CUBING UNITS USING CARRY-SAVE ARRAY IMPLEMENTATIONS

By

VRUSHALI ANIL YEWLE

Bachelor of Engineering in Electronics and

Telecommunication

University of Mumbai

Maharashtra, India

2006

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2009

CUBING UNIT USING CARRY-SAVE ARRAY

IMPLEMENTATIONS

Thesis Approved:

Dr. James E. Stine, Jr.

Thesis Adviser

Dr. Louis Johnson

Dr. Sohum Sohoni

Dr. A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank my adviser Dr. James E. Stine, Jr. for his outstanding support and encouragement. I have learnt more than just Digital VLSI Design from him. Dr. Stine, you are a real inspiration for me.

I would also like to express my gratitude to Dr. Louis Johnson and Dr. Sohum Sohoni whose classes on Digital Computer Design and Computer Architecture have tremendously helped me in understanding the higher levels of abstraction of computer design while principally working at the lower levels of VLSI designs as a design engineer.

To my parents, Anil and Sulabha Yewle who always believed in me and unconditionally supported me in all my academic and non-academic endeavors.

To my sisters Bhagyashree and Resham and my friends Emmanuel, Ritesh, Sweety and Neha who've been sweet and understanding. Thank you for keeping me sane and grounded.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Research in the field of computer architecture shows that the trend in super scalar processors is towards wider and more diverse pipelines, employing more specialized functional units for important operations. One such critical operation is that of *Cubing.* *Cubing* can be defined as an operation involving the multiplication of three numbers. For example, $a \times b \times c$. A number can also be multiplied with itself thrice. For example, a^3. Cubing is a crucial operation is because it is an important step in many algorithms.

1.1Previous Research Work

The Goldschmidt's algorithm [1] is used to perform division, calculate the reciprocal and the reciprocal square root in the arithmetic logic units of many processors. An essential step during the process involves the calculation of the product: $a \times b \times c$. In graphics, algorithms used in shading [2], etc. also require a similar operation. Prior research in this area includes the simulation and error analysis of the arithmetic cube optimized for DFT and convolution applications [13] and an effort to develop better algorithms for computing the cube on very large compresses data sets[14].

In most processors today, cubing is performed with the help of look-up tables that are generated based on a polynomial [3]. The problem with this method is that look-up tables consume too much memory. Memory is expensive. Besides hardware implementations are much faster than look-up table implementations. If cubing is to be implemented using hardware, current methods would require two passes through a multiplier. So, the first pass would perform the multiplication $p = a \times b$ and the second pass would perform the operation $m = p \times c$. Besides, the fact that instead of calculating the cube in a single pass through the multiplier, such a method would require two passes and hence twice the number of clock cycles, this has another disadvantage. Consider the multiplication of three *6*-bit numbers, *a, b* and *c*. The product $a \times b$ is a *12*- bit number. The multiplication of a *12*-bit number and a *6*-bit number requires the use of an irregular sized multiplier which is not always available in a processor.

1.2 Parallel Cubing Engines

This study discusses the design for a single functional unit implemented in hardware that performs cubing in a single pass. The design has been implemented based on the concept of carry-save array multipliers. Carry-save array multipliers are built using carry-save adders in which the carry-outs from one column are propagated as carry-ins to the next column. The motivation behind this comes from the fact that addition can be accelerated by delaying carry propagation till the last step. Carry-save based arithmetic architectures are widely popular in VLSI design [4]. Array structures are regular. It is easier to build larger systems more economically and efficiently in a hierarchical manner by exploiting the pattern seen in the smaller structures. Also, this facilitates the coding of a software generator that can be used to automatically generate the HDL for the corresponding length of input words to the multiplier.

1.3 Organization

This paper first takes a look at the problems of unsigned multiplication and unsigned cubing. In the real world, one will always have to work with negative numbers. There are different number systems that can be used to represent positive and negative numbers like sign magnitude representation, one's complement representation, two's complement representation, etc. The two's complement number system is the most commonly used number system that represents positive as well as negative numbers. It has its advantages like a single and hence, conflict-free representation for zero. Regular mathematical operations like addition, subtraction and multiplication work well with it. An n-bit two's complement number represents any number in the range $\{-2^{n-1}, 2^{n-1}-1\}$. Chapter *2* looks at two's complement number system and the modifications made to the unsigned multiplier to deal with signed numbers.

Also in chapter *2*, some background related to multiplication, multipliers, two's complement multipliers while chapter *3* forms the crux of this study- the problem of cubing. Chapter *3* also visits the cubing unit designed to perform the cubing operation using signed operands, called two's complement cubing. Both, chapters *2* and *3* take a look at examples that use *3*-bit operands to gain a better understanding of how the design actually works. Similar to that in chapter *2*, chapter *3* once again revisits the two's complement representation for signed numbers and the modifications applied to the unsigned version to arrive at a design that can handle the cubing of three signed numbers. Chapter *4* then looks at the algorithmic analysis of the various designs and compares them for area and delay. It also reviews the results obtained after testing and synthesis. Chapter *5* discusses the conclusions that were arrived at, post the completion of this study.

CHAPTER 2

BACKGROUND - MULTIPLICATION

Multiplication involves the use of addition in some way to produce a product ‘p’ from a multiplicand ‘*a*’ and a multiplier ‘*b*’ such that:

$$p = a \times b$$

2.1 Multipliers and their types

The hardware designed to carry out multiplication is known as a Multiplier. High speed multipliers can be divided into two categories:

1. Serial multipliers: Serial or sequential multipliers compute the product sequentially, usually utilizing storage elements so that the hardware in the multiplier is reused in each iteration. Serial multipliers have low area requirements but are slow since they calculate partial products sequentially.
2. Parallel multipliers: Parallel multiplication involves the use of hardware to multiply an m-bit number by an n-bit number to completely produce a $n + m$ bit product. Parallel multipliers can also be pipelined to reduce the cycle time and increase the throughput by introducing storage elements within the multiplier.

2.2 Steps involved in multiplication

Multiplication usually involves three separate steps as listed below. Although there are implementations that can theoretically be reduced to the generation of shifted multiples of the multiplicand and multi-operand addition (i.e. the addition of two or more operands), most multipliers utilize the steps listed below.

1. Partial Product generation
2. Partial Product reduction
3. Carry Propagate Adder (CPA)

The following dot diagram [7] further explains the process of multiplication better. The multiplier and the multiplicand are both *2*-bit words in this case.

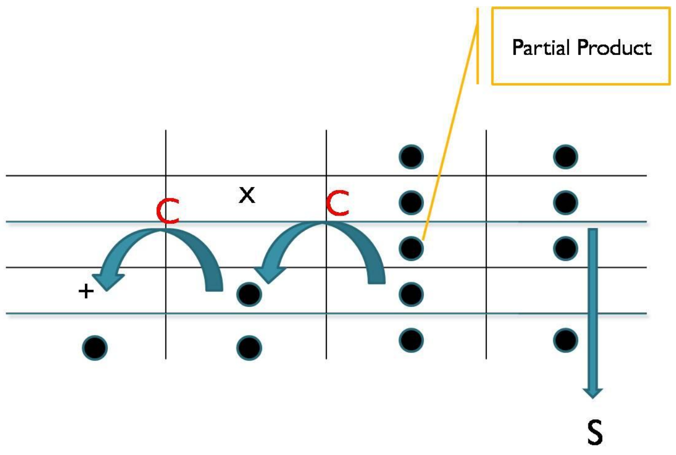


Figure *1*: The Dot Diagram for Multiplication

The process of multiplying each bit of the multiplier with each bit of the multiplicand is termed as partial product generation. The process of performing addition on a given array that produces an output array with smaller number of bits is called reduction. The process of partial product reduction produces a sum and a carry. The sum is the partial product of the corresponding column while the carry must be propagated to the next column as in any addition operation. Hence the use of the carry propagate adder in the multiplier.

2.3 The carry-save concept

When several numbers are added up sequentially, it is not necessary to propagate the carries during each addition. Instead, the carries generated during an addition can be saved as partial carries and added with the next operand during the next addition. Namely, each addition can be accelerated by postponing the carry propagation. This leads to the concept of carry save addition. The numbers can be added up by a series of carry-save additions, followed by a carry propagate addition. Namely, for multi-operand addition, only one carry-save addition is required. An adder for carry save addition is referred to as a carry save adder. A carry save adder sums up a partial sum and a partial carry from the previous stage as well as an operand and produces a new partial sum and partial carry. An n-bit carry save adder comprises of just n full adders without interconnections among them. The following Figure [7] shows a one such carry save adder or *(3, 2)* counter which is nothing but a full adder:

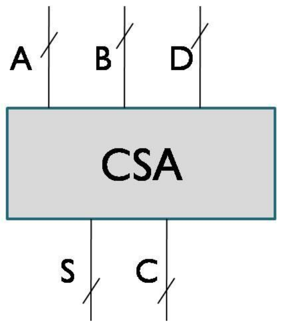


Figure *2*: Carry Save Adder

Figures *3* and *4* [7] show the half and full adder implementations at the gate level.

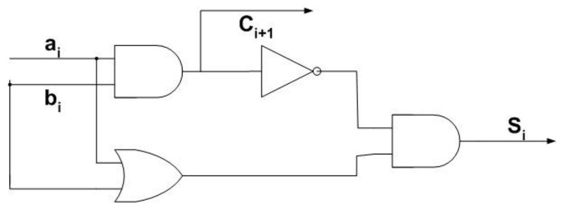


Figure *3*: Half Adder

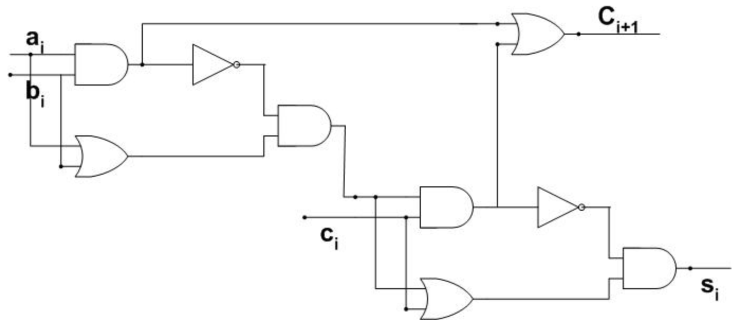


Figure *4*: Full Adder

The CSA utilizes many topologies of adder so that the carry-out from one adder is not connected to the carry-in of the next adder. This is in contrast to what is done in ripple carry addition [5]. Eventually a carry propagate adder can be utilized to compute the result. This organization of utilizing *m*-word by *n*-bit multi operand adders together to add *m*-operands or words each of which is *n*-bits long is called a multi-operand adder (MOA). An *m*-word by *n*-nit multi-operand adder can be implemented by using (*m-2*) *n*-bit CSA's and *1* CPA. Unfortunately, because the number of bits added together increases the result, the partial sum and carry must grow as well. Therefore, the result will contain $[n + \log_2 (m)]$ bits.

2.4 Carry Propagate Adder

By cascading *n* full adders, an *n* bit binary adder capable of handling two *n* bit operands can be designed. The implementation of a *4*-bit ripple carry adder or binary adder is shown in Figure *5* [7]. When two unsigned integers are added, the input carry c_0 is always zero. The *4*-bit adder is also called a carry propagate adder (CPA), because the carry is propagated serially through each full adder. $C_0 = 0$ or *1* for multiprecision addition. The design of a carry propagate adder is straight forward but the carry propagation time limits the speed of operation.

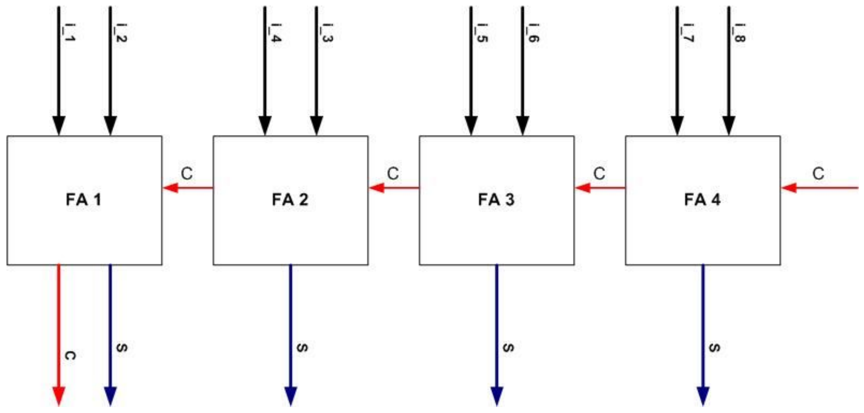


Figure *5*: Ripple Carry Adder

2.5 Carry Save Array Multipliers (CSAM)

In this section, we look at carry save multipliers – their architecture, performance and speed considerations.

2.5.1 Architecture of the CSAM

The simplest of all multipliers is the carry save array multiplier. The principle idea behind the CSAM is that it is basically doing paper and pencil style multiplication. In other words, each partial product is being added as seen in the dot diagram of Figure *1*. Figure *7* shows a *3*-bit x *3*-bit CSAM [7]. Each column of the multiplication matrix corresponds to a diagonal in the carry save array multiplier. The reason CSAMs are always done as a square is because it allows metal tracks or interconnect to have less congestion. This has a tendency to have less capacitance as well as making it easier for engineers to organize the design.

The CSAM performs partial product generation utilizing AND gates and uses an array of carry save adders to perform reduction. The AND gates form the partial products and the CSAs sum these partial products together or *reduce* them. Since most of the reduction computes the lower half of the product, the final carry propagate adder only needs to add the upper half of the product. Array multipliers are typically easy to build both using Verilog code as well as in custom layout, hence there are many implementations that employ both.

For the CSAM implementation, each adder is modified so that it can perform partial product generation and an addition. A modified half adder (MHA) comprises of an AND gate that creates the partial product bit and a half adder (HA). The MHA adds the partial product bit from the AND gate with a partial product bit from the previous row. The modified full adder (MFA) consists of an AND gate and that creates the partial product bit and a full adder (FA) that adds this partial product bit with the sum and carry bits from the previous rows.

An n-bit by m-bit CSAM has $n \cdot m$ number of AND gates, m HAs and $((n-1) \cdot (m-1))-1 = n \cdot m - n - m$ FAs. The final row of $(n-1)$ adders is a ripple carry adder CPA. Further algorithmic analysis of area and delay has been done in Chapter *4*.

This CSAM was first proposed by Braun Edward Louis in *1963*. It is restricted to performing the addition of two unsigned numbers. This is also known as a non-additive multiplier since it does not add an additional operand to the result of the multiplication.

The carry save array multiplier is a parallel multiplier. For an *m*-bit multiplicand and an n-bit multiplier, it is an m-bit by n-bit digital logic system that produces a *(m + n)* product combinationally. It has the following advantages:

1. The CSAM has a linear delay directly proportional to the length of the input word.
2. The number of gates required to implement the CSAM is directly proportional to the product n x m or n^2 for $m = n$.
3. It has regular area. This enhances the reproducibility of the design.

The following diagram further explains the above statements:

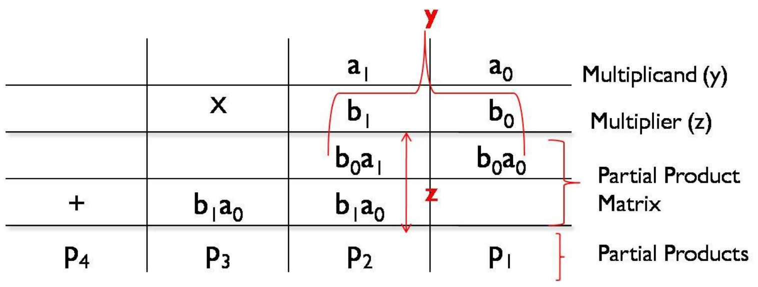


Figure *6*: Multiplication of two *2*-bit operands

When building the array multiplier to implement the operation shown in Figure *3*, the following are the main components:

1. AND gates: For partial product generation
2. Half Adders, Full Adders: For Partial product reduction.

Based on all these concepts, a rectangular array multiplier to perform the multiplication of two *3* bit numbers a[*2:0*] and b[*2:0*] is shown in Figure *7*.

Shown below in Figure *7* [7] is a *3*-bit x *3*-bit rectangular array multiplier. The acronyms MHA and MFA stand for 'modified half adder' and 'modified full adder' respectively'. The critical path is shown by the blue dashed line in the Figure.

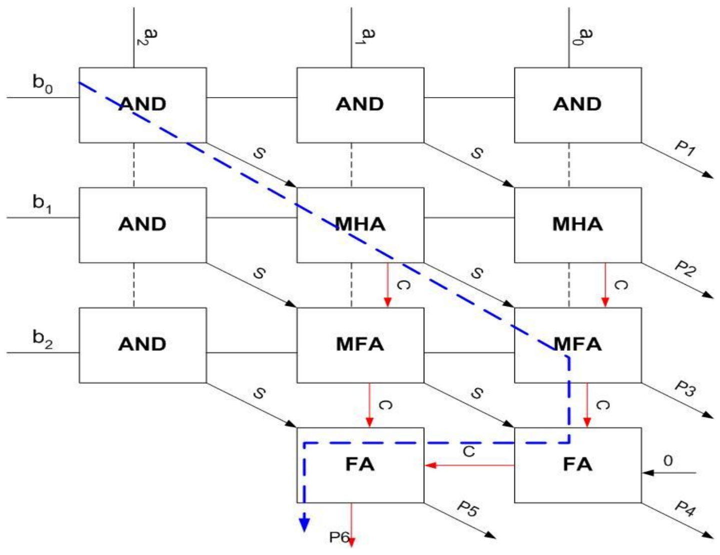


Figure *7*: a *3*-bit x *3*-bit CSAM

Figures *8* and *9* [7] show diagrammatic representations for the modified half and full adders respectively.

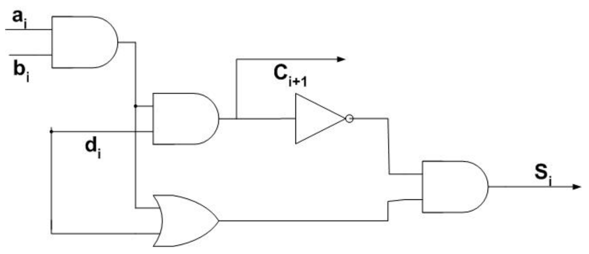


Figure *8*: Modified Half Adder

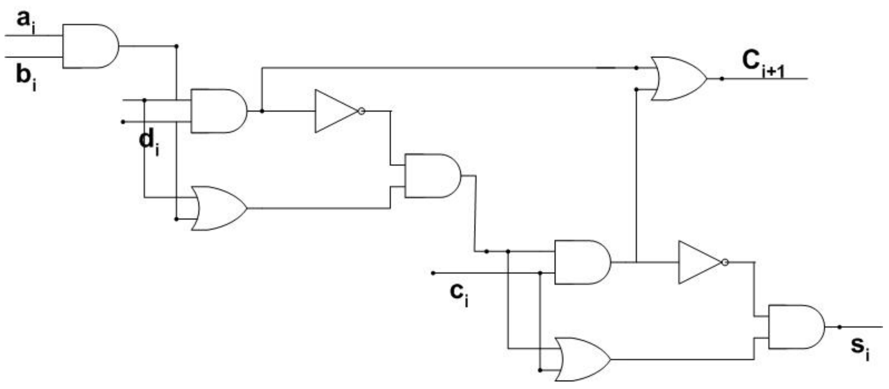


Figure *9*: Modified Full Adder

The modified half and full adders are similar to the conventional half and full adders respectively, except for the fact that the modified versions of these entities have an additional AND gate in order to calculate the product $a_i \cdot b_j$. The modified half adders and full adders have an extra AND gate when compared to the traditional half and full adder. This extra gate is required to calculate the partial products b_0a_0, b_0a_1, etc as in Figure *3*. The blue dotted line shows the critical path through the multiplier. This is the path of longest delay from the input to the output of the multiplier. The partial products are calculated along the diagonals of the array. The carries are denoted in red. They must be carefully carried over to the next column for accurate calculation of all the partial products.

2.5.2 Example

Let us further understand this by looking at the example shown in Figure *10*. It shows the multiplication of $(7)_{10}$ and $(5)_{10}$.

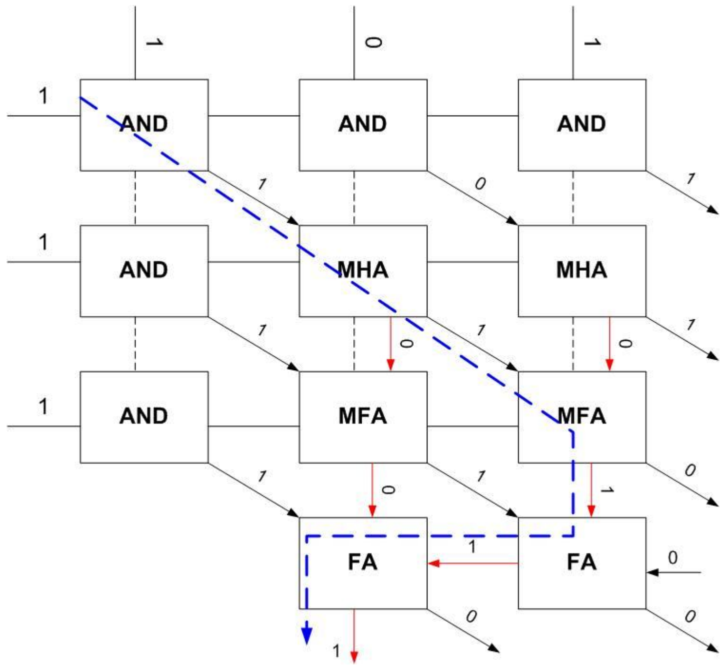


Figure *10*: Example of a *3*-bit x *3*-bit CSAM

The result is $(100011)_2 = (35)_{10}$ which is correct.

2.5.3 Performance of the CSAM or Braun multiplier

The Braun multiplier performs well for unsigned operands that are less than *16* bits, in terms of speed, power and area. Besides, it has a simple and regular structure compared to most multiplier schemes. However, the number of components required to build the multiplier increases quadratically with the number of bits. This makes this design inefficient, so it is rarely employed while dealing with large operands. Another pitfall of the Braun multiplier is its potential susceptibility to glitching problems at the last stage of full adders due to its exploitation of the carry propagate adder.

2.5.4 Speed Consideration

The delay of the CSAM is dependent on the delay of the full adder cell and also the CPA in the last row of the design. In the multiplier array, a full adder with balanced carry and sum delays is desirable because the sum and carry, both are in the critical path. The speed and power of the full adder are very important in large arrays. More about this will be discussed in Chapter *4*.

2.6. Two's Complement Version (the Baugh-Wooley multiplier)

The Baugh-Wooley multiplier [6] is an enhanced version of the carry save array multiplier (CSAM). It is designed to cater to the multiplication of both, signed and unsigned operands which are represented by the two's complement number system.

2.6.1. Architecture of the Baugh-Wooley Multiplier

The two's complement number system represents positive as well as negative numbers. They have the following format:

$$A = -a_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} a_i .2^i$$

The range of an n-bit *2*'s complement number is $\{-2^{n-1}, 2^{n-1} - 1\}$

Squaring two *2*'s complement numbers:

If two *2*'s complement numbers were multiplied together, the resulting multiplication would result in *4* terms. Let us consider the case of squaring an n-bit *2*'s complement number. We get:

$$A^2 = a_{n-1}.a_{n-1}.2^{2.n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i\,.a_j.2^{i+j} - \sum_{i=0}^{n-2} a_i\,.a_{n-1}.2^{i+n-1} - \sum_{j=0}^{n-2} a_{n-1}.a_j.2^{j+n-1}$$

The Baugh-Wooley matrix is achieved by converting the subtractions by complementing the partial products and adding a unit in the last place or ulp to convert the value to its two's complement equivalent. Therefore, the equations above now become:

$$A^2 = a_{n-1}.a_{n-1}.2^{2.n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i\,.a_j.2^{i+j} + \sum_{i=0}^{n-2} \overline{a_i.a_{n-1}}\,.2^{i+n-1} + \sum_{j=0}^{n-2} \overline{a_{n-1}.a_j}\,.2^{j+n-1} + [2^{2.n-2} + 2^{2.n-2}] + [2^{n-1} + 2^{n-1}]$$

The last constant terms *(2^{n-1})* come from the ulp that is needed after each term is complemented (.e.g $a_i \cdot a_{n-1}$). On the other hand, the constant $2^{2\cdot n-2}$ is needed since this term is originally a *0* before the term is one's complemented. These terms, both constants and partial products, can be utilized to form the product with typical array or tree multiplication hardware schemes.

The resulting Baugh-Wooley Carry save array multiplier to perform two's complement multiplication of two *3*-bit numbers is shown in the Figure.

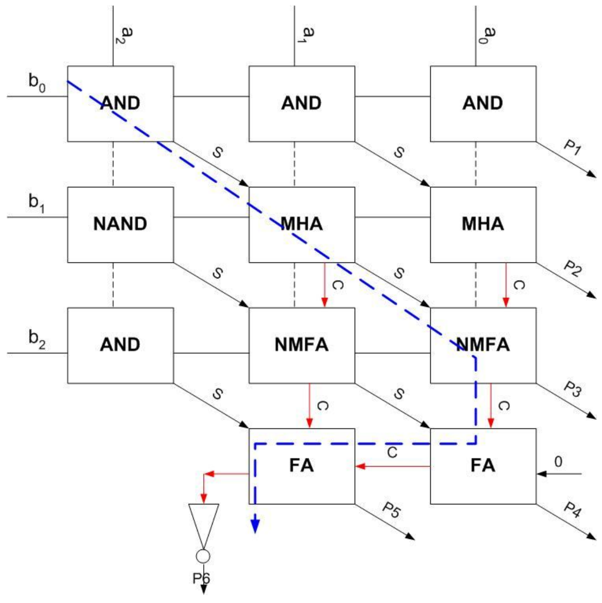


Figure *11*: *3*-bit Baugh Wooley Carry Save Array Multiplier

Some of the AND gates change to NAND gates and the modified full adders change to the NAND modified full adders (NMHAs) which is the same as the modified full adder except that the extra AND gate is replaced by a NAND gate.

The constant that must be added into the array is $2^3 + 2^5 = 40 = (101000)_2$. Hence a binary *1* is added into the *4th* and the *6th* partial products.

2.6.2. Example

Let us see if this works with an example as shown in Figure *12* below. The product of $(3)_{10}$ and $(-1)_{10}$ is $(-3)_{10} = (111101)_2$. As previously mentioned, the value $(101000)_2$ must be added into the array to get the correct result.

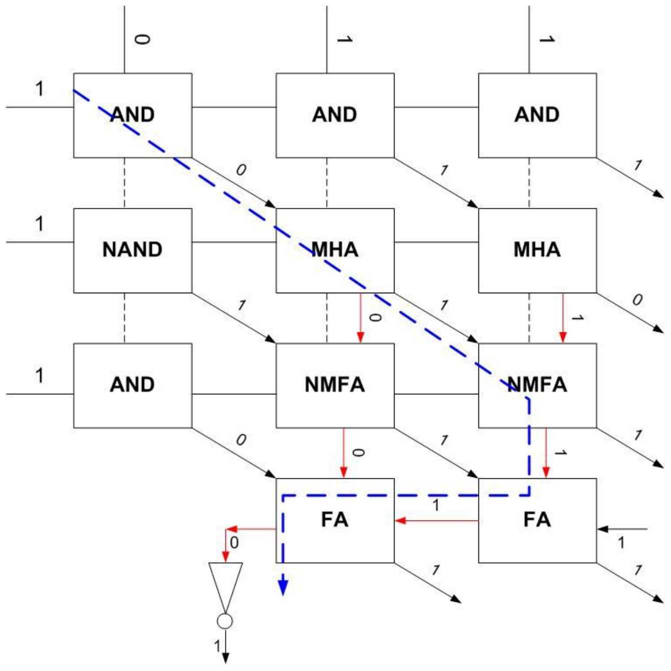


Figure *12*: Example of a *3*-bit Baugh Wooley array multiplier

The result obtained is as expected. Hence, the modification to the original Baugh-Wooley carry-save array multiplier resulted in a *2*'s complement carry-save array multiplier.

The architecture of the Baugh-Wooley multiplier is also based on the carry save array algorithm. It inherits the regular and repeating structure of the array multiplier.

2.6.3. Performance consideration

The area and power consumption of a number of multiplier structures vary with the number of bit operands and layout strategies. Increasing regularity and locality at the silicon level reduces the power consumption in a standard cell based design flow.

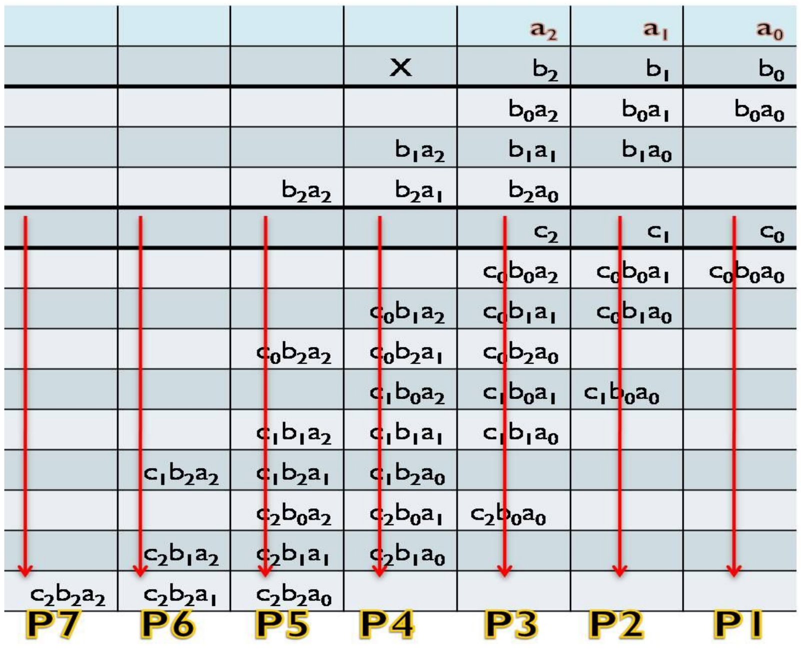
CHAPTER 3

THE PROBLEM OF CUBING

As mentioned before, the operation of cubing is calculation of the product *a x b x c* where *a*, *b* and *c* are three *n*-bit numbers. In this chapter, we look at parallel cubing, propose architectures for cubing units that deal with unsigned and two's complement numbers and discuss the components used in their implementations.

3.1. Parallel Cubing

Figure *13* shown below helps explain the concept of parallel cubing.



				a_2	a_1	a_0
			×	b_2	b_1	b_0
				b_0a_2	b_0a_1	b_0a_0
			b_1a_2	b_1a_1	b_1a_0	
		b_2a_2	b_2a_1	b_2a_0		
				c_2	c_1	c_0
				$c_0b_0a_2$	$c_0b_0a_1$	$c_0b_0a_0$
			$c_0b_1a_2$	$c_0b_1a_1$	$c_0b_1a_0$	
		$c_0b_2a_2$	$c_0b_2a_1$	$c_0b_2a_0$		
			$c_1b_0a_2$	$c_1b_0a_1$	$c_1b_0a_0$	
		$c_1b_1a_2$	$c_1b_1a_1$	$c_1b_1a_0$		
	$c_1b_2a_2$	$c_1b_2a_1$	$c_1b_2a_0$			
		$c_2b_0a_2$	$c_2b_0a_1$	$c_2b_0a_0$		
	$c_2b_1a_2$	$c_2b_1a_1$	$c_2b_1a_0$			
$c_2b_2a_2$	$c_2b_2a_1$	$c_2b_2a_0$				
P7	P6	P5	P4	P3	P2	P1

Figure *13*: Parallel Cubing

The multiplication of three *3*-bit numbers yields a *9*-bit number and hence there must be *9* partial products at the end of the operation. But Figure *7* shows only 7 partial products, from *P1* to *P7*. The two remaining partial products *P8* and *P9* are obtained after adding the carry propagate adder to the array implementation of the cubing unit. Also, as evident from Figure *7*, the partial product matrix is not rectangular. It comprises of three parallelograms. This makes it very challenging to arrive at a rectangular implementation of the cubing unit.

3.2 Architecture of the Cubing Unit

To arrive at a two dimensional array implementation of the cubing unit, first consider a *3* dimensional view of the three parallelograms comprising the partial product matrix. This is show in the Figure below:

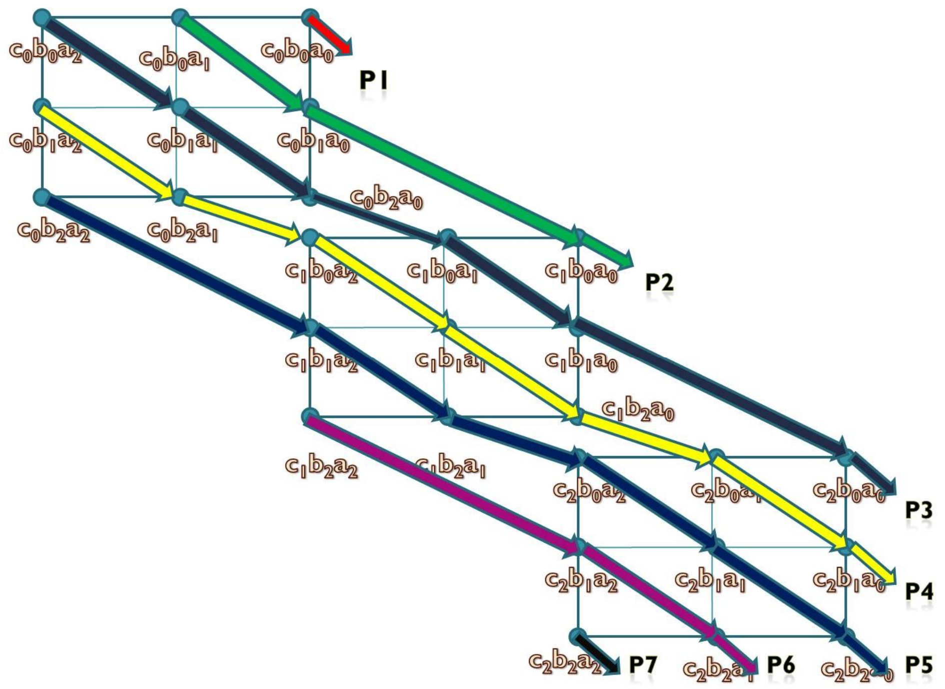


Figure *14*: Parallelograms in Space

In Figure *8*, the partial products are computed along the corresponding diagonals of the array cubing unit. The different diagonals haven been represented by different colors since this makes HDL coding of the implementation easier and the Figure, more legible. The columns traverse not only along the x and y but also the z axis. This very fact makes a rectangular array implementation of the cubing problem extremely challenging.

3.2.1. Cubing Unit Implementation

In order to save area and to have as regular a design as possible, the following design is arrived at, as shown in Figure 8. This two dimensional implementation shows that it is not necessarily area-efficient in terms of a VLSI layout. The Figure shows that a new special unit in the carry propagate unit of the implementation has been used. This new block is the *(7, 3)* adder. The *(7, 3)* adder comprises of four half adders. It takes in seven *1*-bit inputs and produces three 1-bit outputs. Care has to be taken to ensure that the carries traverse to the correct corresponding columns of the partial product matrix so that the partial products are computed accurately.

It has been discussed earlier that the multiplication of three *3*-bit numbers produces a *9*-bit output results which means that we must have *9* partial products at the end of the operation as opposed to *10* partial products *P1, P2… P10*, as seen in Figure *15*. This is a fallout of using *(7, 3)* adders in our design. They have *3* outputs which mean that the most significant bit of the result must be carried to the second column to the right and the next bit must be carried to the adder on the immediate right. To understand this better, take a look at Figure *17* and section *3.2.2.* which shows the *(7, 3)* adder in better detail.

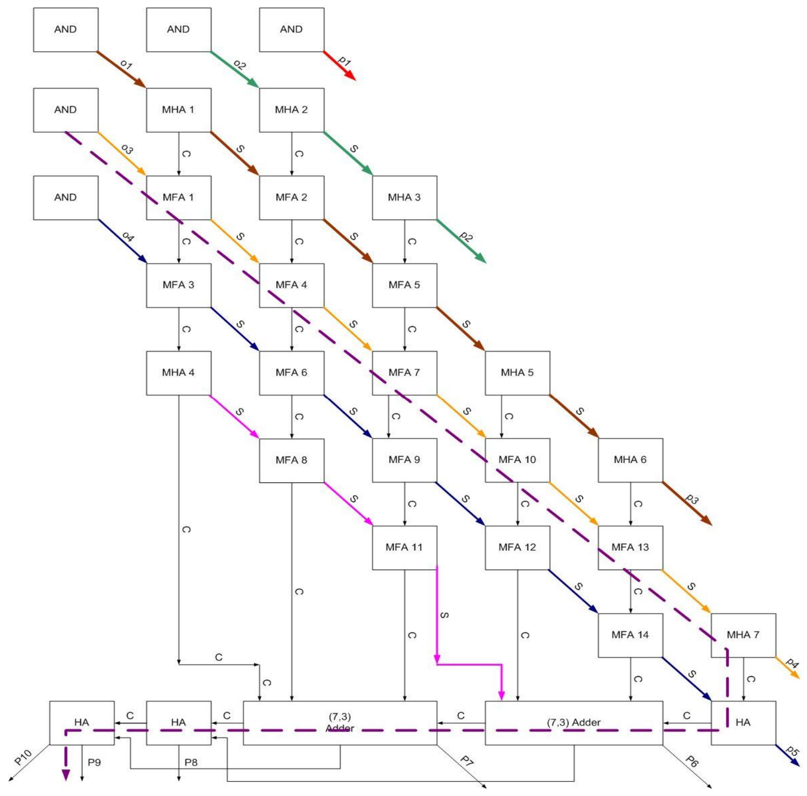


Figure *15*: Cubing Unit Implementation

The last row in the implementation shown in Figure *15* comprises the carry propagate adder part of the *3*-bit x *3*-bit x *3*-bit cubing unit. Every component in the implementation represents one node from the *3*-dimensional view of the parallelograms of the partial product matrix.

3.2.2. The (7, 3) Adder

The *(7, 3)* adder produces *3* outputs for example x_0, x_1 and x_2. While x_0 is the current partial product, x_1 is the carry that must be propagated to the immediate next column and x_2 is the carry to be propagated to the column after that. The size of the carry propagate adder increases with the size of the input operands since the number of columns and the disparity in the column sizes increases.

Let's take a closer look at the *(7, 3)* adder used in the carry propagate section of the cubing unit. Consider the case in which seven binary *1*'s need to be added. The result in binary $y_2y_1y_0$ would *1 1 1*. This means that while $y_0 = 1$ is added to the same column while adding this binary result to another binary number, $y_1 = 1$ must be taken to the immediate next column on the left and $y_2 = 1$ to the column after that.

Figure *17* shows the this component in greater detail

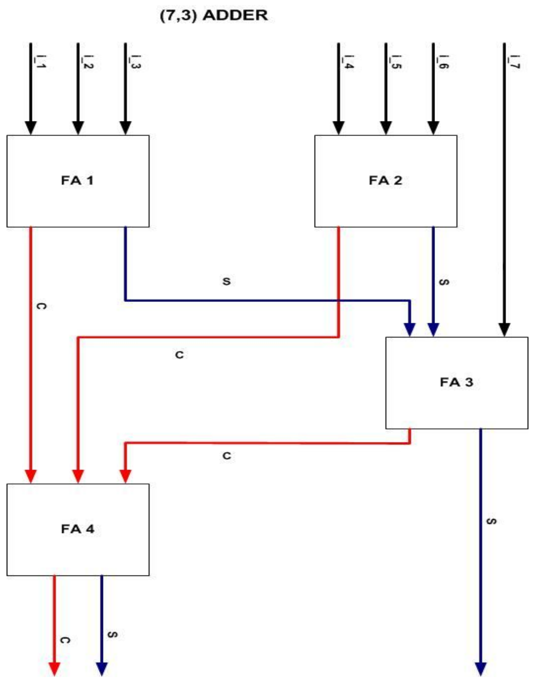


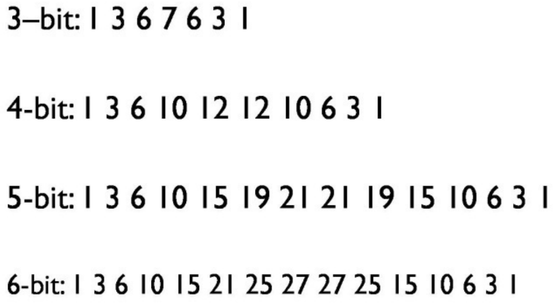
Figure *17*: The *(7, 3)* Adder

The black wires represent the inputs; the red wires represent the carry-outs or carry-ins while the blue wires represent the sums at the end of the computation.

As mentioned earlier, the extra bit *P10* in Figure *9* is a fallout of using *(7, 3)* adders in the implementation of the carry propagate part of the cubing unit. The partial product *P10* is always a binary *0* and can be ignored. To understand this better, consider the case where the binary number *111* i.e. decimal number *7*, is multiplied thrice by itself. This means that the operation *7*^{3} is being performed. The result is decimal *343* or binary *0101010111*. Hence *P10* will always be *0*. The half adder can therefore be modified to a Simplified Half Adder or SHA such that the carry out from it is ignored.

3.2.3 Trends in the partial product matrix

The trends in the columns of the partial product matrix are as shown below in Figure *10*.


3–bit: 1 3 6 7 6 3 1

4-bit: 1 3 6 10 12 12 10 6 3 1

5-bit: 1 3 6 10 15 19 21 21 19 15 10 6 3 1

6-bit: 1 3 6 10 15 21 25 27 27 25 15 10 6 3 1

Figure *16*: Trends in the partial product matrix

In the above Figure, the different numbers represent the number of partial products in each column of the corresponding partial product matrices. So for example, in the case where three *3*-bit numbers are being multiplied, the partial product matrix has *7* columns each containing *1, 3, 6, 7, 6, 3* and *1* partial products. It is important to analyze this trend in order to arrive at an algorithmic regularity between the designs corresponding to different sized operands.

3.2.4. The Algorithm Problem

Studies conducted on the *3-, 4-, 5-* and *6*-bit versions of the cubing unit reveal that in each case, the maximum number of carry-ins from the columns to the different blocks of the carry propagate adder in the last row are *3, 4, 5, 6* respectively. For the smaller designs, the use of the *(7, 3)* adder in the CPA is sufficient. But as the length of the input word increases, there will be a requirement for using larger adders like the *(15, 4)* adder, etc. Not only does this translate into higher area requirement for the implementation but also leads to larger delays through the CPA which in turn leads to the problem of glitching. This is one important aspect of array structures- the problem of glitching becomes more and more important as the length of the input word increases since this automatically means larger CPAs. This has been seen in the case of the CSAM and Baugh Wooley multipliers as well.

3.3. Two's complement Cubing

Similar to the Baugh Wooley multiplier, the cubing unit must be modified in order to be able to perform the multiplication of three signed numbers.

A two's complement number has the following representation:

$$A = -a_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} a_i .2^i$$

3.3.1. Architecture of the two's complement cubing unit

Two's complement cubing is complicated by the fact that there are three terms to multiply producing *8* terms, similar to the multiplication scheme presented above. So, for cubing of two's complement integers, the new partial product matrix, which forms a $2^{3\cdot n-2}$ answer, is formed by the following terms:

$$A^3 = -a_{n-1}.a_{n-1}.a_{n-1}.2^{3.n-3} - \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i.a_j.a_{n-1}.2^{i+j+n-1}$$

$$+ \sum_{i=0}^{n-2} a_i.a_{n-1}.a_{n-1}.2^{i+2.n-2} + \sum_{j=0}^{n-2} a_j.a_{n-1}.a_{n-1}.2^{j+2.n-2}$$

$$+ \sum_{k=0}^{n-2} a_k.a_{n-1}.a_{n-1}.2^{k+2.n-2} + \sum_{k=0}^{n-2}\sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i.a_j.a_k.2^{i+j+k}$$

$$- \sum_{k=0}^{n-2}\sum_{i=0}^{n-2} a_i.a_k.a_{n-1}.2^{i+k+n-1} - \sum_{k=0}^{n-2}\sum_{j=0}^{n-2} a_j.a_k.a_{n-1}.2^{j+k+n-1}$$

Using a similar analysis to the Baugh-Wooley multiplication matrix, the negative terms in the two's complement cubing matrix can be complemented as previously done. The new cubing matrix now becomes:

$$A^3 =$$

$$\overline{a_{n-1}}.2^{3.n-3} + 2^{3.n-3} + \textstyle\sum_{i=0}^{n-2}\sum_{j=0}^{n-2}\overline{a_i.a_j.a_{n-1}}.2^{i+j+n-1} + K +$$

$$\textstyle\sum_{i=0}^{n-2} a_i.a_{n-1}.2^{1+2.n-2} + \sum_{j=0}^{n-2} a_j.a_{n-1}.2^{j+2.n-2} + \sum_{k=0}^{n-2} a_k.a_{n-1}.2^{2.n+k-2} +$$

$$\textstyle\sum_{k=0}^{n-2}\sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i\ .a_j.a_k.2^{i+j+k} + \sum_{k=0}^{n-2}\sum_{i=0}^{n-2}\overline{a_i.a_k.a_{n-1}}.2^{i+k+n-1} + K +$$

$$\textstyle\sum_{k=0}^{n-2}\sum_{j=0}^{n-2}\overline{a_j.a_k.a_{n-1}}.2^{j+k+n-1} + K$$

The value *K* represents the terms for adding an ulp and most significant bits that need to be complemented, as described previously with the multiplication example. However, cubing is significantly more complicated due to *8* terms that are produced. Moreover, the value of *K* must be analyzed for a two-dimensional summation. In multiplication, there was no need to worry about this term, because it was only a single summation, however, in cubing the two-dimensional sum produces several ulps and complementations. In order to Figure out *K*, it is easiest to start with an example, and then simplify it into a closed form solution. Since all values of *K* are the same, since the weights are all the same (i.e. *2j*+*k*+*n*−1), the value of *K* can be computed for one term and then multiplied by *3*. For *n* = *4* cubing matrix, the value of *K* is given below:

$$K = [(2^{3.n-6} + 2^{3.n-5} + 2^{3.n-4} + 2^{3.n-3}) + (2^{3.n-5} + 2^{3.n-4} + 2^{3.n-3}) + (2^{3.n-4} + 2^{3.n-3})] + (2^{n-1} + 2^{n} + 2^{n+1})$$

$$K = [3.2^{3.n-3} + 3.2^{3.n-4} + 2.2^{3.n-5} + 2^{3.n-6}] + [2^{n-1} + 2^{n} + 2^{n+1}]$$

Examining these terms and subtracting n arbitrary terms forms two simple summations:

$$K = 2^{3.n-3} + \sum_{i=1}^{n-1}\sum_{j=1}^{i} 2^{3.n-3} + \sum_{k=1}^{n-1} 2^{n+k-2}$$

The value of K can be simplified into a close form value, as follows:

$$K = \left[\frac{1}{4}.n.2^{3.n} + 2^{3.n-2}.(\frac{1}{2})^{n} - \frac{1}{4}.2^{3.n} - 2^{3.n-3}\right] + [2^{2.n-2} - 2^{n-1}]$$

For example, cubing *4*-bit integers produces the value of *3 · K = 8721*. Compared to multiplication, this constant is larger; however, having a closed-form solution allows any cubing matrix to be formed for two's complement operations.

For a cubing operation involving *3*-bit operands, *n = 3*. Hence the value of *K = 220*. The value *3.K = 660 =* $(1010010100)_2$ must be added into the array. The following Figure shows the modified cubing unit to perform *2*'s complement cubing operation:

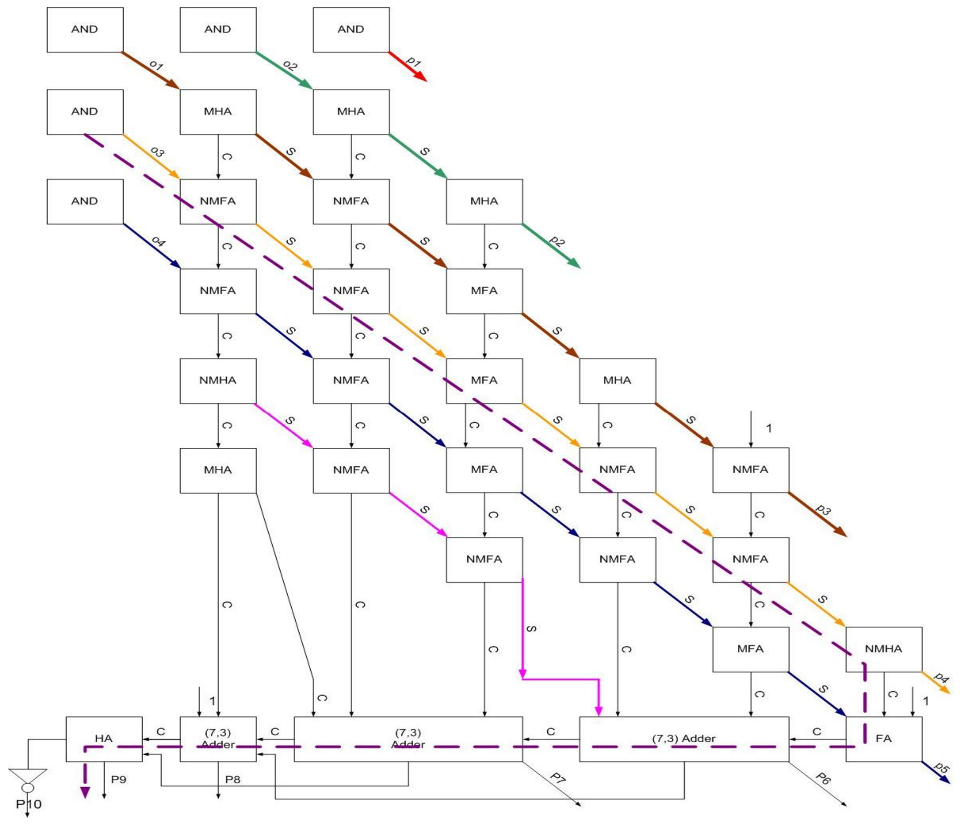


Figure *18*: Implementation of a two's complement cubing unit

The dashed line shows the critical path through the array unit. Figure *18* shows that some components have been NAND-modified, similar to what we looked at in the *3*-bit two's complement multiplier case.

3.3.2. Example

Once again, in order to verify that this design indeed works, let's look at the example shown below. The operation *3 x 3 x (-1) = -9 = (1111110111)*$_{2.}$

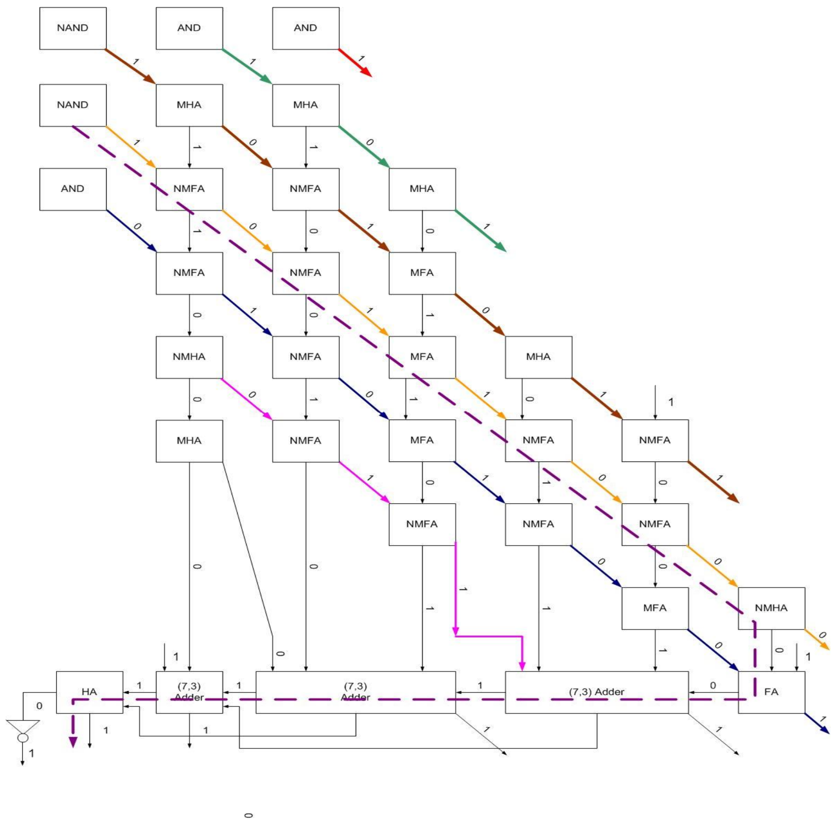


Figure *19*: Example of a *3*-bit two's complement Cubing Unit

3.3.3. Performance Consideration

The carry-save implementations of the cubing unit have performed well for small length operands as will be seen in Chapter *4*. The delay of the CSAM and Baugh Wooley multipliers increases quadratically with the size of the input operands whereas the delay of the cubing unit is seen to increase somewhat exponentially with the size of the input operand. The area requirement increases dramatically with the increasing size of the input operands as well. Careful floor planning is required since the number of components in the design increases as well. The delay through the CPA is a point of major concern since it falls in the critical path and the delay increases as the size of the CPA increase with larger input operands.

CHAPTER 4

RESULTS

This chapter takes a look at deriving the delay and area requirements of various architectures using algorithmic analysis. The results obtained from testing and synthesis for these architectures are also discussed and compared.

4.1 Algorithmic analysis delay:

Before analyzing the array multipliers or cubing units for different *n*-bit operands, let's take a look at the delays along the critical paths of the various components that form these structures like the full adder, half adder, etc.

When included in an array, the critical path along the modified half adder (or half adder) and the modified full adder (or full adder) is shown in Figures *5* and *6* respectively. The delays are enumerated below:-

1. Half Adder/Modified Half Adder:

   a_i, b_i to c_{i+1} : 1Δ

   a_i, b_i to s_i : 3Δ

2. Full Adder/Modified Full Adder:

   a_i, b_i to s_i : 6Δ

   a_i, b_i to c_{i+1} : 5Δ

c_i to s_i : *3*Δ

c_i to c_{i+1} : *2*Δ

3. Ripple Carry Adder (RCA): the delay of an n-bit RCA is given as $(2n + 4)\ \Delta$.
4. The *(7,3)* adder: the critical path of the *(7,3)* adder passes through *3* full adders. The delay along its critical path was computed as 17Δ.

Where 1Δ is the delay of a basic gate such as an OR, NOT and AND gate with a fan-in of *4* or less.

The delay along the critical path of an (n x m) array multiplier is characterized as follows:

Delay $_{(n\ x\ m)}$ = *1* AND gate + *1* modified half adder + *(m-2)* modified full adders + *(n-1)* ripple carry adders

$$= 1\Delta + 3\Delta + (m-2)\ \Delta + \{2(n-1) + 4\}\ \Delta$$

Using the above equation, the computed delays for the various array multipliers are as follows:

(3 x 3) array multiplier has the delay *18* Δ along its critical path.

(6 x 3) array multiplier has the delay *24* Δ along its critical path.

In order to compute the product of three *3*-bit operands using traditional hardware techniques, the first pass would require the use of a *(3 x 3)* array unit and the second pass would require the use of a *(6x3)* array unit. The total time required for a single computation to complete would be the addition of the delays along the critical paths of these two array units. Hence, the total delay sums up to *42* Δ.

Similarly, the critical path delays for the *(4 x 4)* and *(4 x 8)* array units were computed as *26* Δ and *34* Δ respectively. Hence the total time required for the computation *(a x b x c)* where *a*, *b*, and *c* are *4*-bit operands can be summed up as *60* Δ.

To find the product of three *6*-bit operands using the traditional array multipliers, it would require one pass through a *(6 x 6)* array unit and a second pass through a *(12 x 6)* array unit. The delays along the critical paths for these designs are *42* Δ and *54* Δ respectively. Hence, the total delay can be summed up as *96* Δ.

The delay along the critical path for the *3*-bit cubing unit that was designed in Chapter *4* can be characterized as follows:

Delay $_{(3 \times 3 \times 3)}$ = *1* AND gate + *1* modified half adder + *5* modified full adders + *3* half adders + *2 (7, 3)* adders

= *61* Δ

Similarly, we computed the delays along the critical paths of the *(4 x 4 x 4)* and *(6 x 6 x 6)* cubing units. They were found to be *130* Δ and *270* Δ respectively. These results from our algorithmic analysis have been tabulated in Appendix A. It is seen that while the delays for the traditional multipliers increase linearly with increase in input operand size, the growth in the critical path delays of the cubing unit in our designs is almost exponential.

4.2 Algorithmic Analysis of Area

An estimate the areas of the various components in terms of the area of a basic gate, *1* g can be made. This section deals with the algorithmic analysis if the area requirement for different array structures.

1. Half Adder: It comprises of *4* gates as shown in Figure. Hence its area is *4*g.
2. Modified Half Adder: this unit uses an additional AND gate as compared to the half adder. Hence its area is *5*g.
3. Full Adder: the full adder comprises of *2* half adder and an OR gate for the final carry-out computation. Hence its area is *9*g.
4. Modified Full Adder: the modified full adder comprises of an extra AND gate as compared to the full adder. Its area is *10*g.

The algorithmic analysis of area gives the following results for the traditional multiplier implementations for the *3*-bit, *4*-bit and *6*-bit versions: *171* g, *348* g and *882* g respectively. The algorithmic analysis of the area of the cubing unit designs corresponding to *3*-bit, *4*-bit and *6*-bit operands yielded the numbers *274* g, *534* g, *2144* g respectively.

4.3 Testing

The HDL for the various designs was written in Verilog. The test vectors were generated using a test bench, also written in Verilog. The design was debugged using the emacs editor till all the errors and warnings were carefully dealt with. The results are shown in the Appendix C. A different and random test vectors were used. Appendix C shows one instance.

4.4. Synthesis

The design was synthesized using the Synopsys Design Vision tool in the *180 nm* SCMOS MOSIS process at a frequency of *200 MHz*, a temperature of *25 degrees* Celsius and a supply voltage of *1.8 Volts*. The HDL codes for the 6-bit versions of the array multipliers and the cubing unit were submitted to the synthesizer and the results that were obtained were quite surprising. Contrary to the algorithmic analysis of delay, the synthesis showed that the delay along the critical paths of the *(6 x 6)* and *(12 x 6)* array units was *3.77 ns* and *4.00 ns* respectively. This adds up to *7.77 ns* for a single computation. In contrast, the cubing unit required *7.47 ns* to complete a single pass through it. The cubing unit design is clearly better in terms of delay. But it has high area requirements. The total area required for the array implementation for the *6*-bit version is *705* cells as compared to *1319* cells required for our design. These results have been tabulated in Appendix B.

CHAPTER 5

CONCLUSION

This study has looked at array multipliers and cubing units and their two's complement versions. Due to the interesting nature of the partial product matrix of the cubing operation, it is quite challenging to arrive at a rectangular structure during synthesis. However, during layout, the lower half of the design can be flipped in order to fill in the blank spaces and a compact floorplan and hence, a compact layout can be arrived at.
Also, it is seen that as the length of the input words increases, so does the requirement for larger adders in the carry propagate chain in the last row of the cubing unit. This automatically translates into more area for the design.

Our design is generic in the sense that our goal was to design an array structure to perform the operation *a x b x c* where *a, b* and *c* and three n-bit quantities. The design can be optimized for operations involving cubes of the same number, for example, x^3. For example, the nodes in the design that require an AND gate for the operation $a_0.b_0.c_0$ could be replaced by a single wire that propagates a_0. Similar optimizations can be applied to other aspects of the design as well.

As mentioned earlier, an important point of concern is also the delay along the carry propagate adder. Also, the length of the ladder increases as the size of the input operand increases, which means increased delay since the carry-outs from the previous adders are connected to the carry-ins of the successive adders. This means that at some components along the critical path, some of the signals arrive earlier at a particular component. This is known as glitching where in a single transition on a primary input may give rise to multiple switchings on the internal nodes [12].

The results showed that the design for the cubing unit in the *6*-bit case worked faster compared to the method employing the traditional CSAM. It will be interesting to further study similar comparisons for the *8*-bit, *16*-bit and *32*-bit case.

While the area and delay for the Braun multiplier or the carry save array multiplier increases quadratically with the number of bits in the input word, the study shows that the area and delay for the array based cubing unit increases almost exponentially. Further research into this would be worthwhile to find out the maximum size of the input words for which the cubing unit can be employed without much performance degradation.

Another interesting aspect that can be looked at is the power dissipation from the cubing unit and comparison with the power dissipated by corresponding carry save array multipliers. .

REFERENCES

[1] R. E. Goldschmidt. Applications of Division by Convergence. MSc dissertation, M.I.T., 1964

[2] Q. Zhang, R. Eagleson and T. M. Peters. Gpu- Based Image Manipulation And Enhancement Techniques For Dynamic Volumetric Medical Image Visualization. *4th IEEE International Symposium* , April 2007, pp. 1168 – 1171

[3] P. K. Bora, Y. V. Venkatesh and K. R. Ramakrishnan, *Shape From Shading Using Discrete Polynomials*, India, 1990.

[4] R. K. J. Raghunath, H. Farrokh, N. Naganathan, M. Rambaud, K. Mondal, and F. Masci, Hollopeter, *A Compact Carry-Save Multiplier Architecture And Its Applications*, Murray Hill, NJ

[5] S. Winograd, *Journal of the Association for Computing Machinery,* April 1965.

[6] C. R. Baugh and B. A.Wooley. A Two's Complement Parallel Array Multiplication Algorithm. *IEEE Trans. Comput.*, vol. C-22, pp. 1045–1047, 1973.

[7] J. E. Stine . *Digital Computer Arithmetic Datapath Design Using Verilog HDL.* Springer, 2004.

[8] K. S. Yeo and K. Roy. *Low Voltage, Low Power VLSI Subsystems*. McGraw-Hill Professional, 2005

[9] W. Chen. *The VLSI Handbook.* CRC Press, 2000

[10] M. Rafiquzzaman . *Fundamentals of Digital Logic and Microcomputer Design.* Wiley-IEEE, 2005

[11] J. E. Stine and J. M. Blank. Partial Product Reduction for Parallel Cubing. *IEEE Computer Society Annual Symposium* , March 2007, pp. 337 – 342

[12] C. Teng; A. M. Hill and S. Kang. Estimation Of Maximum Transition Counts At Internal Nodes In CMOS VLSI Circuits. *Computer-Aided Design, Digest of Technical Papers: IEEE/ACM International Conference,* Nov. 1995 pp. 366 – 370

[13] M. Vishwanath, R. M. Owens and M. J. Irwin. The Arithmetic Cube: Error Analysis And Simulation. *Application Specific Array Processors, Proceedings of the International Conference,* Sept. 1991 pp. 129 - 143

[14] W. Wu, H. Gao and J. Li. New Algorithm for Computing Cube on Very Large Compressed Data Sets. *Knowledge and Data Engineering IEEE Transactions,* Vol. 18, Issue 12, Dec. 2006, pp. 1667 - 1680

APPENDICES

A. Algorithmic analysis of area and delay of the CSAMs and cubing units

	3-bit	4-bit	5-bit
Cubing Unit Area	274 gates	534 gates	2,144 gates
CSAM Area	171 gates	348 gates	882 gates
Cubing Unit Delay	61 Δ	130 Δ	270 Δ
CSAM Delay	42 Δ	64 Δ	102 64 Δ

B. Results after Synthesis – Synopsis Design Vision

Functional Unit	Critical Path Delay (ns)	Area (No. of Cells)
6-bit x 6-bit Array Multiplier	3.77 ns	177
12-bit x 6-bitArray Multiplier	4.00 ns	528
6-bit Cubing Unit	7.47 ns	1319

C. Results after testing the Verilog code for the 6-bit x 6-bit x 6-bit cubing unit:

```
Input operands   ||     Output in binary form

(in decimal)

33   33    33        || 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1

33   33    33        || 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1
```

VITA

Vrushali Yewle

Candidate for the Degree of

Master of Science

Thesis: CUBING UNITS USING CARRY-SAVE ARRAY IMPLEMENTATIONS

Major Field: Electrical Engineering

Biographical:

Education: Received Bachelor of Engineering degree in Electronics and Telecommunications from University of Mumbai, Mumbai, India in August 2006. Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2009.

Experience: Worked as a graduate teaching assistant from Fall 2006 to Fall 2008 on the courses Electronic Devices and Circuits and Semiconductor Device Physics. It also involved work on a project in the field of Engineering Education Research.

:

Name: Vrushali Yewle Date of Degree: May, 2009

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: CUBING UNITS USING CARRY-SAVE ARRAY IMPLEMENTATIONS

Pages in Study: 38 Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

Scope and Method of Study: This study is aimed at designing a specialized functional unit to perform the operation of Cubing. The design has been implemented based on the concept of carry-save array multipliers. The carry save concept aims at accelerating the process of addition by delaying the carry propagation operation till the last step. The motivation behind using array structures is simple. Array structures are regular and easy to design. This paper first looks at a simple cubing unit that can accept only unsigned inputs. This design is then modified based on mathematical derivations and architecture for signed cubing unit or two’s complement cubing unit is derived. The designs have been tested, synthesized and compared with the traditional multiplication techniques for area and delay.

Findings and Conclusions: When the different designs for various bit sizes of operands were tested and synthesized, some really interesting results were arrived at. The algorithmic analysis showed that the cubing unit implementation would require more delay and area as compared to two passes for the same operation through corresponding traditional CSAMs. After synthesis, while the results agreed with the area requirement, for the 6-bit version, it was seen that the cubing design is actually faster as compared to the traditional implementation. The interesting thing about the cubing unit design is that its area requirement and delay increases almost exponentially with the length of the input operands. Careful floorplanning and layout techniques must be employed in order to arrive at a good design.

ADVISER’S APPROVAL: Dr. James E. Stine, Jr.