximated as

$$N_{HA} \approx \frac{13n}{8} - 9 \tag{7.29}$$

Total FAs for this unit is

$$\begin{aligned} N_{FA} &= N_{PPB} - (2n-1) \\ &= \frac{n^2}{4} - \frac{n}{4} - 1 \end{aligned} \tag{7.30}$$

According to Equation 4.20, the number of gated for generating P_i is

$$\begin{aligned} N_{P_i} &= 7 + 5 + 10(2m - 2i - 2) + 8 + 4 \\ &= 20m - 20i + 4 \end{aligned} \tag{7.31}$$

$$\begin{aligned} N_{gate_{matrix}} &= 5 + \sum_{i=0}^{m-1} (N_{P_i}) \\ &= \frac{5n^2}{2} + 17n - 15 \end{aligned} \tag{7.32}$$

From Equation 7.29, 7.30 and 7.32, total gate for squaring unit in *Booth* 2 *Left-to-Right Recoding* technique can be calculated as

$$\begin{aligned} N_{Matula} &= 4N_{HA} + 9N_{FA} + N_{gate_{matrix}} \\ &= \frac{19n^2}{4} + \frac{51n}{4} - 60 \end{aligned} \tag{7.33}$$

7.2 Delay Estimation of Squaring Units

The delay of squarer unit depends on the delay of generating PPM, the height of this matrix and the length of CPA. Define Δ is delay for one gate, these parameters can be calculated as

- Delay for generating PPM:

$$\begin{aligned} t_{ppm_{folded}} &= 1\Delta \\ t_{ppm_{merged}} &= 2\Delta \\ t_{ppm_{Divide\&Conquer}} &= t_{ppm_{multiplier}} \\ t_{ppm_{Strollo}} &= 5\Delta \\ t_{ppm_{Matula}} &= 5\Delta \\ t_{ppm_{proposed}} &= 3\Delta \end{aligned} \tag{7.34}$$

- The height of PPM: According to Figure 4.1 and 4.2, the shape of PPM is different when n is odd or even, but its height can be formulated as:

$$PPH_{folded} = \left\lfloor \frac{n}{2} \right\rfloor + 1 \tag{7.35}$$

For merged method, 2 bits $x_i x_{i-1}$ and x_i in the same column are grouped together. It produces 2 other bits, one in the same and one in the next column. The shape of this matrix depends on either n is odd or even. For whatever n, the height of merged matrix can be calculated as

$$PPH_{squarer_{merged}} = \left\lceil \frac{n}{2} \right\rceil \tag{7.36}$$

For the Divide & Conquer method, the height of PPM is always equal 2

$$PPH_{sqr_{Divide\&Conquer}} = 2 \tag{7.37}$$

According to Figure 4.9, proposed matrix is established by reducing merged PPM 2 more steps. The height of proposed matrix is

$$PPH_{proposed} = \left\lceil \frac{n}{2} \right\rceil - 1 \tag{7.38}$$

According to Figure 4.11, 4.12 the matrix height of squaring unit using *Booth*

2 *folding* and *Left-to-Right* recoding are calculated as

$$\begin{aligned} PPH_{Strollo} &= \left\lceil \frac{n+3}{4} \right\rceil + 1 \\ PPh_{Matula} &= \left\lceil \frac{n+3}{4} \right\rceil \end{aligned} \tag{7.39}$$

- The length of CPA: The carry and sum array of folded method run from the column 2^{th} to the $(2n-2)^{th}$, so the length of CPA can be calculated as

$$\begin{aligned} CPA_{folded} &= (2n-2) - 2 + 1 \\ &= 2n - 3 \end{aligned} \tag{7.40}$$

The carry and sum arrays of merged method run from the column 3^{th} to the $(2n-1)^{th}$, so the length of CPA can be calculated as

$$\begin{aligned} CPA_{merged} &= (2n-1) - 3 + 1 \\ &= 2n - 3 \end{aligned} \tag{7.41}$$

The carry and sum of Divide & Conquer method run from $(m+1)^{th}$ to $(4m)^{th}$, so the length of CPA can be calculated as

$$\begin{aligned} CPA_{Divide\&Conquer} &= (4m) - (m+1) + 1 \\ &= 3m \end{aligned} \tag{7.42}$$

The carry and sum arrays of proposed methods run from the column 5^{th} to the $(2n-1)^{th}$, so the length of CPA can be calculated as

$$\begin{aligned} CPA_{proposed} &= (2n-1) - 5 + 1 \\ &= 2n - 5 \end{aligned} \tag{7.43}$$

The carry and sum arrays of *Booth* 2 *folding* method run from 4^{th} to the $(2n-1)^{th}$, so the length of CPA can be calculated as

$$\begin{aligned} CPA_{Strollo} &= (2n-1) - 4 + 1 \\ &= 2n - 4 \end{aligned} \tag{7.44}$$

The carry and sum arrays of *Booth 2 Left-to-Right* method run from 3^{th} to the $(2n-1)^{th}$ so the length of CPA can be calculated as

$$\begin{aligned} CPA_{Matula} &= (2n-1) - 34 + 1 \\ &= 2n - 3 \end{aligned} \tag{7.45}$$

According to [31], with the length of CPA is k the delay of RCA can be calculated as

$$t_{RCA} = (2k+2)\Delta \tag{7.46}$$

So the delay of CPA for each method can be represented as

$$\begin{aligned} t_{CPA_{folded}} &= t_{CPA_{merged}} \\ &= t_{CPA_{Matula}} \\ &= [2(2n-3)+2]\Delta \\ &= (4n-4)\Delta \\ t_{CPA_{proposed}} &= [2(2n-5)+2]\Delta \\ &= (4n-8)\Delta \\ t_{CPA_{Strollo}} &= [2(2n-4)+2]\Delta \\ &= (4n-4)\Delta \end{aligned} \tag{7.47}$$

7.3 Area Estimation of Cubing Units

7.3.1 Liddicoat and Flynn Method.

According to Figure 5.2, 5.3 the terms $x_i x_j$ appear two times and the terms $x_i \cdot x_j \cdot x_k$ appear one time in the minimized matrix, so the number AND gates in *Liddicoat's* method is calculated as

$$\begin{aligned} N_{AND} &= C_2^{2m} + C_3^{2m} \\ &= \frac{4m^3 - m}{3} \end{aligned} \tag{7.48}$$

and the number of bits in this matrix is

$$\begin{aligned} N_{PPB} &= 2 \times C_2^{2m} + C_3^{2m} \\ &= \frac{4m^3 + 6m^2 - 4m}{3} \end{aligned} \tag{7.49}$$

Because this matrix is reduced to get sum and carry arrays, which run from the row 1^{th} through the row $(6m-2)^{th}$, hence, if S is the number of Wallace's stages, so the total bits in sum and carry arrays can be calculated as

$$\begin{aligned} N_{sum} &= [6m-2] - 1 + 1 \\ &= 6m - 2 \\ N_{carry} &= [6m-2] - 1 - S \\ &= 6m - 3 - S \end{aligned} \tag{7.50}$$

The carry and sum arrays have weight of 3, they must be doubled and combined with $2m$ of $1x$ term x_i to calculate the final result. Because the final result has $6m$-bit length, so the total FAs for $Liddicoat's$ cubing unit is

$$\begin{aligned} N_{FA} &= N_{PPB} - (N_{sum} + N_{carry}) \\ &\quad +2(N_{sum} + N_{carry}) + 2m - 6m \\ &= \frac{4m^3 + 6m^2 + 20m}{3} - 5 - S \end{aligned} \tag{7.51}$$

The number of HAs can be calculates as

$$N_{HA} \approx \frac{m^2}{6} + 32m - 60 \tag{7.52}$$

From Equation 7.48, 7.51, 7.52, the total gates for implementing $Liddicoat's$ cubing unit is

$$\begin{aligned} N_{Lidd} &= N_{AND} + 4N_{HA} + 9N_{FA} \\ &\approx \frac{40m^3 + 55m^2 + 564m}{3} - 285 - 9S \end{aligned} \tag{7.53}$$

7.3.2 Stine and Blank method.

For *Stine's* cubing unit in [25], $3\times$ terms in each row are replaced by two bits, one in this row and one in the next row on the left hand side. The same thing is applied for the $6\times$ terms but these two bits are shifted one position to the left. In the step of improving the regularity, three terms $x_i \cdot x_{i-1}$ with two are in the same row $(3i-1)^{th}$ and one in the next row $(3i)^{th}$, are replaced by only one term in the row $(3i+1)^{th}$, so two bits are reduced. Because i runs from 1 to $2m$, hence the total number of bits that are saved in this step is $2(2m-1)$. With this approach, the total number of AND gates and bits in this matrix are calculated as

$$\begin{aligned} N_{AND} &= C_2^{2m} + C_3^{2m} \\ &= \frac{4m^3 - m}{3} \end{aligned} \tag{7.54}$$

$$\begin{aligned} N_{PPB} &= 4C_2^{2m} + 2C_3^{2m} + 2m - 2(2m-1) \\ &= \frac{8m^3 + 12m^2 - 14m}{3} + 2 \end{aligned} \tag{7.55}$$

Because *Dadda* tree is applied for column compression, the number of HAs and FAs can be calculated as

$$N_{HA} \approx \frac{15m}{4} \tag{7.56}$$

$$\begin{aligned} N_{FA} &= N_{PPB} - 6m \\ &= \frac{8m^3 + 12m^2 - 32m}{3} + 2 \end{aligned} \tag{7.57}$$

Hence, the total gates of *Stine's* cubing unit is

$$\begin{aligned} N_{Stine} &= N_{AND} + 4N_{HA} + 9N_{FA} \\ &\approx \frac{76m^3 + 108m^2 - 274m}{3} + 18 \end{aligned} \tag{7.58}$$

7.3.3 Divide and Conquer method.

For cubing unit in [75], to calculate the output of $2m$-bit input, cubing of m-bit and squarer-multiply of m-bit, must be calculated first. After that using Equation 5.3 to get the final result. These iterations are repeated until the input is equal to 4. So $n = 2m$ should be power of 2. In general, set $n = 2^k$, the total number of gates of 2^k-bit cubing unit is expressed as

$$N_{cube(2^k)} = 2N_{cube(2^{k-1})} + 2N_{sqrmult(2^{k-1})} + N_{Dadda(2^{k-1})} \tag{7.59}$$

The total gates of (2^{k-1})-bit cubing unit and then is calculated as

$$N_{cube(2^{k-1})} = 2N_{cube(2^{k-2})} + 2N_{sqrmult(2^{k-2})} + N_{Dadda(2^{k-2})} \tag{7.60}$$

Replace Equation 7.60 to Equation 7.59 ones have

$$\begin{aligned} N_{cube(2^k)} &= 2^{k-2}N_{cube(2^2)} \\ &\quad +2N_{sqrmult(2^{k-1})} + 2^2N_{sqrmult(2^{k-2})} + \ldots + 2^{k-2}N_{sqrmult(2^2)} \\ &\quad +N_{Dadda(2^{k-1})} + 2N_{Dadda(2^{k-2})} + \ldots + 2^{k-3}N_{Dadda(2^2} \\ &= 2^{k-2}N_{cube(2^2)} + \sum_{i=2}^{k-1} 2^{k-i}N_{sqrmult(2^i)} + \sum_{i=2}^{k-1} 2^{k-i-1}N_{Dadda(2^i)} \end{aligned} \tag{7.61}$$

When the length of operand X is 4, $X = x_3x_2x_1x_0$, the value of A and B can be expressed as $A = x_3x_2$, and $B = x_1x_0$, so the value of A^3, B^3, $A^2 \cdot B$, $A \cdot B^2$ will have

five bits length. They can be represented as

$$\begin{aligned}
A^3 &= (x_3x_2)^3 \\
&= (x_3 \cdot x_2)\ (x_3)\ (0)\ (x_3 \cdot x_2)\ (x_2) \\
B^3 &= (x_1x_0)^3 \\
&= (x_1 \cdot x_0)\ (x_1)\ (0)\ (x_1 \cdot x_0)\ (x_0) \\
A^2B &= (x_3x_2)^2(x_1x_0) \\
&= (x_3 \cdot x_2 \cdot x_1)\ (F)\ (x_3 \cdot \overline{x_2} \cdot x_0)\ (x_2 \cdot x_1)\ (x_2x_0) \\
AB^2 &= (x_3x_2)(x_1x_0)^2 \\
&= (x_3 \cdot x_1 \cdot x_0)\ (E)\ (x_2 \cdot x_1 \cdot \overline{x_0})\ (x_3 \cdot x_0)\ (x_2 \cdot x_0)
\end{aligned} \tag{7.62}$$

in which

$$\begin{aligned}
E &= x_2 \cdot x_1 \cdot x_0 + x_3 \cdot x_1 \cdot \overline{x_0} \\
F &= x_3 \cdot x_2 \cdot x_0 + x_3 \cdot \overline{x_2} \cdot x_1
\end{aligned} \tag{7.63}$$

The partial product matrix of a 4-bit cubing can be generated as in the Figure 5.7. Implementation of this unit in *Dadda* tree can be seen as in Figure 7.2 . To establish E and F, 8 gates are needed. Hence, to generate this matrix total 16 gates are used. According to this figure, 6 HAs and 16 FAs are used to get the final result. So the total gates of a 4-bit cubing unit is

$$\begin{aligned}
N_{cube(2^2)} &= 16 + 6 \times 4 + 16 \times 9 \\
&= 184(gates)
\end{aligned} \tag{7.64}$$

In [75], the folded method is used to get the squarer. According to Equation 7.5, if the length of input is 2^i, the number of gates used to build squarer unit is

$$N_{sqr(2^i)} = 5(2^i)^2 - 10(2^i) + 5 \tag{7.65}$$

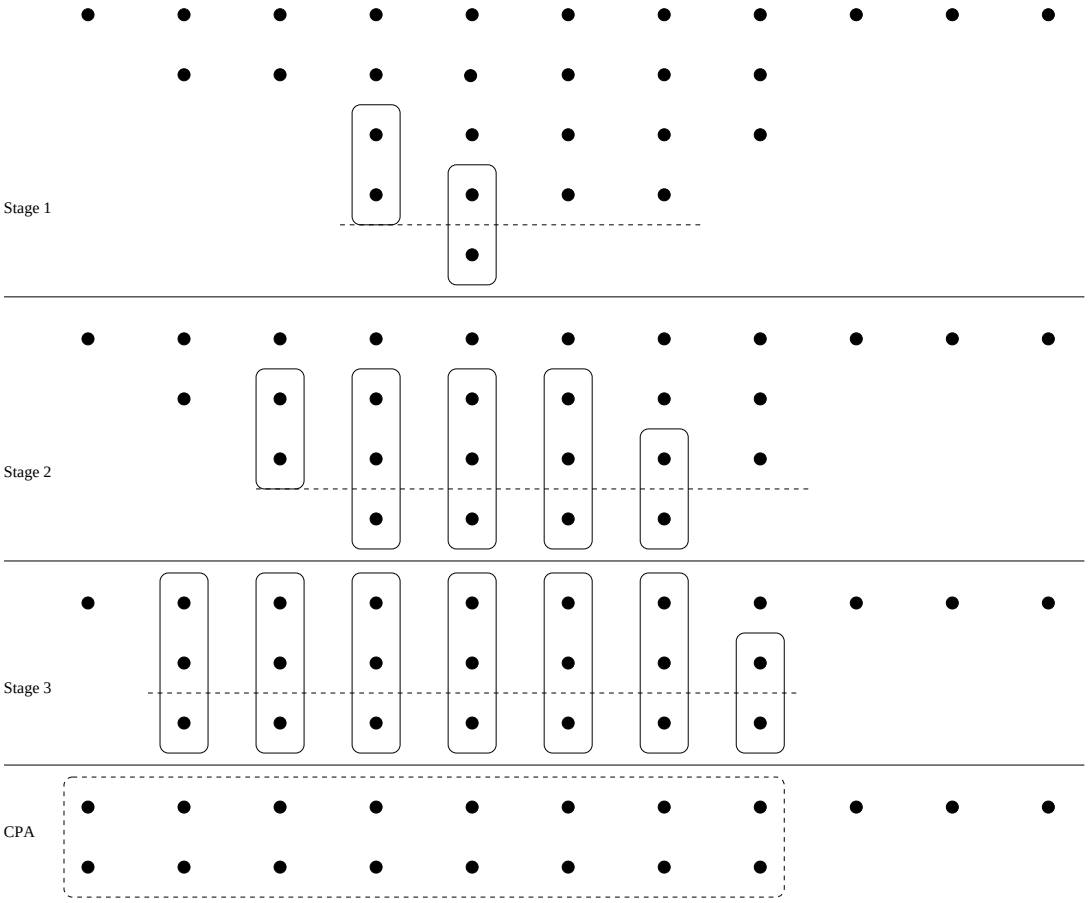

Figure 7.2: Dadda Tree Reduction of a 4 Cubing Unit in Vedic Mathematics

The number of AND gates to establish PPM of squarer-multiplier unit is

$$\begin{aligned} N_{AND_{sqrmult(2^i)}} &= [2(2^i)-1](2^i) \\ &= 2(2^i)^2-(2^i) \end{aligned} \tag{7.66}$$

Because *Dadda* tree is applied to reduce this matrix, so the number of HAs and FAs can be calculated as

$$N_{HA_{sqrmult(2^i)}} = 2^i \tag{7.67}$$

$$\begin{aligned} N_{FA_{sqrmult(2^i)}} &= [2(2^i)-1](2^i)-3(2^i) \\ &= 2(2^i)^2-4(2^i) \end{aligned} \tag{7.68}$$

Hence, the total gates using in squarer-multiplier unit is

$$\begin{aligned} N_{sqrmult(2^i)} &= N_{sqr(2^i)}+N_{AND_{sqrmult(2^i)}} \\ &\quad +4N_{HA_{sqrmult(2^i)}}+9N_{FA_{sqrmult(2^i)}} \\ &= 25(2^i)^2-43(2^i)+5 \end{aligned} \tag{7.69}$$

According to Figure 5.6 the number of HAs and FAs used in (2^i)-bit *Dadda* tree are

$$N_{HA_{Dadda(2^i)}} = 2(2^i) \tag{7.70}$$

$$N_{FA_{Dadda(2^i)}} = 12(2^i) \tag{7.71}$$

The total gates used in *Dadda* tree is

$$\begin{aligned} N_{Dadda(2^i)} &= 4N_{HA_{Dadda(2^i)}}+9N_{FA_{Dadda(2^i)}} \\ &= 116(2^i) \end{aligned} \tag{7.72}$$

Replace Equation 7.64, 7.69 , and 7.72 into 7.61, the total gates for *Divide* & *Conquer* method can be calculated as

$$N_{Divide\&Conquer} = 100m^2-133m+15m\log_2(m)[1+log_2(m)]-10 \tag{7.73}$$

7.3.4 Proposed method.

For proposed method, the input X is break into smaller pieces. Each piece X_i has 2-bit length, so the each terms X_i^3, $X_i^2 \cdot X_j$, $X_i \cdot X_j^2$, $X_i \cdot X_j \cdot X_k$ will have 5-bit length. According to Equation 7.62, to get the output of X_i^3, only one gate is needed. $X_i^2 \cdot X_j$, $X_i \cdot X_j^2$ can be expressed as

$$\begin{aligned}
X_{i^2j} &= X_i^2 X_j \\
&= (x_{2i+1}x_{2i})^2 \cdot (x_{2j+1}x_{2j}) \\
X_{i^2j}[0] &= x_{2i} \cdot x_{2j} \\
X_{i^2j}[1] &= x_{2i} \cdot x_{2j+1} \\
X_{i^2j}[2] &= x_{2i+1} \cdot \overline{x_{2i}} x_{2j} \\
X_{i^2j}[3] &= x_{2i+1} \cdot x_{2i} \cdot x_{2j} + x_{2i+1} \cdot \overline{x_{2i}} x_{2j+1} \\
X_{i^2j}[4] &= x_{2i+1} \cdot x_{2i} \cdot x_{2j+1} \\
X_{ij^2} &= X_i X_j^2 \\
&= (x_{2i+1}x_{2i}) \cdot (x_{2j+1}x_{2j})^2 \\
X_{ij^2}[0] &= x_{2i} \cdot x_{2j} \\
X_{ij^2}[1] &= x_{2i+1} \cdot x_{2j} \\
X_{ij^2}[2] &= x_{2i} \cdot x_{2j+1} \cdot \overline{x_{2j}} \\
X_{ij^2}[3] &= x_{2i} \cdot x_{2j+1} \cdot x_{2j} + x_{2i+1} \cdot x_{2j+1} \cdot \overline{x_{2j}} \\
X_{ij^2}[4] &= x_{2i+1} \cdot x_{2j+1} \cdot x_{2j}
\end{aligned} \tag{7.74}$$

As can be seen in Equation 7.74, 9 gates are needed for each $X_i^2 \cdot X_j$ or $X_i \cdot X_j^2$ unit. $X_i \cdot X_j \cdot X_k$ is implemented as in the Figure 7.3. To generate this matrix in Figure 7.3, 8 gates are used. To get the output of $X_i \cdot X_j \cdot X_k$, 1 HA and 3 FAs are employed.

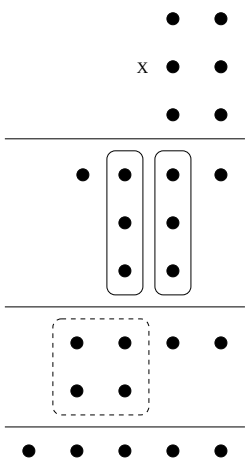

Figure 7.3: Implementation of $X_i X_j X_k$

Hence, the total number of gates for this unit is

$$\begin{aligned} N_{X_i X_j X_k} &= 8 + 4 \times 1 + 3 \times 9 \\ &= 39 \end{aligned} \tag{7.75}$$

Hence, the number of gates to generate the proposed matrix is

$$\begin{aligned} N_{matrix} &= 9 \times 2C_2^m + 39 \times C_3^m \\ &= \frac{13m^3 - 21m^2 + 8m}{2} \end{aligned} \tag{7.76}$$

The number of bits in this matrix is

$$\begin{aligned} N_{PPB} &= 5 \times 2C_2^m + 5 \times C_3^m \\ &= \frac{5}{6}(m^3 + 3m^2 - 4m) \end{aligned} \tag{7.77}$$

The number of *HAs* for *Dadda* reduction is

$$N_{HA} \approx 3m \tag{7.78}$$

As can be seen in Figure 5.8, the sum array of $3\times$ term is from row 2^{th} to row $(6m-3)^{th}$. In the final stage, the sum array is doubled and combined with $5 \times 2m$

bit of X_i^3 terms, so the number of FAs can be calculated as

$$\begin{aligned} N_{FA} &= N_{PPB} - [(6m-3) - 2 + 1] \\ &\quad +2[(6m-3) - 2 + 1] + 5 \times 2m - 6m \\ &= \frac{5}{6}(m^3 + 3m^2 + 8m) - 4 \end{aligned} \tag{7.79}$$

The number of gates for proposed cubing unit is

$$\begin{aligned} N_{Proposed} &= N_{matrix} + 4 \times N_{HA_{Proposed}} + 9 \times N_{FA_{Proposed}} \\ &= 14m^3 + 12m^2 + 76m - 36 \end{aligned} \tag{7.80}$$

7.4 Delay Estimation of Cubing Units

- Delay for generating PPM:

$$\begin{aligned} t_{ppm_{Liddicoat}} &= 1\Delta \\ t_{ppm_{Stine}} &= 1\Delta \\ t_{ppm_{Divide\&Conquer}} &= t_{squarer-multiplier}(m) \\ t_{ppm_{proposed}} &= 3\Delta \end{aligned} \tag{7.81}$$

(7.82)

- The height of PPM:

$$\begin{aligned} PPH_{cube_{Liddicoat}} &= \frac{1}{2}m^2 + \frac{1}{2}m \\ PPH_{cube_{Stine}} &= m^2 + m \\ PPH_{cube_{Divide-Conquer}} &= 5 \\ PPH_{cube_{proposed}} &= \frac{3}{8}m^2 + \frac{1}{4}m \end{aligned} \tag{7.83}$$

CHAPTER 8

Hardware Implementation and Conclusions

8.1 Hardware Implementation

8.1.1 Implementation on FPGA Technology

The design flow diagram on $FPGA$ is described as in Figure 8.1. The architecture design phase for each squaring and cubing unit was presented in previous chapter. In the HDL design entry phase, $Verilog$ code for each module is generated by using a C program. These modules are simulated in Behavioral simulation phase to verify the correction. If these modules work correctly, they will be synthesized and then implemented. Timing, area analyses and power estimation will given the performance of each unit.

In this dissertation, a $Xilinx\ XC5VTX240T\ FPGA$ device with a $FF1759$ package at speed -2 of a $Xilinx\ Virtex\ 5$ family is used. Area is estimated by calculating total number of slices that are used in each design and delay is obtained from the placed and routed layout. The power estimation are generated from a $100,000$-vector VCD file and analyzed back through $Xilinx's\ Xpower\ Analyzer$ tool. All modules are also coded to account for $fast\ carry\text{-}chains$ within the final CPA of the $FPGA$.

Table 8.1 gives the area, delay and power comparison of squaring units with the input operand sizes running from 8 to 32 bits for each method in $FPGA$ technology. As can be seen , for all values of input length, the delay of the proposed method is always smaller than those one of previous methods. The area of proposed method is greater than those one in both [56] and [74], but it is smaller than that one in [66],

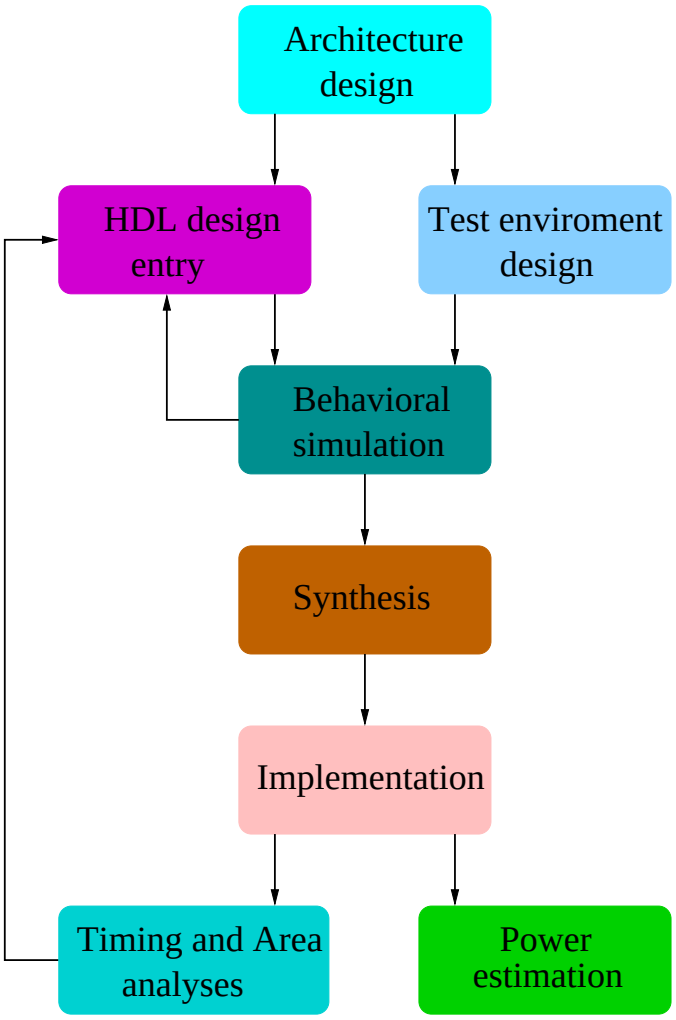

Figure 8.1: FPGA Flow Diagram

[69] and [71]. This is suitable with the analyses in previous section.

Table 8.2, gives a comparison of truncated squaring units with $k = 2, k = 4$ and $n = 8, 16, 32$, respectively. As can be seen in these tables, truncated squaring unit for proposed method always give the best performance on delay, but for area and power consumption, other methods are better. Table 8.3 gives the area, delay and power of cubing units with the input operand sizes running from 8 to 32 bits for each method in $FPGA$ technology. As can be seen in Table 8.3, the delay of the proposed method are significantly smaller than those one of previous methods, but the area is greater than that one in [75]. However, the method in [75] is computed

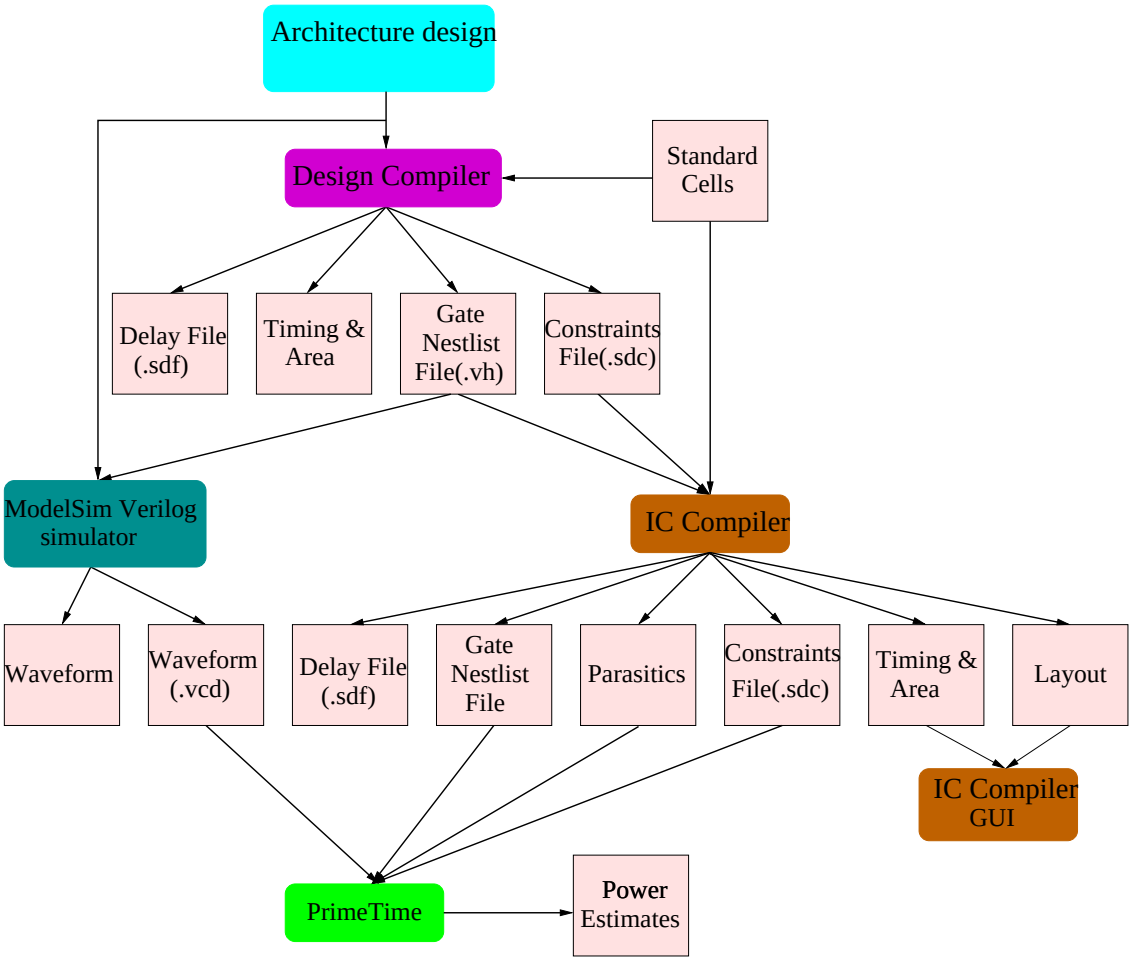

Figure 8.2: ASIC Flow Diagram

serially and consumes much delay for larger operand sizes. Therefore, it is conceivable that the method in [75] is not practical for operands greater than 8-bit. On the other hand, the proposed method demonstrates significant saving in area (30.2%) and delay (9.2%) over method presented by *Liddicoat/Flynn* and could potentially make cubing readily available for many 32-bit and 64-bit architectures. For truncated cubing units, method in [76] will be used. Comparison is given as in Table 8.4

8.1.2 Implementation on ASIC Technology

The Application Specific Integrated Circuit *ASIC* flow diagram is presented as in Figure 8.2. A Hardware Description Language Verilog code is simulated using *ModelSim*

running on *Linux*. After testing the output results, *Synopsys Design Compiler* is used along with *Oklahoma State University IBM* 65*nm* cell library for logic syntheses. Delay, Timing&Area, Gate Nestlist and Constrain files are generated. After that *IC Compiler* is used to get the chip layout.

Table 8.5 is are Area, Delay comparison of all squaring units implemented in *ASIC*. *IBM* 65*nm* technology. Table 8.6, gives a comparison of truncated squaring units with the length of input is 8, 16, 32 respectively. Table 8.7 and 8.8 are results of cubing and truncated cubing units implemented in ASIC.

8.2 Summary and Future Work

This work starts with a review of basic squaring and cubing architectures. Both unsigned and signed units were discussed with algorithmically analysis. The trade-offs between delay and area is presented. After that, structures of squaring and cubing are introduced, which offers a broad insight in how to build them.

This dissertation proposed a new architecture of both squaring and cubing units. For unsigned squaring units, *PPM* was optimized more deeper, so the number of partial product bits and the height of this matrix was reduced more compared with previous methods. For singed squaring units, *Booth* 2 *Folding*, *Booth* 2 *Left-to-Right Recoding* and *Divide* & *Conquer* techniques are utilized to get a comparison for each method. For cubing unit, a new architecture was proposed by dividing the larger input to smaller pieces. This actually is the combination of previous methods to reduce the height and number of partial product bits in *PPM* to improve its performance.

The truncated units with the *CCT* technique is applied to reduce the area and power consumption on both squaring and cubing units. For each type of these units, dissertation shows how to calculate the correction constant C. Errors are generated for comparison to determine how many bits we want to kept. Diagram for this work

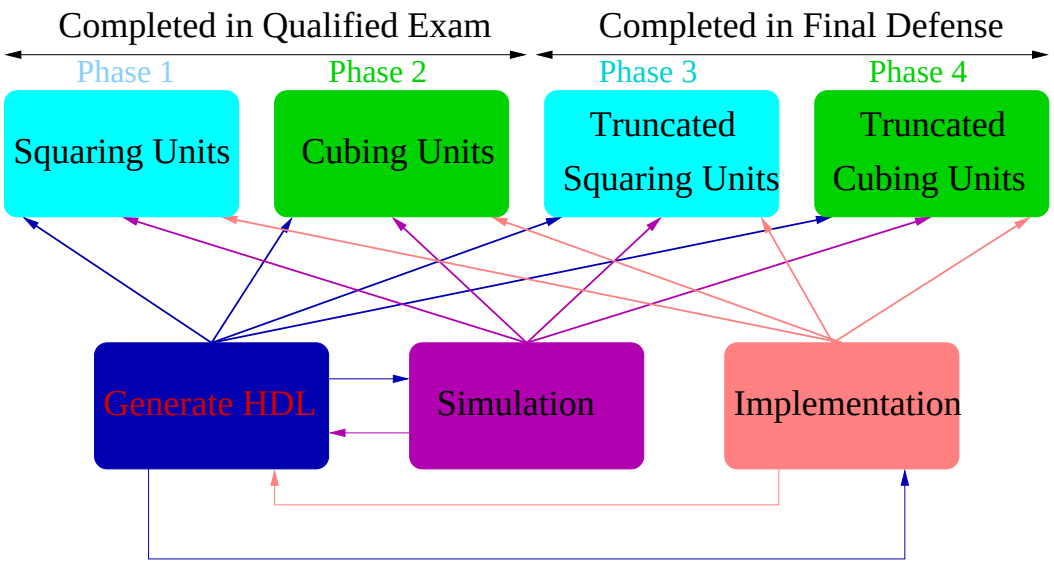

Figure 8.3: Research Flow Diagram

is followed as in Figure 8.3

Area and Delay of squaring and cubing units are formulated by using *Linear-Delay* model. From that, general idea about performance of each unit can be viewed. These units then are implemented in both techniques $FPGA\ Xilinx$ and $ASIC\ IBM\ 65nm$.

Because of time limitations, just the CCT method is applied for truncated units, and just proposed truncated cubing unit is generated. In the future, the VCT, HCT will be applied and all types of truncated cubing units will be generated and implemented to get comparison between them.

Table 8.1: Area, Delay, and Power Comparison of Squaring Units on Virtex 5 FPGA

n	Method	Area(Slices)	Delay(ns)	Power(mW)
8	Folded [66]	29	6.532	3.083
	Merged [69]	34	6.903	3.083
	Divide & Conquer [71]	44	7.079	3.083
	Booth 2 Folding [56]	34	6.594	3.083
	Booth 2 Dual Recoding [74]	37	6.301	3.082
	Proposed [77]	28	5.932	3.082
16	Folded [66]	178	9.037	3.786
	Merged [69]	183	8.943	3.790
	Divide & Conquer [71]	215	9.738	3.775
	Booth 2 Folding [56]	152	9.670	3.752
	Booth 2 Dual Recoding [74]	154	9.316	3.740
	Proposed [77]	166	8.591	3.782
32	Folded [66]	871	11.205	5.166
	Merged [69]	892	10.439	5.163
	Divide & Conquer [71]	941	11.981	5.071
	Booth 2 Folding [56]	644	11.013	5.166
	Booth 2 Dual Recoding [74]	633	10.816	5.156
	Proposed [77]	786	10.370	5.156

Table 8.2: Performances Truncated Squaring Units on FPGA Virtex 5.

n	k	Method	Area(Slices)	Delay(ns)	Power(mW)
8	2	Divide & Conquer [71]	40	7.116	2.810
		Strollo [56]	32	6.746	2.810
		Matula [74]	26	6.512	2.806
		Proposed [77]	24	6.128	2.808
	4	Divide & Conquer [71]	45	7.074	2.809
		Strollo [56]	36	6.645	2.809
		Matula [74]	34	7.076	2.810
		Proposed [77]	26	6.182	2.808
16	2	Divide & Conquer [71]	154	9.737	3.152
		Strollo [56]	118	8.901	3.172
		Matula [74]	101	8.936	3.163
		Proposed [77]	114	8.015	3.162
	4	Divide & Conquer [71]	175	9.163	3.157
		Strollo [56]	134	9.018	3.173
		Matula [74]	116	9.005	3.162
		Proposed [77]	133	8.591	3.163
32	2	Divide & Conquer [71]	661	11.978	3.760
		Strollo [56]	424	10.540	3.823
		Matula [74]	378	10.423	3.818
		Proposed [77]	468	10.168	3.817
	4	Divide & Conquer [71]	663	11.979	3.765
		Strollo [56]	461	10.486	3.841
		Matula [74]	413	10.504	3.821
		Proposed [77]	511	10.187	3.816

Table 8.3: Gates, Delay, and Power Comparison of Cubing Units on FPGA Virtex 5

n	*Method*	Area(Slices)	Delay(ns)	Power(mW)
4	Liddicoat/Flynn [67]	18	4.786	2.915
	Stine/Blank [25]	17	4.782	2.916
	Divide and Conquer [75]	17	4.682	2.916
	Proposed [76]	17	4.678	2.916
8	Liddicoat/Flynn [67]	243	10.506	3.452
	Stine/Blank [25]	339	11.002	3.484
	Divide and Conquer [75]	199	9.926	3.448
	Proposed [76]	170	10.191	3.436
16	Liddicoat/Flynn [67]	1,618	14.633	4.447
	Stine/Blank [25]	2,704	14.077	4.940
	Divide and Conquer [75]	855	14.835	4.433
	Proposed [76]	1,333	13.813	4.386
32	Liddicoat/Flynn [67]	12,674	17.171	6.613
	Stine/Blank [25]	21,437	17.662	10.418
	Divide and Conquer [75]	3,960	18.871	6.496
	Proposed [76]	9,161	16.442	6.435

Table 8.4: Comparison for Truncated Cubing Units in Virtex 5 FPGA

n	k	Area (Slices)	Delay (ns)	Power (mW)
8	4	96	10.385	2.779
	8	142	10.389	2.781
16	8	680	13.381	3.124
	16	1,128	14.288	3.127
32	16	4,670	16.417	3.727
	32	7,161	16.758	3.730

Table 8.5: Area, Delay, and Power Comparison of Squaring Units on $ASIC\ IBM\ 65nm$

n	Method	Num. Cells	Area(μm^2)	Delay(ns)	Power(mW)
8	Folded	218	1,261.44	0.564	1.465×10^{-3}
	Merged	194	1,154.40	0.585	1.337×10^{-3}
	Divide & Conquer	259	1,434.73	0.615	1.670×10^{-3}
	Booth 2 Folding	219	1,323.36	0.597	1.723×10^{-3}
	Booth 2 Dual Recoding	236	1,221.60	0.606	1.513×10^{-3}
	Proposed	191	952.80	0.517	1.127×10^{-3}
16	Folded	986	5,258.40	0.895	6.190×10^{-3}
	Merged	982	5,148.00	0.903	5.976×10^{-3}
	Divide & Conquer	1,008	5,708.16	1.009	6.400×10^{-3}
	Booth 2 Folding	835	5,018.40	0.883	6.073×10^{-3}
	Booth 2 Dual Recoding	864	5,282.40	0.889	6.766×10^{-3}
	Proposed	880	4,675.20	0.868	5.631×10^{-3}
32	Folded	3,548	19,956.00	1.281	0.0245
	Merged	3,728	21,161.28	1.257	0.0256
	Divide & Conquer	4,172	23,258.40	1.449	0.0280
	Booth 2 Folding	3,237	19,679.04	1.285	0.0236
	Booth 2 Dual Recoding	3,016	15,535.68	1.293	0.0202
	Proposed	3,560	19,392.96	1.254	0.0238

Table 8.6: Performances Truncated Squaring Units on ASIC IBM 65nm.

n	k	Method	Num. Cells	Area(μm^2)	Delay(ns)	Power(mW)
8	2	Divide & Conquer	193	1,135.20	0.605	1.330*10
		Strollo	195	1,110.24	0.591	1.424*10
		Matula	195	1,015.20	0.587	1.247*10
		Proposed	163	737.76	0.544	0.898*10
	4	Divide & Conquer	200	1,060.80	0.641	1.209*10
		Strollo	220	1,199.04	0.639	1.576*10
		Matula	207	1,189.92	0.598	1.511*10
		Proposed	164	957.60	0.542	1.170*10
16	2	Divide & Conquer	773	4,659.36	0.961	5.449*10
		Strollo	672	4,183.20	0.860	5.645*10
		Matula	540	3,324.00	0.865	4.322*10
		Proposed	673	3,696.96	0.838	4.447*10
	4	Divide & Conquer	791	4,602.72	0.957	5.321*10
		Strollo	732	4,121.76	0.919	5.518*10
		Matula	629	3,963.84	0.861	5.311*10
		Proposed	757	4,083.36	0.846	4.915*10
32	2	Divide & Conquer	2,887	16,667.04	1.294	0.0200
		Strollo	2,157	11,953.44	1.140	0.0160
		Matula	2,017	11,063.52	1.136	0.0145
		Proposed	2,267	12,189.60	1.173	0.0148
	4	Divide & Conquer	2,876	16,197.12	1.325	0.0195
		Strollo	2,222	12,887.52	1.187	0.0175
		Matula	2,187	12,456.48	1.152	0.0165
		Proposed	2,411	13,732.80	1.215	0.0169

Table 8.7: Gates, Delay, and Power Comparison of Cubing Units in ASIC IBM $65nm$

n	Method	Num. Cells	Area(μm^2)	Delay(ns)	Power(mW)
8	Liddicoat [67]	1,044	9,454.08	1.146	0.0109
	Stine [25]	1,267	11,109.12	1.111	0.0125
	Divide & Conquer [75]	862	6,202.08	1.136	7.222×10^{-3}
	Proposed [76]	976	7,233.60	1.147	8.217×10^{-3}
16	Liddicoat [67]	5,705	45,445.44	1.791	0.0513
	Stine [25]	8,639	67,039.20	1.828	0.0734
	Divide & Conquer [75]	4,127	23,978.88	2.022	0.0280
	Proposed [76]	5,298	39,013.92	1.770	0.0447
32	Liddicoat [67]	33,257	220,509.60	3.141	0.2398
	Stine [25]	56,729	348,901.44	2.989	0.3588
	Divide & Conquer [75]	17,832	102,316.32	2.903	0.1211
	Proposed [76]	34,438	219,468.96	2.874	0.2377

Table 8.8: Comparison of Truncated Cubing Units on ASIC

n	k	Num. Cells	Area (μm^2)	Delay (ns)	Power (mW)
8	4	511	2,952.96	1.148	3.086×10^{-3}
	8	845	4,968.48	1.245	5.465×10^{-3}
16	8	3,254	18,964.32	1.862	0.0203
	16	5,206	32,597.92	1.990	0.0341
32	16	20,704	103,946.40	2.954	0.1063
	32	32,566	158,580.48	3.320	0.1589

REFERENCES

[1] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. USA: Addison-Wesley Publishing Company, 4th ed., 2010.

[2] L. F. Shaw, "Arithmetic operations in binary computer," *Rev. Sci. Instrum*, pp. 687–693, 1950.

[3] A. D. Booth, "Signed binary multiplication technique," *Birkbeck Coolege Electronic Computer Projet , 21 Torrington Square, London, W.C1*, 1950.

[4] C. S. Wallace, "A suggestion for a fast multiplier," *Computers, IEEE Transactions on*, vol. EC-13, pp. 14–17, 1964.

[5] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza 34*, pp. 349–365, 1965.

[6] L. Dadda, "On parallel digital multipliers," *Alta Frequenza 45*, pp. 574–580, 1976.

[7] J. T. Yoo, K. F. Smith, and G. Gopalakrishnan, "A fast parallel squarer based on divide-and-conquer," *Solid-State Circuits, IEEE Journal of*, vol. 32, no. 6, pp. 909–912, 1997.

[8] Y. Fengqi and A. N. Willson, "Multirate digital squarer architectures," in *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*, vol. 1, pp. 177–180 vol.1, 2001.

[9] G. Kempa and P. Jung, *FPGA based logic synthesis of squarers using VHDL*, vol. 705 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1993.

[10] A. Strollo, E. Napoli, and D. De Caro, "New design of squarer circuits using booth encoding and folding techniques," in *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*, vol. 1, pp. 193–196 vol.1, 2001.

[11] A. Strollo and D. De Caro, "Booth folding encoding for high performance squarer circuits," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 50, pp. 250–254, May 2003.

[12] J. A. Pineiro, M. D. Ercegovac, and J. D. Bruguera, "Algorithm and architecture for logarithm, exponential, and powering computation," *Computers, IEEE Transactions on*, vol. 53, pp. 1085 – 1096, sept. 2004.

[13] A. Eshraghi, T. S. Fiez, K. D. Winters, and T. R. Fischer, "Design of a new squaring function for the viterbi algorithm," *Solid-State Circuits, IEEE Journal of*, vol. 29, no. 9, pp. 1102–1107, 1994.

[14] A. A. Hiasat and H. S. Abdel-Aty-Zohdy, "Combinational logic approach for implementing an improved approximate squaring function," *Solid-State Circuits, IEEE Journal of*, vol. 34, no. 2, pp. 236–240, 1999.

[15] J. A. Pineiro, J. D. Bruguera, and J. M. Muller, "Faithful powering computation using table look-up and a fused accumulation tree," in *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pp. 40 –47, 2001.

[16] E. G. Walters III and M. J. Schulte, "Efficient function approximation using truncated multipliers and squarers," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 232–239, 2005.

[17] M. Sadeghian and J. E. Stine, "Optimized low-power elementary function approximation for chebyshev series approximations," in *Signals, Systems and Com-*

puters (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on, pp. 1005–1009, 2012.

[18] D. Harris, "A powering unit for an opengl lighting engine," in *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, vol. 2, pp. 1641 –1645 vol.2, 2001.

[19] D. Harris, "An exponentiation unit for an opengl lighting engine," *Computers, IEEE Transactions on*, vol. 53, pp. 251 – 258, march 2004.

[20] M. D. Ercegovac, L. Imbert, D. W. Matula, J. M. Muller, and G. Wei, "Improving goldschmidt division, square root, and square root reciprocal," *Computers, IEEE Transactions on*, vol. 49, pp. 759 –763, jul 2000.

[21] M. J. Schulte, J. E. Stine, and J. G. Jansen, "Reduced power dissipation through truncated multiplication," *Low-Power Design, 1999. Proceedings. IEEE Alessandro Volta Memorial Workshop on*, pp. 61–69, Mar 1999.

[22] J. A. Pineiro and J. D. Bruguera, "High-speed double-precision computation of reciprocal, division, square root, and inverse square root," *Computers, IEEE Transactions on*, vol. 51, pp. 1377 – 1388, dec 2002.

[23] F. de Dinechin and A. Tisserand, "Multipartite table methods," *Computers, IEEE Transactions on*, vol. 54, pp. 319 – 330, march 2005.

[24] P. Lapsley, J. Bier, E. Lee, and A. Shoham, *DSP Processor Fundamentals: Architectures and Features*. Wiley-IEEE Press, 1st ed., 1997.

[25] J. E. Stine and J. M. Blank, "Partial product reduction for parallel cubing," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, (Washington, DC, USA), pp. 337–342, IEEE Computer Society, 2007.

[26] K. C. Bickerstaff, E. Swartzlander, Jr., and M. J. Schulte, "Analysis of column compression multipliers," in *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pp. 33 –39, 2001.

[27] C. J. Nicol and P. Larsson, "Low power multiplication for fir filters," in *Low Power Electronics and Design, 1997. Proceedings., 1997 International Symposium on*, pp. 76 – 79, Aug. 1997.

[28] J. E. Stine and O. M. Duverne, "Variations on truncated multiplication," *Digital Systems Design, Euromicro Symposium on*, vol. 0, p. 112, 2003.

[29] E. de Angel and E. E. Swartzlander, Jr., "Low power parallel multipliers," in *VLSI Signal Processing, IX, 1996., [Workshop on]*, pp. 199 –208, Oct 1996.

[30] M. J. Flynn and S. S. Oberman, *Advanced Computer Arithmetic Design*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[31] J. E. Stine, *Digital Computer Arithmetic Datapath Design Using Verilog Hdl (International Series in Operations Research"and Management Science)*. Norwell, MA, USA: Kluwer Academic Publishers, 2004.

[32] B. E. L., *Digital computer design*. New York: Academic Press, 1963.

[33] K. A. C. Bickerstaff, M. J. Schulte, and E. E. Swartzlander, Jr., "Parallel reduced area multipliers," *J. VLSI Signal Process. Syst.*, vol. 9, pp. 181–191, April 1995.

[34] A. Habibi and P. A. Wintz, "Fast multipliers," *Computers, IEEE Transactions on*, vol. C-19, pp. 153 – 157, Feb 1970.

[35] K. A. C. Bickerstaff, M. J. Schulte, and E. E. Swartzlander, Jr., "Reduced area multipliers," in *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, pp. 478 –489, Oct. 1993.

[36] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, vol. 22, pp. 1045–1047, Dec. 1973.

[37] O. L. Macsorley, "High-speed arithmetic in binary computers," *Proceedings of the IRE*, vol. 49, pp. 67–91, Jan 1961.

[38] G. W. Bewick, *Fast multiplication: algorithms and implementation.* PhD thesis, Stanford University, 1994.

[39] J. E. Stine, S. Gavai, and S. A., "Revisiting redundant booth with bias multipliers," Aug 2014.

[40] Y. C. Lim, "Single-precision multiplier with reduced circuit complexity for signal processing applications," *Computers, IEEE Transactions on*, vol. 41, pp. 1333–1336, Oct 1992.

[41] M. J. Schulte and E. E. Swartzlander, Jr., "Truncated multiplication with correction constant [for dsp]," in *VLSI Signal Processing, VI, 1993., [Workshop on]*, pp. 388 –396, oct 1993.

[42] S. S. Kidambi, F. El-Guibaly, and A. Antoniou, "Area-efficient multipliers for digital signal processing applications," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 43, pp. 90–95, Feb 1996.

[43] J. M. Jou, S. R. Kuang, and R. D. Chen, "Design of low-error fixed-width multipliers for dsp applications," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 46, pp. 836–842, Jun 1999.

[44] E. J. King and E. E. Swartzlander, Jr., "Data-dependent truncation scheme for parallel multipliers," in *Signals, Systems Computers, 1997. Conference Record of the Thirty-First Asilomar Conference on*, vol. 2, pp. 1178 –1182 vol.2, nov 1997.

[45] E. E. Swartzlander, Jr., "Truncated multiplication with approximate rounding," *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, vol. 2, pp. 1480 –1483 vol.2, 1999.

[46] M. J. Schulte, K. E. Wires, and J. E. Stine, "Variable-correction truncated floating point multipliers," in *in Proceedings of the Thirty Fourth Asilomar Conference on Signals, Circuits and Systems*, pp. 1344–1348, 2000.

[47] S. J. Jou and H. H. Wang, "Fixed-width multiplier for dsp application," *Computer Design, International Conference on*, vol. 0, p. 318, 2000.

[48] S. J. Jou, M. H. Tsai, and Y. L. Tsao, "Low-error reduced-width booth multipliers for dsp applications," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 50, pp. 1470–1474, Nov. 2003.

[49] L. D. Van, S. S. Wang, and W. S. Feng, "Design of the lower error fixed-width multiplier and its application," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 47, pp. 1112–1118, Oct 2000.

[50] L. D. Van and C. C. Yang, "Generalized low-error area-efficient fixed-width multipliers," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 52, pp. 1608–1619, Aug. 2005.

[51] Y. C. Liao, H. C. Chang, and C. W. Liu, "Carry estimation for two's complement fixed-width multipliers," in *Signal Processing Systems Design and Implementation, 2006. SIPS '06. IEEE Workshop on*, pp. 345 –350, oct. 2006.

[52] K. J. Cho, K. C. Lee, J. G. Chung, and K. K. Parhi, "Design of low-error fixed-width modified booth multiplier," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, pp. 522–531, May 2004.

[53] T. B. Juang and S. F. Hsiao, "Low-error carry-free fixed-width multipliers with low-cost compensation circuits," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 52, pp. 299–303, June 2005.

[54] T. B. Juang, C. C. Wei, and C. H. Chang, "Area-saving technique for low-error redundant binary fixed-width multiplier implementation," in *Integrated Circuits, ISIC '09. Proceedings of the 2009 12th International Symposium on*, pp. 550 – 553, Dec 2009.

[55] H. Park and E. E. Swartzlander, Jr., "Truncated multiplication with symmetric correction," *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, pp. 931–934, 29 2006-Nov. 1 2006.

[56] A. G. M. Strollo, N. Petra, and D. DeCaro, "Dual-tree error compensation for high performance fixed-width multipliers," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 52, pp. 501 – 507, aug. 2005.

[57] N. Petra, D. De Caro, and A. G. M. Strollo, "Design of fixed-width multipliers with minimum mean square error," in *Circuit Theory and Design, 2007. ECCTD 2007. 18th European Conference on*, pp. 464 –467, aug. 2007.

[58] V. Garofalo, N. Petra, D. De Caro, A. G. M. Strollo, and E. Napoli, "Low error truncated multipliers for dsp applications," in *Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on*, pp. 29 –32, aug. 2008.

[59] I. C. Wey and C. C. Wang, "Low-error and area-efficient fixed-width multiplier by using minor input correction vector," in *Electronics and Information Engineering (ICEIE), 2010 International Conference On*, vol. 1, pp. V1–118 –V1–122, aug. 2010.

[60] N. Petra, D. De Caro, A. G. M. Strollo, V. Garofalo, E. Napoli, M. Coppola, and P. Todisco, "Fixed-width csd multipliers with minimum mean square error," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pp. 4149 –4152, may. 2010.

[61] N. Petra, D. De Caro, V. Garofalo, E. Napoli, and A. G. M. Strollo, "Truncated binary multipliers with variable correction and minimum mean square error," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 57, pp. 1312 –1325, jun. 2010.

[62] S. R. Kuang and J. P. Wang, "Low-error configurable truncated multipliers for multiply-accumulate applications," *Electronics Letters*, vol. 42, no. 16, pp. 904–905, 2006.

[63] R. Michard, A. Tisserand, and N. Veyrat-Charvillon, "Carry prediction and selection for truncated multiplication," *Signal Processing Systems Design and Implementation, 2006. SIPS '06. IEEE Workshop on*, pp. 339–344, Oct. 2006.

[64] L. Dadda, "Squarers for binary numbers in serial form," in *Computer Arithmetic (ARITH), 1985 IEEE 7th Symposium on*, pp. 173–179, 1985.

[65] A. Deshpande and J. Draper, "Comparing squaring and cubing units with multipliers," in *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pp. 466–469, 2012.

[66] T. C. Chen, "A binary multiplication scheme based on squaring," *Computers, IEEE Transactions on*, vol. C-20, pp. 678 – 680, june 1971.

[67] A. A. Liddicoat and M. J. Flynn, "Parallel square and cube computations," in *Signals, Systems and Computers, 2000. Conference Record of the Thirty-Fourth Asilomar Conference on*, vol. 2, pp. 1325 –1329 vol.2, 2000.

[68] K. E. Wires, M. J. Schulte, L. P. Marquette, and P. I. Balzola, "Combined unsigned and two's complement squarers," in *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, vol. 2, pp. 1215 –1219 vol.2, 1999.

[69] R. H. Strandberg, L. G. Bustamante, V. G. Oklobdzija, M. A. Soderstrand, and J. C. Le Duc, "Efficient realizations of squaring circuit and reciprocal used in adaptive sample rate notch filters," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 14, no. 3, pp. 303–309, 1996.

[70] R. K. Kolagotla, W. R. Griesbach, and H. R. Srinivas, "Vlsi implementation of 350 mhz 0.35 mu;m 8 bit merged squarer," *Electronics Letters*, vol. 34, no. 1, pp. 47–48, 1998.

[71] H. Thapliyal, S. Kotiyal, and M. B. Srinivas, "Design and analysis of a novel parallel square and cube architecture based on ancient indian vedic mathematics," in *Circuits and Systems, 2005. 48th Midwest Symposium on*, pp. 1462 – 1465 Vol. 2, aug. 2005.

[72] V. Kunchigi, L. Kulkarni, and S. Kulkarni, "Low power square and cube architectures using vedic sutras," in *Signal and Image Processing (ICSIP), 2014 Fifth International Conference on*, pp. 354–358, Jan 2014.

[73] S. Datla, M. Thornton, and D. Matula, "A low power high performance radix-4 approximate squaring circuit," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pp. 91–97, July 2009.

[74] D. Matula, "Higher radix squaring operations employing left-to-right dual recoding," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pp. 39–47, June 2009.

[75] M. Ramalatha, K. Thanushkodi, D. K. Deena, and P. Dharani, "A novel time and energy efficient cubing circuit using vedic mathematics for finite field arithmetic," *Advances in Recent Technologies in Communication and Computing, International Conference on*, vol. 0, pp. 873–875, 2009.

[76] S. Bui, J. E. Stine, and M. Sadeghian, "Approaches for high speed parallel cubing unit," in *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*, Jul 2014.

[77] S. Bui and J. E. Stine, "Additional optimizations for parallel squarer units," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, (Melbourne, Australia), IEEE International Symposium on circuits and Systems, Jun 2014.

[78] W. J. Townsend, E. Swartzlander, Jr., and J. A. Abraham, "A comparison of dadda and wallace multipliers delays," *Proc. SPICE, Advanced Signal Processing Algorithms, Architectures, and Implementation XIII*, pp. 552 –560, 2003.

[79] L. Rubinfield, "A proof of the modified booth's algorithm for multiplication," *Computers, IEEE Transactions on*, vol. C-24, pp. 1014–1015, Oct 1975.

[80] P. E. Blankenship, "Comments on "a two's complement parallel array multiplication algorithm"," *IEEE Trans. Comput.*, vol. C-23, p. 1327, 1974.

[81] E. E. Swartzlander, Jr., *Computer Arithmetic*, vol. I, II. Dowden, Hutchinson and Ross, Inc., 1990.

[82] P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," in *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*, pp. 232 –236, jun 1991.

[83] J. Pihl and E. J. Aas, "A multiplier and squarer generator for high performance dsp applications," in *Circuits and Systems, 1996., IEEE 39th Midwest symposium on*, vol. 1, pp. 109–112 vol.1, 1996.

[84] N. Takagi, "Powering by a table look-up and a multiplication with operand modification," *Computers, IEEE Transactions on*, vol. 47, pp. 1216 –1222, nov 1998.

[85] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *Computers, IEEE Transactions on*, vol. 48, pp. 842 –847, aug 1999.

[86] M. D. Ercegovac, T. Lang, J. M. Muller, and A. Tisserand, "Reciprocation, square root, inverse square root, and some elementary functions using small multipliers," *IEEE Trans. Comput.*, vol. 49, pp. 628–637, July 2000.

[87] K. E. Wires, M. J. Schulte, and J. E. Stine, "Combined ieee compliant and truncated floating point multipliers for reduced power dissipation," *Computer Design, International Conference on*, p. 0497, 2001.

[88] K. J. Cho, K. C. Lee, J. G. Chung, and K. K. Parhi, "Low error fixed-width modified booth multiplier," *Signal Processing Systems, 2002. (SIPS '02). IEEE Workshop on*, pp. 45–50, Oct. 2002.

[89] D. De Caro and A. G. M. Strollo, "Parallel squarer using booth-folding technique," *Electronics Letters*, vol. 37, pp. 346–347(1), March 2001.

[90] J. S. Wang, C. N. Kuo, and T. H. Yang, "Low-power fixed-width array multipliers," in *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, (New York, NY, USA), pp. 307–312, ACM, 2004.

[91] A. Cosoroaba and F. Rivoallon, "Archieving higher system performance with the virtex-5 family of fpgas," *Xilinx WP245 V*, vol. 1, 2006.

[92] Q. Zhang, R. Eagleson, and T. M. Peters, "Gpu-based image manipulation and enhancement techniques for dynamic volumetric medical image visualization," in *Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on*, pp. 1168 –1171, april 2007.

[93] M. Gok, "Enhanced-functionality multipliers," *Journal of Systems Architecture*, vol. 54, no. 8, pp. 742 – 756, 2008.

[94] P. Zicari and S. Perri, "A fast carry chain adder for virtex-5 fpgas," in *MELECON 2010 - 2010 15th IEEE Mediterranean Electrotechnical Conference*, pp. 304 –308, april 2010.

[95] S. Jalaja and A. Prakash, "High speed vlsi architecture for squaring algorithm using retiming approach," in *Advances in Computing and Communications (ICACC), 2013 Third International Conference on*, pp. 233–238, Aug 2013.

VITA

SON VIET BUI

Candidate for the Degree of

Doctor of Philosophy

Dissertation: **ADAPTIVE AND HYBRID SCHEMES FOR EFFICIENT PARALLEL SQUARING AND CUBING UNITS**

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Thai Binh City, Thai Binh Province, Vietnam on June 27, 1977.

Education:
Received the B.S. degree from Hanoi University of Science and Technology, Hanoi, Vietnam, 2000, in Electrical Engineering
Received the M.S. degree from Hanoi University of Science and Technology, Hanoi, Vietnam, 2002, in Electrical Engineering
Completed the requirements for the degree of Doctor of Philosophy with a major in Electrical Engineering Oklahoma State University in December, 2014.