1-1-2006

Fixed-width digital multipliers based on recursive architectures.

Kevin Biswas
University of Windsor

FIXED-WIDTH DIGITAL MULTIPLIERS BASED ON RECURSIVE ARCHITECTURES

by

Kevin Biswas

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Electrical Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

2006

Your file Votre référence
ISBN: 978-0-494-35955-6
Our file Notre référence
ISBN: 978-0-494-35955-6

ABSTRACT

Signal processing applications, in general, require a constant word size throughout the processing system. This poses a problem for basic integer arithmetic operations, where the result of each operation has a tendency of differing from the original operand size. Multiplication is of the biggest concern since each operation results in a product that is potentially twice as large as the original operand widths. To alleviate the problem of expanding word widths, fixed-width multipliers are utilized.

This thesis will present some novel architectures for fixed-width recursive multipliers. The high-performance recursive multiplier exhibits an inherent hierarchical structure consisting of several sub-multipliers, which makes it suitable for fixed-width applications. Four truncation schemes targeting the recursive multiplier have been proposed, all of which improve error statistics and generally reduce gate complexity, propagation delay, and power consumption, with respect to the original full-width multiplier. A fixed-width architecture targeting multi-level recursive multipliers will also be presented.

To my beloved family.

ACKNOWLEDGEMENTS

The work presented in this thesis would not have been possible without the support from my colleagues, mentors and family. I am deeply indebted to all of them.

Firstly, I would like to express my sincere gratitude and appreciation to my supervisor Dr. Majid Ahmadi for his generous support and guidance. He has made, and will continue to make, tremendous impact on me academically, socially and personally, through his endless enthusiasm towards my work and excellent advice in all of my endeavours.

I would like to thank the faculty at the University of Windsor, including my internal reader, Dr. Huapeng Wu, and Dr. Maher Sid-Ahmed for their advice, guidance and words of inspiration. I would also like to thank my external reader, Dr. Arunita Jaekel for her patience and support.

To Mr. Ashkan Hosseinzadeh Namin and Mr. Pedram Mokrian I also extend my sincere gratitude and appreciation, not only for their sound technical advice, but for their great friendship.

Finally to my parents, Olive and Nihar, my brother, Robert, and Nichole Jun, I must extend my most sincere love and gratitude, for their unending support, patience and enthusiasm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION TO COMPUTER ARITHMETIC

1.1 Overview of Computer Arithmetic

The computer has permeated our professional and private lives by simplifying tasks which were once difficult or even impossible to carry out. Computers have a long history, dating back several centuries, when mathematicians and scientists first developed machines to help them manipulate and compute numbers [1]. The field of computer arithmetic was established at the birth of these electronic computing machines. Today the field is a sub-set of computer architecture and deals with the implementation of arithmetic algorithms in hardware and software for processor architectures and, more specifically, arithmetic logic units (ALU). This thesis deals with the multiplication architectures, which are critical components of ALUs and other systems which perform numerical processing. Specifically, multiplication in fixed-width applications will be studied.

1.2 Thesis Highlights

This thesis will present a general investigation of fixed-width multiplication and truncation schemes, and will describe some novel architectures for fixed-width recursive multipliers [2]. The recursive multiplier, presented by Swartzlander et al. [3] exhibits an inherent hierarchical structure consisting of several sub-multipliers, which makes it suitable for fixed-width applications. Four truncation schemes targeting the recursive multiplier have been proposed, all of which improve error statistics and generally reduce gate complexity, propagation delay, and power consumption with respect to the full-width multiplier. Detailed error analysis and architectural complexity analysis have been carried out for each design.

Hardware implementation of the proposed fixed-width multiplier architectures has been carried out in Altera Stratix EP1S10F484C5 FPGA. The resulting reductions in propagation delay, power consumption and logic complexity with respect to the full-width recursive multiplier have been tabulated and analyzed.

Further, the idea of fixed-width multipliers based on *multi-level* recursive architectures has been studied in detail. The previous work regarding fixed-width single-level recursive multiplication has been extended to the multi-level case, and error analysis and complexity analysis have been carried out. New mathematical expressions have been derived to estimate potential maximum error and complexity savings for the general case of k levels of recursion.

1.3 Thesis Organization

The thesis will begin with a general overview of digital multiplication, briefly highlighting serial and parallel multiplication algorithms, in Chapter 2. Chapter 3 will give an overview of fixed-width multiplication and truncated multipliers. Further, some of the most well-known truncation schemes available will be described.

Chapter 4 is dedicated to the Recursive Multiplier. An overview of the recursive or "divide and conquer" algorithm for multiplication proposed by Karatsuba and Ofman (1962) [4] will be first given. Application of the algorithm in the digital recursive multiplier [2] will be subsequently presented.

Chapter 5 will present novel architectures for fixed-width recursive multipliers. Four new truncation schemes targeting recursive multipliers will be presented in this chapter along with detailed error and complexity analysis. Chapter 6 will focus on hardware implementation and simulation results of proposed architectures. Chapter 7 investigates fixed-width multiplication using multi-level recursive architectures. The thesis will conclude with a highlight of contributions and some closing remarks in Chapter 8.

CHAPTER 2

DIGITAL MULTIPLICATION OVERVIEW

In modern digital systems, the component responsible for handling arithmetic operations is the Arithmetic Logic Unit (ALU). These units mainly lie in the critical data path of the core data processing system elements. These include microprocessors (CPU), digital signal processors (DSP), in addition to application specific (ASIC) and programmable (FPGA) processing and addressing integrated circuits. Performance of a system, in regards to numerical applications, is directly related to the structure and design of the ALU.

The numerical operations carried out by the arithmetic unit may include, but are not limited to: addition/subtraction, shift/extension, comparison, increment/decrement, complement, trigonometric functions, multiplication, division, square root extraction, logarithmic function, exponential function and hyperbolic functions [5].

One of the critical functions carried out by the ALU is multiplication. Although it is not the most fundamentally complex operation, digital multiplication is one of the most frequently used operations in signal processing and other applications. Because of this, digital multiplication is one of the most widely studied areas in the field of computer arithmetic.

2.1 Basics of Digital Multiplication

Generally speaking, digital multiplication involves a sequence of additions carried out on partial products. The means by which the partial products matrix is summed is the key distinguishing factor amongst multiplication schemes [6].

The partial product array of an M x N bit digital multiplication is determined similarly to traditional pen and paper decimal multiplication. For example, multiplication of multiplier $X = [x_{n-1}, x_{n-2}, x_{n-3}, \ldots x_2, x_1, x_0]$ and multiplicand $A = [a_{m-1}, a_{m-2}, a_{m-3}, \ldots a_2, a_1, a_0]$ yields the final product $(n+m)$-bit product:

$$P = [p_{n+m-1}, p_{n+m-2}, p_{n+m-3}, \ldots p_2, p_1, p_0] = x_{n-1}(a_{m-1}, a_{m-2}, a_{m-3}, \ldots, a_2, a_1, a_0) + x_{n-2}(a_{m-1}, a_{m-2}, a_{m-3}, \ldots, a_2, a_1, a_0) + \ldots + x_1(a_{m-1}, a_{m-2}, a_{m-3}, \ldots, a_2, a_1, a_0) + x_0(a_{m-1}, a_{m-2}, a_{m-3}, \ldots, a_2, a_1, a_0)$$

This multiplication can be illustrated in Figure 2.1, below.

				a_3	a_2	a_1	a_0
				x_3	x_2	x_1	x_0
				x_0a_3	x_0a_2	x_0a_1	x_0a_0
			x_0a_3	x_0a_2	x_0a_1	x_0a_0	
		x_0a_3	x_0a_2	x_0a_1	x_0a_0		
	x_0a_3	x_0a_2	x_0a_1	x_0a_0			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Figure 2.1: Example of Pen and Paper Multiplication

A convenient notation for digital multiplication that visually represents the bits in an algorithm is dot notation which was introduced in [7][8]. The nature of the dot diagram is to depict the bits using the relative position of individual bits, and the manner in which they are manipulated, irregardless of the value of each bit. Figure 2.2 shows the partial product array for a 16x16 multiplication [7].

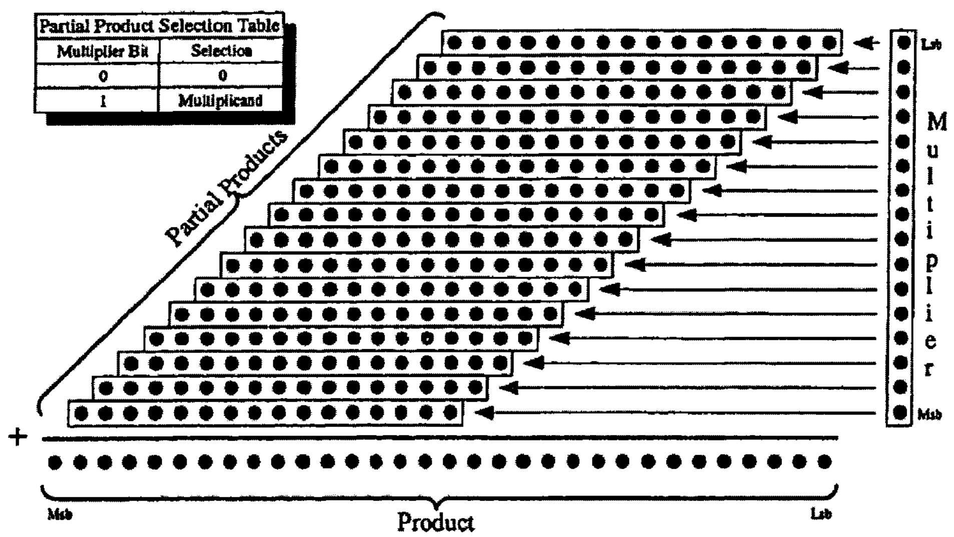


Figure 2.2: Partial Product Array for a 16-bit Multiplication [7]

2.2 Sequential Multiplication

Fundamentally, digital multiplication can be carried out through a sequence of shifts and additions of the *multiplicand* to the partial product accumulator register based on the values of the individual bits comprising the *multiplier*. This primitive form of multiplication, known as shift-add multiplication, is very slow, despite having a very simple implementation. The number of cycles required to perform a full multiplication is linearly proportional with the size of the multiplier, and each cycle has a delay of the required fast adder. A sequential multiplier is shown in Figure 2.3.

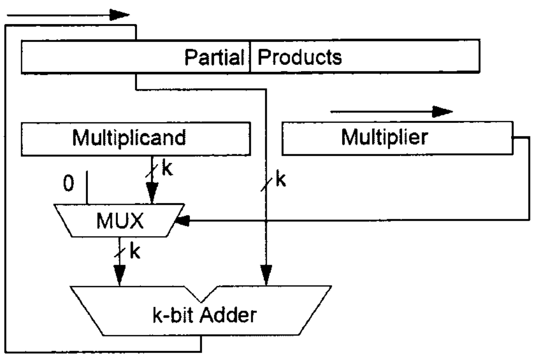


Figure 2.3: Sequential Right-Shift Multiplier [6]

A variation of the basic form of digital multiplication is the high-radix multiplication scheme. This form is similar to the shift-add algorithm mentioned before, but differs in that more than one bit of the multiplier is utilized on each clock cycle. Thus the number of clock cycles is reduced. However, a requirement for this form of multiplication is the availability of fixed multiples of the multiplicand [5]. Figure 2.4 depicts the implementation of a radix-4 multiplier where two bits of the multiplier are used in a clock cycle [6]. As can be seen, the multiples of the multiplicand, A, 2A, and 3A, need to be available. Thus the higher the radix of a multiplier, the more stored values will be required. Higher radix multipliers provide faster computation; but this is at the expense of additional hardware overhead consisting of shift circuitry and storage registers for the required multiples of the multiplicand.

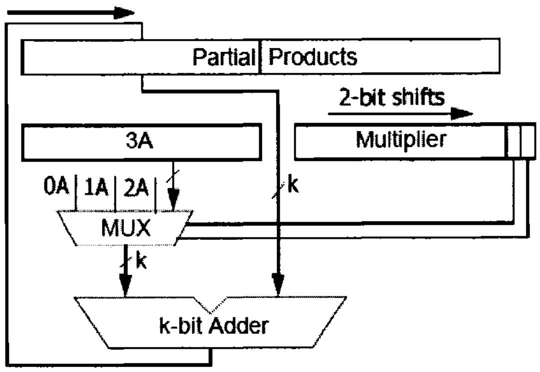


Figure 2.4: Multiplication Performed Using Radix-4

2.3 Parallel Multiplication

As mentioned above, serial multiplication and the concept of shift and add algorithms is a primitive form of multiplication techniques which offers simple implementation, but lacks the performance of parallel multipliers. Most modern high-performance systems require faster algorithms for multiplication to reduce computation latency as much as possible.

There are two distinct categories of parallel multipliers, namely, linear parallel multipliers, and column compression multipliers (tree multipliers). The distinguishing characteristic of parallel multipliers is that partial products are generated simultaneously, and can actually be considered a special case of high-radix multiplication, where the highest possible radix is used, i.e. radix-2^k[6]. As well, parallel multipliers limit latency associated with carry propagation to one final fast adder.

Linear parallel multipliers are more commonly known as array multipliers. The term "linear" comes from the linear relationship that exists between operand size and latency. The array multiplier exhibits a highly regular layout as shown in the 8-bit multiplier in Figure 2.5 [9]. The orderly arrangement of the multiplier cells makes the design ideal for automated layout techniques, where bits of the two input operands are made available across the arrangement of full adder cells. Basically the outputs of the adders trickle accordingly across the array until the edges of the structure, where the product bits are outputted. However, the limitation with the array scheme is that partial products are introduced and reduced only one row at a time, not in parallel like in tree multipliers. This results in slower performance. The delay of the array multiplier has a linear relationship, O(k), with respect to operand size.

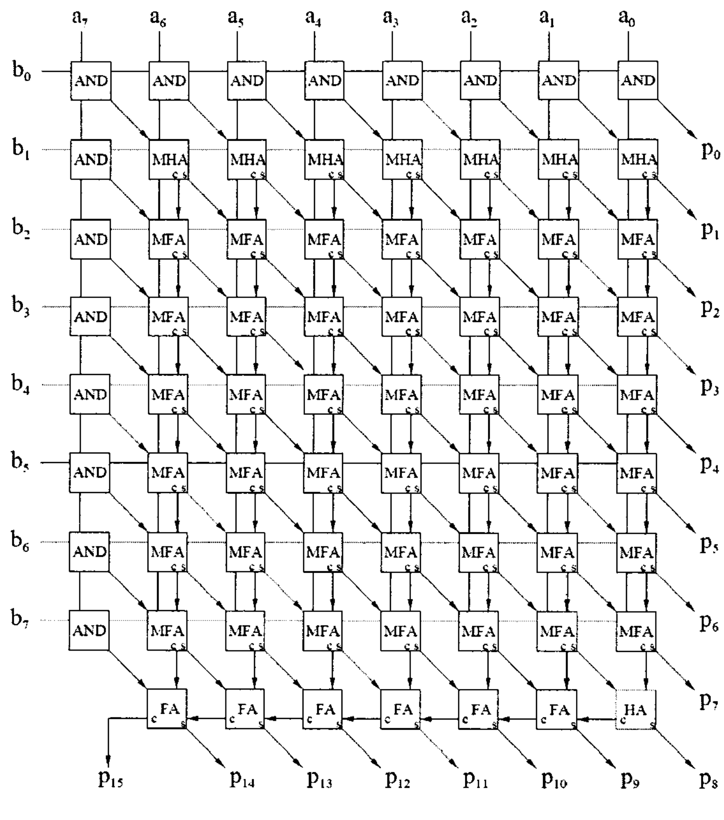


Figure 2.5: Standard Layout of an Array Multiplier
(MFA = full adder + AND gate, MHA = half adder + AND gate)

The tree multiplier, unlike the array multiplier, offers the potential for only a logarithmic increase in delay relative to operand size. The foundation for these multipliers was laid out in the 1960s by work carried about by C.S. Wallace, Luigi

Dadda, and Yu Ofman [10][[11]. In these designs, once bits of the partial product array are generated (in parallel), they are passed onto a reduction network, which performs column-wise compression of the bits, forming two final partial products. Subsequently, a final fast adder is used to sum these last two terms. A flow diagram of the column compression multiplication process is shown in Figure 2.6 [7]. Latency approximation of a column compression multiplier shows that delay is logarithmic O(log(k)) with operand size, a significant improvement over array multipliers, in terms of speed.

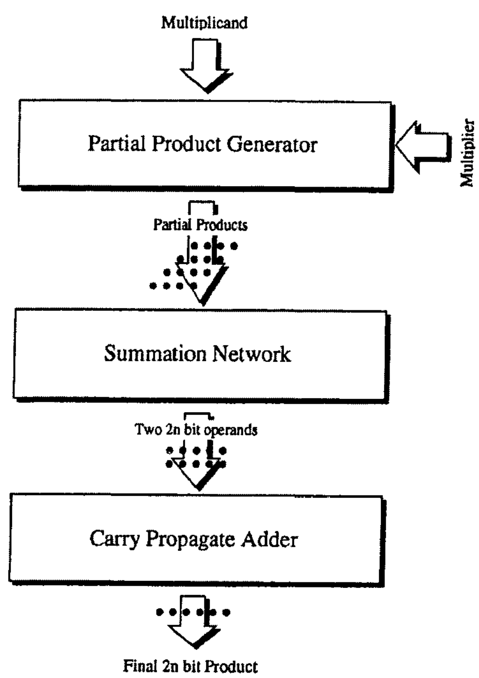


Figure 2.6: Flow Diagram of a Column Compression Multiplier

Wallace [10] initially proposed the method of using Carry-Save Adder (CSA) arrays to carry out the column-wise compression of the partial product bits. Consisting of a series of non-interlinked Full-Adder blocks, CSA is the most commonly used form of multi-operand adder. Luigi Dadda proposed a systematic methodology for laying out the CSA column compression tree so that the minimum number of counters is utilized [11]. Wallace and Dadda multiplication schemes are depicted in Figure 2.7.

Despite the characteristic high speed performance of column compression multipliers, there are several drawbacks when taking into consideration their implementation. Column compression multipliers exhibit a highly irregular architecture leading to inefficient VLSI layout. As process technologies delve into submicron dimensions, irregular interconnections can potentially cause issues like clock skewing and interconnect delay [12].

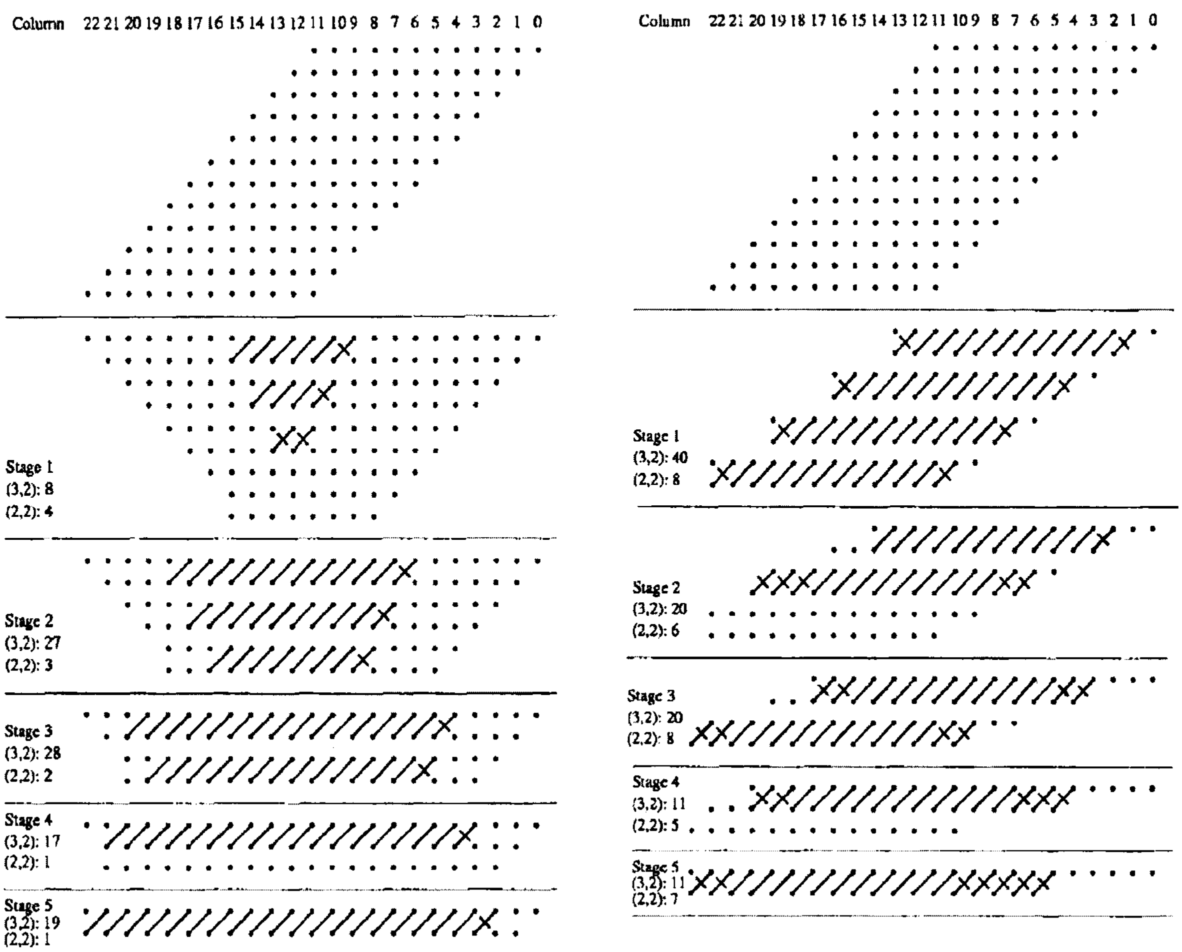


Figure 2.7: Dot Diagrams of Dadda (left) and Wallace (right) Multipliers

2.4 Floating Point Number System and Multiplication

To achieve the levels of precision demanded by modern systems, it becomes necessary to have a number system that is capable of representing real numbers. Fixed-point systems, in which location of the decimal point is pre-defined, suffer from limited range and/or precision. To alleviate this issue, floating-point number systems are utilized [5]. Unlike fixed-point representations, floating-point system allows for extremely large or small numbers to be described with a high degree of precision by using a dynamic range.

According IEEE standard for binary floating-point systems [13], a floating-point value is defined as:

$$x = \pm f \times b^{e}$$

where x is the floating-point value, f is the fraction of mantissa, b is the base (fixed at b=2) and e is the exponent. Floating point numbers have two distinct representations according to the standard depending on operand size. Figure 2.8 depicts the differences between the two floating point standards, in terms of word structure. The sign (s), exponent (e), and fraction/mantissa (f) form the 32 and 64 bit precision formats. The mantissa is normalized to be in the range of [1,2) so that the most significant bit (MSB) is always a 1. In this way, the leading 1 is removed and considered a "hidden one", thus saving one bit in the representation. To ensure a positive value, the signed integer exponent is biased accordingly. The exponent is biased for 127 for single, and 1023 for double precision formats.

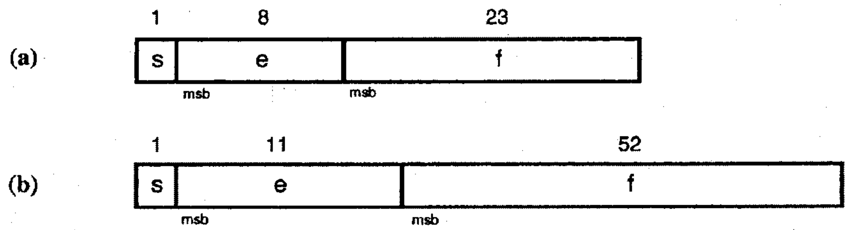


Figure 2.8: IEEE Floating Point Standard Word Widths for (a) Single Precision and (b) Double Precision

Figure 2.9 shows a block diagram of the multiplier implementation for floating point numbers. As described above, floating-point numbers are composed of a biased non-negative integer exponent, and a fixed-point fractional representation of the mantissa. Thus, mathematical operations that are carried out on floating-point numbers will use fixed-point arithmetic units with additional control and rounding circuitry to accommodate for the dynamic range. Because of this, when designing arithmetic hardware, much attention is placed on fixed-point integer units. Conversion to floating point is made possible through additional circuitry. Figure 2.9 also shows the additional blocks surrounding the integer multiplier component.

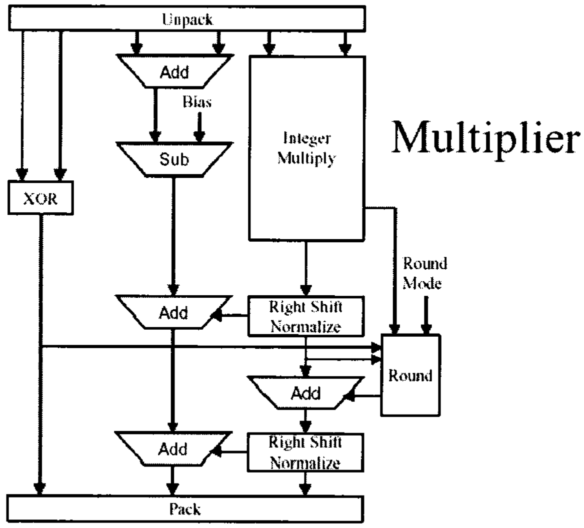


Figure 2.9: A Floating Point Multiplication Scheme [14]

Since the basics of fixed-point arithmetic form the framework for floating point calculations, the remainder of this thesis will target integer arithmetic structures in conventional unsigned binary format.

CHAPTER 3

FIXED-WIDTH MULTIPLICATION

3.1 Overview of Fixed-Width Multiplication

As previously mentioned, multiplication is one of the most widely studied areas in the field of computer arithmetic, due to the frequency of use in numerical applications, such as signal processing. In many of these applications, such as filtering, convolution, Euclidean distance, and Fast Fourier Transform (FFT) [15][23], a constant operand size is required throughout the processing system. When designing arithmetic hardware for such a system, constant operand size is an important constraint to take into consideration. In certain signal processing applications, word sizes could grow significantly large. For example in a complex FFT, if the initial word size is 16 bits real and 16 bits imaginary and the sines/cosines are 16 bits each, maintaining full precision causes a growth of 18 bits (17 bits for the complex multiply and 1 bit for the complex add) per stage. For a 1024 point FFT there are 10 stages producing a final data size of 196 bits [16]. For addition and subtraction, the problem is relatively easy to solve, as the result is potentially only one bit larger than the operands (assuming that the operands are equal in size). Rounding is accomplished by adding a '1' to the least significant bit position and truncating the sum at that position. In many cases the '1' can be added as a carry into the addition so that no extra hardware or time is required to produce a rounded sum or difference [16]. However, of all the arithmetic operations, multiplication is of the biggest concern, because the resulting product of two operands could potentially have a word size that is twice the original operand size.

To alleviate the problem of expanding word widths in multiplication, fixed-width multipliers are utilized [17]. An n x n fixed-width digital multiplier generates only the most significant n product bits with two n-bit inputs. If X and Y are two n-bit unsigned numbers where,

$$X = \sum_{i=0}^{n-1} x_i \cdot 2^i \quad \text{and} \quad Y = \sum_{j=0}^{n-1} y_j \cdot 2^j$$

the product, P, of X and Y, which is a weighted sum of partial products, is therefore:

$$P = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_i y_j \cdot 2^{i+j} = \sum_{k=0}^{2n-1} p_k \cdot 2^k$$

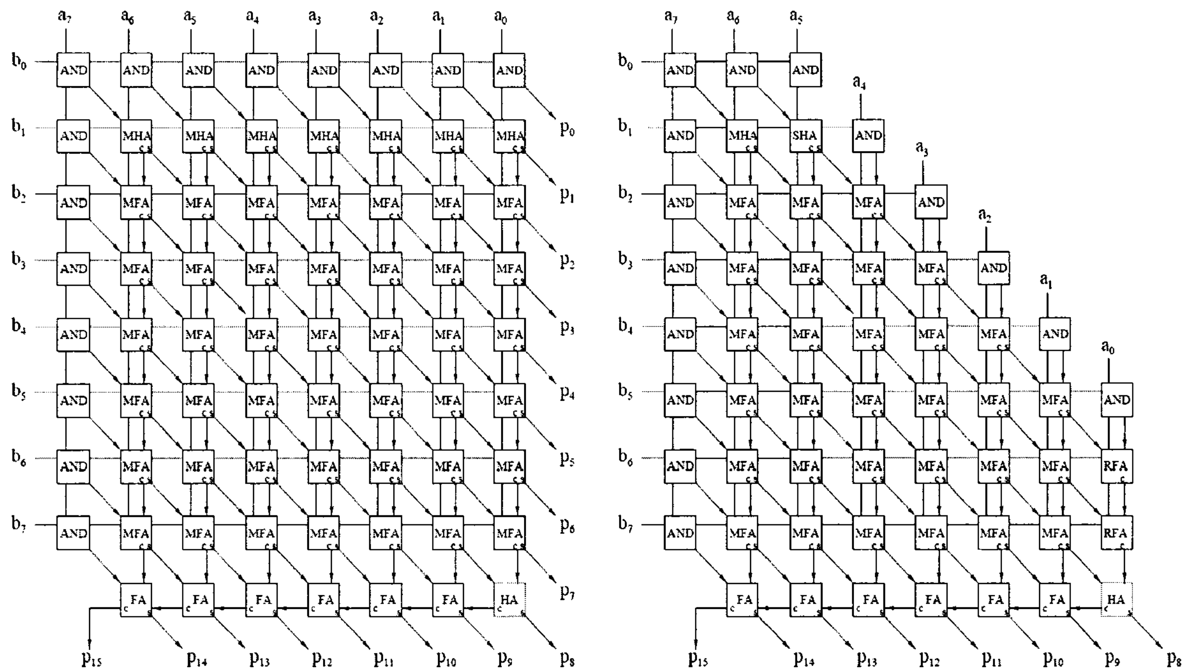
The fixed-width product is:

$$P_{trunc} = \sum_{k=n}^{2n-1} p_k \cdot 2^k$$

Thus a fixed-width multiplier can be easily realized by using only $p_{2n-1}, \ldots, p_n$ outputs of the full-width multiplier. In order to reduce the error due to truncation, output rounding is often carried out. Before truncation, rounding is applied [14] by adding a '1' at the n^{th} least significant position of the product of the full-width multiplier.

3.2 Truncated Multipliers

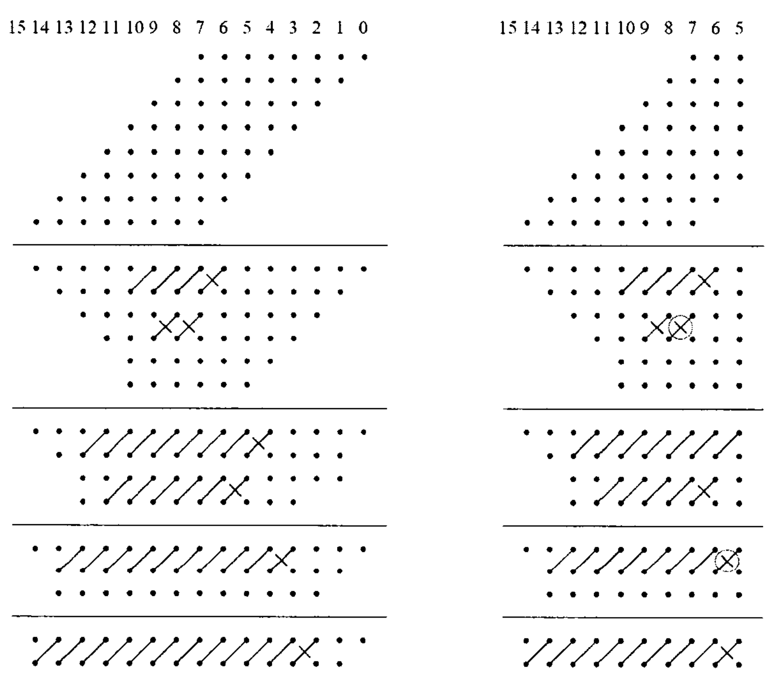
Literature shows that the "fixed-width" property can be exploited to reduce hardware complexity with respect to the full-width multiplier [9]. Truncated multipliers, in which less significant columns of the partial product matrix are removed, are often used in fixed-width applications. Example of a truncated array multiplier and Dadda (tree) multiplier are shown in Figures 3.1 and 3.2.



(a) Standard Array Multiplier

(b) Truncated Array Multiplier

Figure 3.1: Standard and Truncated Array Multipliers



(a) Standard Dadda Tree Multiplier

(b) Truncated Dadda Tree Multiplier

Figure 3.2: Standard and Truncated Tree (Dadda) Multipliers [9]

3.3 Truncation Schemes for Fixed-Width Multipliers

Several truncation schemes have been developed—all of which involve not generating the complete partial products matrix and then applying some correction scheme to reduce the error due to truncation as well as post-rounding. This subsection examines some of the schemes which currently exist for parallel multipliers.

Constant Correction Truncation Scheme

In [18], Schulte and Swartzlander, Jr. presents a technique for parallel multiplication which computes the product of two numbers by summing only the most significant columns of the multiplication matrix, along with a correction constant. This correction constant is chosen such that average and mean square errors, with respect to the full-width multiplication, are minimized.

In the conventional parallel full-width multiplier, n^2 partial product bits are summed to produce the final $2n$ bit product. As mentioned before, the fixed-width multiplier is formed by rounding the $2n$ result to n bits.

Substantial hardware savings can be achieved by truncated multiplication, where only the $n+k$ most significant columns of the partial products matrix are summed. Truncated multiplication involves two sources of error, namely, reduction error and rounding error. Reduction error results from summing the partial products matrix without the $n-k$ least significant columns. Rounding error occurs because the product is rounded to n bits. To compensate for these two sources of errors, a correction constant is added to the $n+k$ most significant columns of the partial products matrix, as shown in Figure 3.3. This is an improvement over Y.C. Lim's constant correction methods presented in [19]. In this paper reduction error and rounding error are treated separately, resulting in a poorly selected correction constant. Also, the constant is allowed to take on arbitrary values, which is unfavourable for practical implementations. The correction constant should be limited to the $n+k$ most significant columns.

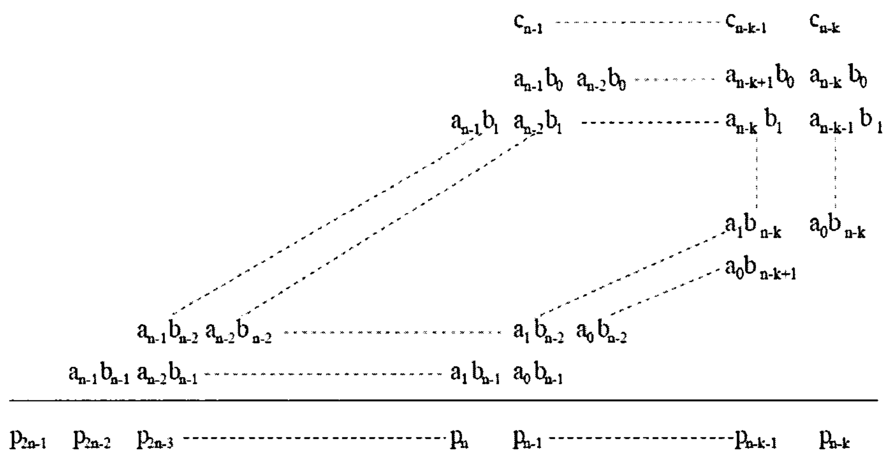


Figure 3.3: Truncated Partial Products Matrix with Constant Correction

The value of the computed product can be expressed in the following way:

$$P' = P + E_{reduct} + E_{round} + C,$$

where P is the true product, E_{reduct} and E_{round} are the reduction and rounding errors, and C is the correction constant. To minimize the average error of the truncated multiplication, or $P'-P$, the correction constant is selected to be as close as possible to the negative of the expected value of the sum of the reduction error and the rounding error. Assuming that the probability of any input bit, a_i or b_j, being a one is 0.5, and a partial product bit, 0.25, the following formula can be used in determining the correction constant, C [18]:

$$C = -\frac{round(2^{n+k} \cdot E_{total})}{2^{n+k}},$$

$$\text{where } E_{total} = -\frac{1}{4}\sum_{q=0}^{n-k-1}(q+1)\cdot 2^{-(2n-q)} - 2^{-(n+1)}\cdot(1-2^{-k})$$

Using exhaustive simulation, error statistics have been determined for multipliers of size n = 8, and 16 bits. Average error, variance and maximum error have been tabulated, as shown in Table 3.1. As can be seen, as k decreases, errors generally tend to increase.

Table 3.1: Error Statistics for Constant Correction Multipliers

n	k	E_{avg}	Variance	E_{max}
8	1	-9.766×10^{-4}	0.1667	2.5039
	2	6.152×10^{-2}	0.1040	1.2539
	3	6.152×10^{-2}	0.0903	0.7539
	4	-1.660×10^{-2}	0.0842	0.6289
	5	-9.766×10^{-4}	0.0834	0.5352
	8	1.953×10^{-3}	0.0833	0.5000
16	1	-3.815×10^{-6}	0.2917	5.5000
	2	6.250×10^{-2}	0.1354	2.7500
	3	6.250×10^{-2}	0.0983	1.5000
	4	-1.563×10^{-4}	0.0861	1.0000
	5	-3.815×10^{-6}	0.0839	0.7188
	16	7.629×10^{-6}	0.0833	0.5000

As described earlier, parallel multipliers are usually implemented as array or tree (column compression) multipliers. Conventional n x n multipliers require n^2 AND gates, $n^2 - 2n$ full adders and n half adders. If the least significant $t = n\text{-}k$ columns are omitted from computation then hardware savings can be approximated as [18]:

$$\frac{t(t+1)}{2} \text{ AND gates, } \frac{(t-1)(t-1)}{2} \text{ Full adders, } (t-1) \text{ Half adders.}$$

A typical n x n bit Dadda multiplier requires n^2 AND gates, n^2-4n+3 full adders and n-1 half adders (for n>2). Similarly the hardware saved with a truncated Dadda multiplier (t>1) is [18]:

$$\frac{t(t+1)}{2} \text{ AND gates, } \frac{(t-1)(t-2)}{2} \text{ Full adders}$$

The following table (Table 3.2), taken from [18], shows hardware savings for various sizes of truncated multipliers with respect to a conventional multiplier utilizing true rounding. This data is calculated based on the assumption that relatives sizes of AND gates, half adders and full adders are 1, 4 and 9, respectively. Complexity savings are slightly higher for Dadda multipliers. As expected, a small value of *k* results in larger complexity savings.

Table 3.2: Complexity Savings From Truncation of Array and Dadda Multipliers

n	*k*	% Complexity Savings (Array)	% Complexity Savings (Dadda)
8	1	35.4	41.8
	2	23.9	28.8
	3	15.2	18.6
	4	9.28	11.9
	5	4.36	6.14
	8	0.00	0.00
16	1	42.6	46.6
	2	36.6	40.0
	3	31.0	34.2
	4	26.2	29.3
	5	21.7	24.2
	16	0.00	0.00

Data-Dependent (Variable) Correction Truncation Scheme

In the constant correction method for truncated multiplication, the correction term does not depend on the values of the bits in the truncated portion of the partial products matrix. Potentially, this could lead to relatively high errors, in the case that the all or the majority of truncated bits are a zero or a one.

In [20], King and Swartzlander, Jr. presents a correction method that uses the information from the partial products bits of the column adjacent to the truncated LSB.

This results in a variable correction term, which can further minimize distortion to the result.

In the method of constant correction, the maximum error occurs when truncated bits (columns $n+k+1$ and beyond) are all zeros or all ones. If the truncated bits are all zeros, then the final error with respect to the full-width multiplier would be equal to the correction value. In this case, ideally, the correction value should be set to zero. If the $n+k+1$ column contains the same number of ones as zeros, then the constant proposed in by Schulte and Swartzlander in [18] should be used. Finally, if the $n+k+1$ column contains all ones, the correction value should be changed to a maximum value. King and Swartzlander use the number of partial products available in the $n+k+1$ column. The correction term is simply added as a "Carry-in" to the $n+k$ column, as shown in Figure 3.4.

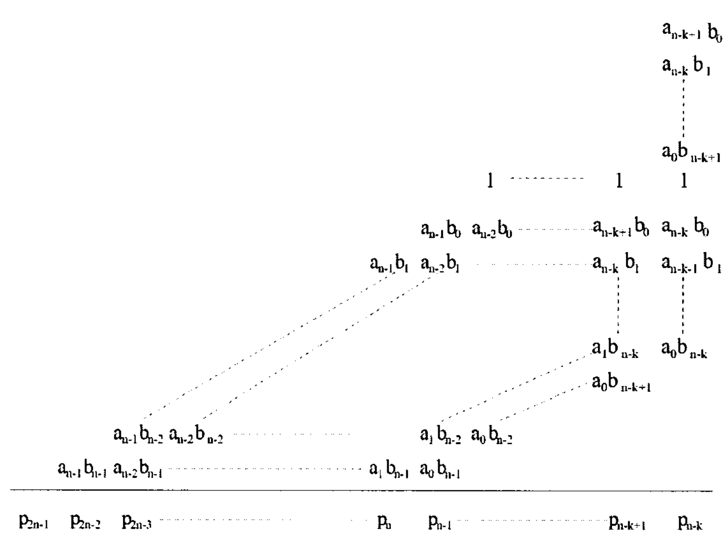


Figure 3.4: Truncated Partial Products Matrix with Data-Dependent Correction

Results show that mean error, maximum error and variance are improved with respect to constant correction. Variable correction scheme is more readily applied to array multipliers, while constant correction schemes are more suitable for tree multipliers [9].

Other Truncation Schemes

Since the work of Swartzlander, Lim, Schulte, and King in the late 1990s, there have been several new schemes, fundamentally based on the concepts of constant and variable correction, presented in literature.

A correction algorithm is developed in [21] by Jou et al. where the partial products in the most significant column of the truncated portion of the matrix are summed together. From this sum, a correction constant is calculated that approximates the sum of the dropped partial products. The method improves error over traditional constant correction, however the implementation is based on a ripple architecture that is slow in speed and consumes much power [15].

In [22], Van et al. propose a fixed-width multiplier architecture that is similar to the constant correction method, where a constant is added to the remaining partial products matrix after truncation. The correction factor, however, is not based on the sum of the most significant column of the truncated portion, but rather is a function of the single partial products of the subset. Only signed multipliers are considered. Implementation of the error-compensation function is based on ripple architecture as well.

In [15] Strollo et al. presents a new error-compensation network for fixed-width multipliers, consisting of two summation trees which are optimally chosen in order to minimize either mean-square error or the maximum absolute error. Their technique gives better accuracy with respect to previous methods, and implementation of the error correction network requires only a few gates with a tree architecture, and thus is best suited for tree multipliers.

Literature shows that many truncation schemes have been proposed that generally target only array and tree multipliers. The next chapters of this thesis are dedicated to the recursive multiplier, originally presented by Danysh and Swartzlander [3]. It will be shown that this multiplier's hierarchical composition makes it very suitable for fixed-width applications. The concepts of truncation schemes described in this chapter will be extended to this multiplier design, resulting in four novel fixed-width multiplier architectures. The following chapter will provide an overview of the recursive multiplier.

CHAPTER 4

THE RECURSIVE MULTIPLIER

4.1 Overview of the Recursive Multiplication Algorithm

One of the pioneering schemes for "divide and conquer" multiplication was proposed by Karatsuba and Ofman in 1962 [4]. The Karatsuba-Ofman Algorithm (KOA) computes the multiplication of two long integers by executing multiplications and additions on their divided parts.

It is possible to perform multiplication of large numbers in significantly fewer operations than the usual brute-force technique of long multiplication. As discovered by Karatsuba and Ofman, multiplication of two n-digit numbers can be done with a bit complexity (number of single operations of addition, subtraction and multiplication) of less than n^2. The algorithm can be illustrated with the following example [24], using two base X numbers, N_1 and N_2, each consisting of two digits:

$$N_1 = a_0 + a_1 X$$
$$N_2 = b_0 + b_1 X$$

Their product can thus be written as:

$$\begin{aligned} P &= N_1 \cdot N_2 \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0)X + a_1 b_1 X^2 \\ &= p_0 + p_1 X + p_2 X^2 \end{aligned}$$

Now let:

$$q_0 = a_0 b_0$$
$$q_1 = (a_0 + a_1)(b_0 + b_1)$$
$$q_2 = a_1 b_1$$

The term q_1 can then be written in terms of p_0, p_1, and p_2:

$$q_1 = p_1 + p_0 + p_2$$

But, since $p_0 = q_0$ and $p_2 = q_2$, it follows that:

$$\begin{aligned} p_0 &= q_0 \\ p_1 &= q_1 - q_0 - q_2 \\ p_2 &= q_2 \end{aligned}$$

Thus the three digits of p have been evaluated using three multiplications rather than four. When the concept is extended to multi-digit numbers, the trade-off of more additions and subtractions becomes evident.

Danysh and Swartzlander have utilized the fundamentals of KOA in their digital recursive multiplication algorithm presented in [3]. Mathematically, the recursive algorithm is established around the fact that any $2n$ x $2n$ bit multiplication may be carried out through four n x n bit sub-multiplications. Consider two unsigned $2n$-bit operands, the multiplicand $A = A_H \times 2^n + A_L$ and multiplier $X = X_H \times 2^n + X_L$, where the subscripts denote the lower and upper n bits respectively. The multiplication of A by X may then be given by:

$$\begin{aligned} Y &= A \cdot X \\ &= \left(A_H \times 2^n + A_L\right) \cdot \left(X_H \times 2^n + X_L\right) \\ &= A_H \cdot X_H \times 2^{2n} + \left(A_L \cdot X_H + A_H \cdot X_L\right) \times 2^n + A_L \cdot X_L . \end{aligned}$$

Multiplication and addition are thus carried out on the divided components of A and X, similar to the technique used in KOA.

4.2 Recursive Multiplication Architecture

Block diagrams illustrating the same recursive multiplier architecture are shown in Figures 4.1 and 4.2.

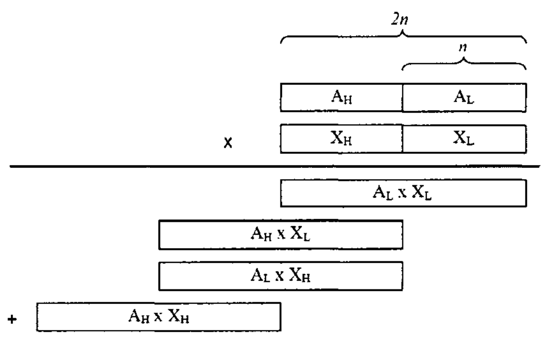


Figure 4.1: Block Diagram of Recursive Multiplier Architecture

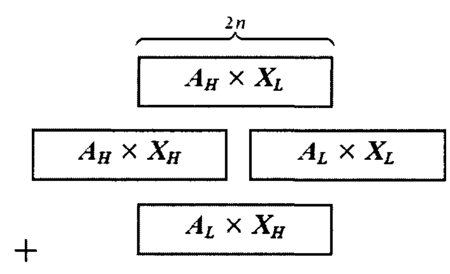


Figure 4.2: Another Block Diagram of Recursive Multiplier Architecture

As can be seen, the overall multiplication may be reduced to four smaller multiplications, and this process may be repeated using even smaller multipliers for the base multipliers. To minimize the resulting reduction delay introduced by subdividing and parallelizing the process, the intermediary products of the sub-multipliers should be kept in carry-save form [5]. In this way, only one final fast adder would be required to yield the final product. A dot diagram, for a typical recursive multiplier with n-bit operands is shown in Figure 4.3.

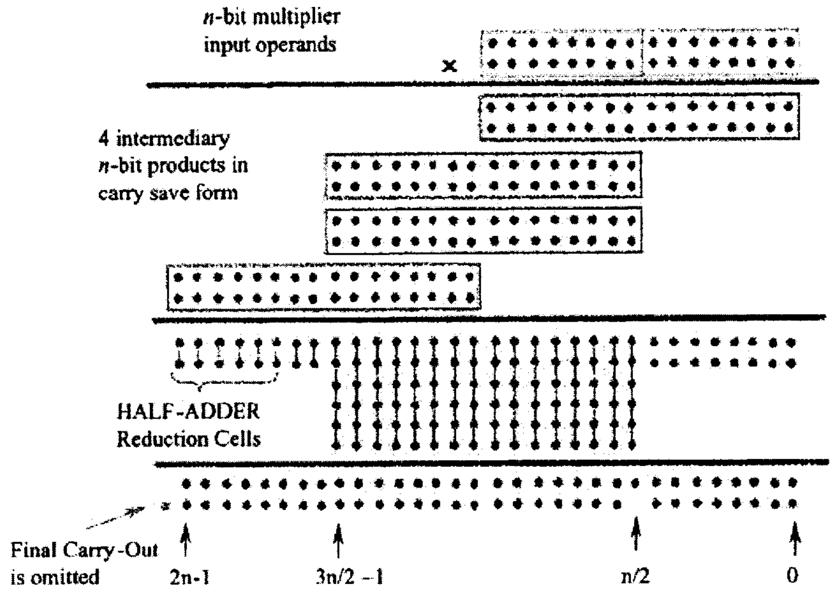


Figure 4.3: Full Dot Diagram of a Recursive Multiplier with n-bit Onput Operands [5]

There are two significant benefits in using the recursive multiplier [3]. Firstly, use of the recursive multiplier allows for a highly regular design and scalability similar to traditional array and modified Booth multipliers. Secondly, unlike array and modified Booth multipliers, the recursive multiplier can achieve a delay of O(log n) similar to fast multipliers such as Dadda and Wallace. Traditional array and modified Booth multipliers are capable of only O(n) delay. Figure 4.4 shows a graph illustrating this delay comparison. The delays for a typical array multiplier, Dadda multiplier and recursive multiplier may be estimated with the following expressions [3]:

$$D_{\text{Array}} = 1 + 3(n-1) + 4\log_2(n-1)$$
$$D_{\text{Dadda}} = 1 + 3(2\log_2(n-1)) + 3\log_2(n+1)$$
$$D_{\text{Recursive}} = 7 + 9\log_2(n-2) + 3\log_2(n+1)$$

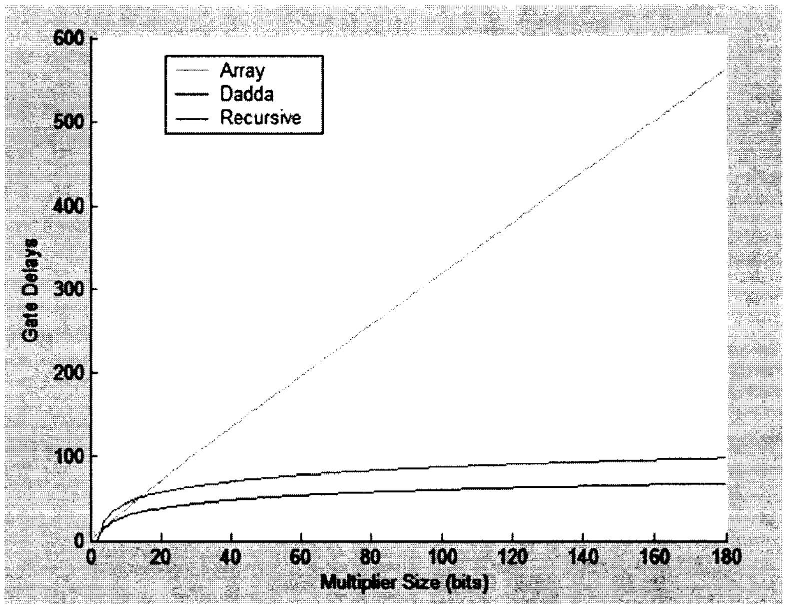


Figure 4.4: Delay Comparison of Array, Dadda and Recursive Multipliers

Essentially, the recursive multiplier reaps the benefits of both worlds: the regularity and scalability of array and Booth multipliers, and the fast performance of Dadda and Wallace tree multipliers. Even with the use of array multipliers as the base multiplier in the recursive hierarchy, a delay of O(log *n*) is achieved. Use of a faster multiplier as the base case can slightly improve performance at the expense of additional complexity and irregularity.

P. Mokrian et al. presented a reconfigurable recursive multiplier architecture that actually outperformed the typical high-performance Booth-recorded Wallace Tree multiplier in terms of delay (17% reduction), dynamic power consumption (20% reduction) and area utilization (12% increase) [5].

The recursive multiplier provides a simple alternative to traditional Booth and array multipliers with speed that is comparable or even faster than Wallace and Dadda multipliers. The recursive hierarchy promotes regularity and allows for short design times. The multiplier is also scalable to higher bit precisions by simply duplicating sub-multipliers and adding additional levels of reduction. A negative aspect of the recursive multiplier is its difficulty in handling 2's complement numbers. However, since we are interested in floating point implementations consisting of fixed-point unsigned integer multipliers, this disadvantage of the recursive multiplier need not be an issue in this study.

After examining the benefits of the recursive multiplier, it was found that the very regular composition of the architecture allows it to be readily applied in systems requiring fixed-width processing. As described before, literature shows that many truncation schemes are available for array and tree multipliers, but none specifically for multipliers based on a recursive architecture. The ensuing chapters will present new truncation schemes that target the recursive multiplier. The standard array multiplier will be used as the base multiplier in all designs, which allows for more convenient complexity calculations and comparisons.

CHAPTER 5

FIXED-WIDTH RECURSIVE MULTIPLIER ARCHITECTURES

The preceding chapter provided an overview of the recursive multiplication algorithm (KOA), as well as the architecture for digital recursive multiplication, presented by Danysh and Swartzlander. The recursive multiplier has an inherent hierarchical structure that consists of several sub-multipliers, making it very suitable for fixed-width applications. Itwill be shown that rather than modifying the sub-multipliers' structure, a truncation scheme can simply remove one sub-multiplier and replace it with a data-dependent correction term.

As mentioned before, fixed-width multipliers have been mainly targeting array and tree structures [9]. Truncation schemes usually involve omitting a certain number of the least significant columns of the partial products matrix and then adding a constant or data-dependent correction term to the truncated partial products matrix to reduce the error due to truncation. Generally, rounding is then applied to the multiplier's output. In this chapter, four new truncation schemes targeting the recursive multiplier are proposed. The associated computation error is analyzed, and a summary of complexity savings incurred as a result of truncation is given as well.

5.1 Proposed Truncation Schemes for Recursive Multipliers

As described before, the overall multiplication in a single-level recursive multiplier is reduced to four smaller sub-multiplications. The product of the multiplicand $A = A_H \times 2^n + A_L$ and multiplier $X = X_H \times 2^n + X_L$ can be written as follows:

$$\begin{aligned} Y &= A \cdot X \\ &= \left(A_H \times 2^n + A_L\right)\cdot\left(X_H \times 2^n + X_L\right) \\ &= A_H \cdot X_H \times 2^{2n} + \left(A_L \cdot X_H + A_H \cdot X_L\right)\times 2^n + A_L \cdot X_L . \end{aligned}$$

Graphically, a fixed-width recursive multiplier can be represented by Figure 5.1. It is clear that the accumulation of four sub-products yields a $4n$ bit result whereas the product (denoted as Y) has only $2n$ bits. In this format, it is evident that the first sub-product, A_LX_L (highlighted), is of minor significance with respect to the rounded $2n$ bit product. The truncation schemes to be presented thus target this particular component.

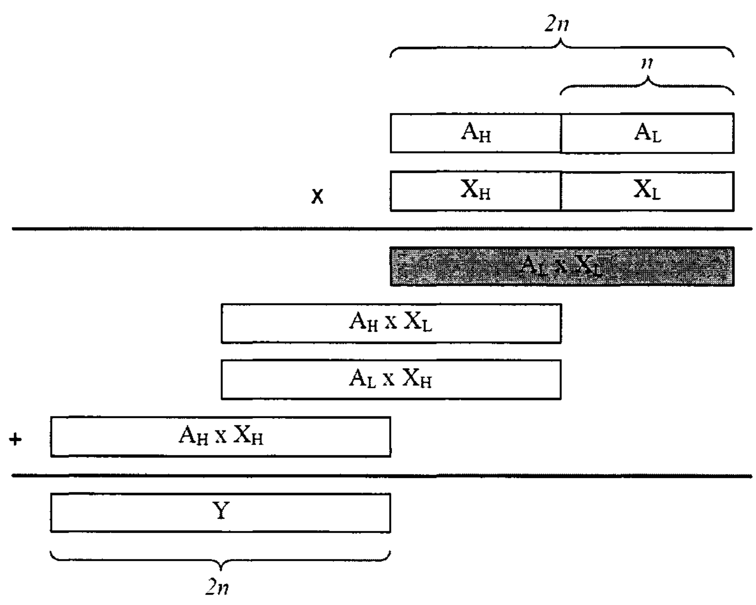


Figure 5.1: Fixed-Width Recursive Multiplier [2]

In all the proposed truncation schemes, the sub-multiplier A_LX_L is removed and subsequently, a data-dependent correction term is added. In the design process of all schemes, it was desirable that the new correction term be relatively easy to generate and, at the same time, maintains some partial information regarding the magnitude of the sub-multiplier, A_LX_L. The proposed truncation schemes are elaborated in the following paragraphs and illustrated in Figures 5.2-5.5. All schemes have a relatively short design time.

In Proposal #1, we simply use A_HX_L or A_LX_H to replace the least significant truncated term A_LX_L. In this fashion, some partial information regarding the magnitude of the partial product is maintained, while no actual multiplication is carried out. The advantage of this scheme lies in the fact that the correction value is a significant term already generated in the calculation, and thus no extra costs are created.

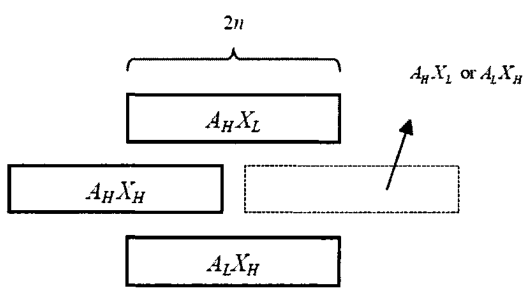


Figure 5.2: Proposal #1

In Proposal #2, the average of the two blocks, A_HX_L and A_LX_H, is placed in the block of A_LX_L after truncation. This approach involves the addition of four rows to the partial product reduction tree of the overall recursive structure, where the rows would be $A_HX_L/2$ and $A_LX_H/2$ in carry save format. This is simply a shifted version of the two previously generated sub-products, thus adding no significant complexity to the architecture. The motivation behind this architecture is that a correction term with a high correlation with the truncated term, A_LX_L, is provided.

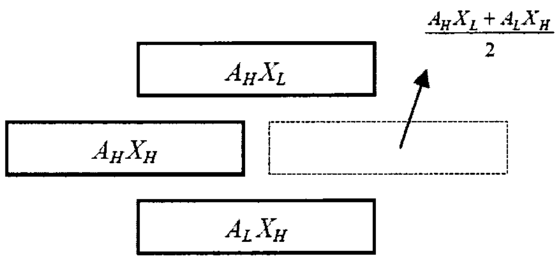


Figure 5.3: Proposal #2

In Proposal #3, the most significant partial product bit, namely $a_{n-1}x_{n-1}$, generated by the block $A_L X_L$, is added at the least significant bit position of block $A_H X_H$. Once again, the aim is to maintain some partial information regarding the magnitude of the partial product without carrying out a full multiplication. The correction bit is simply implemented with one two-input AND gate. With a 1-bit correction term, accumulation of the partial products matrix is simplified, thus requiring less reduction circuitry.

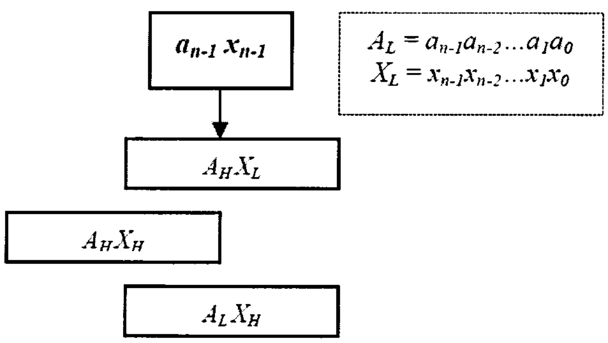


Figure 5.4: Proposal #3

Proposal #4 is essentially an extension of the scheme in Proposal #3, allowing additional correction bits to be used for further error correction. A significant difference, however, is that the first correction bit, $a_{n-1}x_{n-1}$, is added at weight 2^{2n-1}, which is simply the most significant bit position of the truncated sub-multiplier, $A_L X_L$. Additional correction bits, $a_{n-2}x_{n-2}$, $a_{n-3}x_{n-3}$, …, a_0x_0, are added to positions right of the first bit. Mathematically, the correction term with d correction bits, $1 \le d \le n$, can be defined as:

$$C_d = c_{2n-1}^{(d)} c_{2n-2}^{(d)} \cdots c_0^{(d)},$$
$$\text{where } c_i^{(d)} = \begin{cases} a_{i-n}x_{i-n}, & 2n-d \le i \le 2n-1 \\ 0, & 0 \le i \le 2n-d-1. \end{cases}$$

For example, when $d = 2$, the correction term C_2 contains only two bits and has a value of $C_2 = a_{n-1}x_{n-1}2^{2n-1} + a_{n-2}x_{n-2}2^{2n-2}$. Each correction bit can be easily implemented with one two-input AND gate. Similar to the previous scheme, this method allows for a simplified partial products reduction stage.

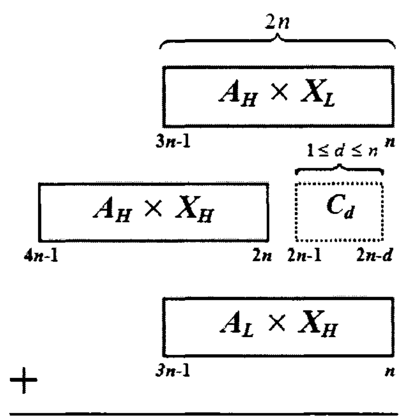


Figure 5.5: Proposal #4

5.2 Error Simulation and Analysis

To determine some of the error statistics associated with each proposed truncation scheme, exhaustive simulations were carried out for a fixed-width recursive integer multiplier as part of a floating point multiplication system. The C code for all simulation programs are provided in Appendix A. Error statistics are tabulated in Tables 5.1 and 5.2. Table 5.1 shows results for the following cases: original full-width multiplier, removal of A_LX_L, Proposals #1 through #3, and Proposal #4 with one and then n

correction bits. Simulations were carried out for three sizes of multipliers, namely $2n$ = 6, 8 and 10. Table 5.2 shows error statistics of the proposed schemes along with those of some tree/array multiplier-based truncation schemes, such as constant and variable correction.

Table 5.1: Error Statistics for Proposed Fixed-Width Recursive Multipliers

$2n$	Correction Method	E_{avg}	E^{+}_{max}	E^{-}_{max}	σ^2_E
6	Full-width (With $A_L X_L$)	0.000	0.500	-0.500	0.083
	Removal of $A_L X_L$	-0.191	0.500	-1.266	0.128
	Proposal #1	-0.000	1.141	-1.156	0.128
	Proposal #2	0.037	0.875	-0.906	0.109
	Proposal #3	0.059	1.250	-0.828	0.173
	Proposal #4 w/ 1 correction bit	-0.067	0.750	-0.828	0.102
	Proposal #4 w/ n=3 correction bits	0.025	0.750	-0.688	0.098
8	Full-width (With $A_L X_L$)	0.000	0.500	-0.500	0.083
	Removal of $A_L X_L$	-0.220	0.500	-1.379	0.128
	Proposal #1	-0.004	1.316	-1.320	0.133
	Proposal #2	0.014	0.938	-1.137	0.113
	Proposal #3	0.030	1.250	-0.910	0.167
	Proposed w/ 1 correction bit	-0.095	0.750	-0.910	0.101
	Proposed w/ $n = 4$ correction bits	0.014	0.750	-0.719	0.095
10	Full-width (With $A_L X_L$)	0.000	0.500	-0.500	0.083
	Removal of $A_L X_L$	-0.235	0.500	-1.438	0.130
	Proposal #1	-0.001	1.407	-1.408	0.136
	Proposal #2	0.005	0.967	-1.253	0.114
	Proposal #3	0.015	1.250	-0.954	0.165
	Proposed w/ 1 correction bit	-0.110	0.750	-0.954	0.101
	Proposed w/ $n = 5$ correction bits	0.008	0.750	-0.734	0.094

Table 5.2: Error Statistics for Different Fixed-Width Multipliers

$2n$	Multiplier Type	Correction Method	E_{avg}	E^+_{max}	E^-_{max}	σ^2_E
6, 8	Rounded full-width multiplier		0.000	0.500	-0.500	0.083
6	Tree or Array	Constant [18]	-0.06	3	-2	0.2
		Variable [20]	0.06	1.4	-0.9	0.1
	Recursive	Removal of A_LX_L	-0.191	0.500	-1.266	0.128
		Proposal #4 w/ 1 correction bit	-0.067	0.750	-0.828	0.102
		Proposal #4 w/ 3 correction bits	0.025	0.750	-0.688	0.098
8	Tree	ROM Max	-0.193	1.316		N/A
		Dual Tree (type 1) [15]	0.122	1.512		N/A
	Recursive	Proposed w/ 4 correction bits	0.014	0.750		0.095

Table 5.1 shows that all truncation schemes provide some degree of error correction. Generally, all schemes lower the average error of the fixed-width multiplier. More specifically, Proposal #1 offers the lowest average error, but relatively larger maximum negative and positive errors. Proposal #2 provides the second best variance of error and relatively low maximum and average errors. Proposal #3 offers an average error that is comparable to others but with a relatively high variance of error. Proposal #4 (with n correction bits) offers the best average error, lowest maximum errors, and lowest variance of errors. Overall, Proposal #4 exhibits better error statistics than the other three schemes. Use of additional correction bits further improves statistics. The maximum positive error remains at 0.75, and additional correction bits reduces the maximum negative error as well as variance of error. For all schemes, average error and variance of error tend to decrease as the size of the multiplier increases, while maximum errors increase slightly. Comparable or better error statistics are expected for larger values of n.

Table 5.2 shows that the proposed fixed-width recursive multiplier based on Proposal #4 truncation scheme has a lower average error, maximum error and variance of error than multipliers found in literature. The other proposed fixed-width multipliers (Proposals 1 through 3) also compare well with these multipliers.

Mathematical analysis of Proposal #4 truncation scheme has proven to be helpful in discovering further some important properties regarding maximum positive and negative errors. The analysis is given below:

It is clear that the term $A_L X_L$ is approximated by the correction expression C_d:

$$A_L X_L = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(a_i x_j)\cdot 2^{i+j} \approx C_d = \sum_{k=1}^{d} a_{n\text{-}k} x_{n\text{-}k} 2^{2n\text{-}k}.$$

A normalized error function, e(n,d), can thus be defined such that:

$$e(n,d) = \frac{1}{2^{2n}}(C_d - A_L X_L) = \sum_{k=1}^{d} a_{n\text{-}k} x_{n\text{-}k} 2^{\text{-}k} - \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(a_i x_j)\cdot 2^{i+j-2n}.$$

When d = 1, the error function e(n,1) is given by:

$$e(n,1) = a_{n-1} x_{n-1} 2^{-1} - \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(a_i x_j)\cdot 2^{i+j-2n}.$$

Then range of $e(n,1)$ can consequently be shown below as:

$$-\left(0.5 - \frac{3}{2^{n+1}} + \frac{1}{2^{2n}}\right) \le e(n,1) \le 0.25.$$

Thus, Proposal #4 truncation scheme with one correction bit would introduce a maximum positive error of 0.25 and a maximum negative error of $-\left(0.5 - \frac{3}{2^{n+1}} + \frac{1}{2^{2n}}\right)$. Considering that a maximum error of ± 0.5 is introduced by final rounding, the fixed-width multiplier using the proposed truncation scheme with one correction bit therefore has a maximum positive error of 0.75 and a maximum negative error of $1-\frac{3}{2^{n+1}}+\frac{1}{2^{2n}}<1$. It can be seen that maximum positive error is independent of the multiplier size. Also the maximum negative error is always less than 1 for any multiplier size. Simulation results show that additional correction bits reduce this negative error.

5.3 Complexity Analysis

Architectural estimations for complexity savings were carried out for each of the proposed fixed-width multiplier designs. Tables 5.3 and 5.4 show complexity savings incurred for multipliers of sizes $2n$ = 8, 16 and 32 bits. Calculations are made assuming that the array multiplier is used as the base multiplier in the recursive architectures.

The complexity of a truncation scheme consists of three parts: the base multipliers' complexity, complexity in generating the correction term, and complexity of the reduction circuits. It is assumed that 4-bit, 8-bit and 16-bit array sub-multipliers are for multipliers of size $2n$ =8, 16 and 32, respectively. For a k-bit array multiplier, the gate count can be estimated by:

$$G_{array(k)} = k^2 + 12(k-2)(k-1) + 4(k-1),$$

where one full adder is estimated as 12 gates and one half-adder as 4 gates [25].

We can take Proposal #4 truncation scheme with one correction bit for an 8-bit multiplier as an example. The gate count for three base multipliers is $3G_{array(4)} = 300$. Generation of one correction bit requires one gate. The complexity for the reduction stage can be estimated as 12 full adders and 5 half adders, which is 164 gates. The total gate count for the proposed truncation scheme with one correction bit is thus 300+1+164=165.

Table 5.3: Complexity Savings Comparison of Each Scheme ($2n = 8$)

Correction Method	Complexity (# of gates)	Complexity Savings (%)
Original (With A_LX_L)	596	—
Removal of A_LX_L	452	24.16
Proposal #1	496	16.78
Proposal #2	584	2.01
Proposal #3	461	22.65
Proposal #4 w/ 1 correction bit	465	21.98
Proposal #4 w/ $n = 4$ correction bits	500	16.11

Table 5.4: Complexity Savings Comparison for Larger Multipliers ($2n$ =16, 32, 64)

$2n$	Original	Proposal #1		Proposal #2		Proposal #3		Proposal #4 w/ n correction bits	
	No. of Gates	No. of Gates	Percent. Savings	No. of Gates	Percent. Savings	No. of Gates	Percent. Savings	No. of Gates	Percent. Savings
16	3196	2600	18.65	2884	9.76	2452	23.23	2575	19.42
32	12956	10120	21.89	10692	17.47	9812	24.27	10220	21.11
64	52444	40136	23.47	41284	21.28	39508	24.66	40832	22.14

From the complexity estimations, it can be seen that savings can potentially reach 25% as n becomes larger for all truncation schemes. More specifically, it can be seen from Table 5.3 that Proposal #4 with n and then 1 correction bits have similar complexity savings as Proposal #1 and Proposal #3, respectively. Proposal #2's low complexity savings for smaller multipliers is due to the fact that the scheme involves addition of two more rows to the partial product matrix, thus increasing the circuitry required for reduction.

To briefly summarize the overall performance of each proposed fixed-width recursive multiplier, Table 5.5 has been created to compare the relative error statistics and complexity savings for each truncation scheme. Simple scores for error statistics and complexity savings were assigned to each scheme based on the results in previous tables. The performance scoring is as follows: 1 = Satisfactory, 2 = Good, 3 = Best.

Table 5.5: Overall Performance Comparison of Proposed Truncation Schemes

Truncation Scheme	Error Statistics Performance Score	Complexity Savings Performance Score
Proposal #1	1	2
Proposal #2	2	1
Proposal #3	1	3
Proposal #4 /w 1 correction bit	2	3
Proposal #4 /w *n* correction bits	3	2

This chapter has provided an in-depth study of new truncation schemes targeting recursive multipliers. All proposed schemes are relatively easy to implement and require short design times. The presented error statistics and complexity comparisons can aid one in selecting the reduced hardware truncation scheme that is best suited for a given application.

It should be noted that architectural complexity savings which have been estimated mathematically cannot always be used as a true metric of multiplier performance. To determine performance characteristics such as propagation delay and power consumption, it is necessary to implement the designs in hardware and carry out simulations. Hardware implementation is presented in the subsequent chapter.

CHAPTER 6

HARDWARE IMPLEMENTATION

To further assess the performance characteristics of the proposed fixed-width recursive multiplier architectures, valid models must be created for each design, and then compared against a model of original full-width recursive multiplier. Multipliers of sizes 16 and 32 bits have been modelled and implemented in Altera Stratix EP1S10F484C5 Field Programmable Gate Array (FPGA).

Since several designs needed to be implemented, FPGA technology was the most feasible method of hardware implementation. A major advantage of FPGAs over ASIC (application specific integrated circuit) designs is their rapid-prototyping capabilities [26]. FPGA implementation allowed for the following performance comparisons to be made between the proposed architectures: propagation delay, power consumption, and complexity in terms of logic elements (LEs).

This chapter begins with a description of the hierarchical design of the multiplier using Verilog Hardware Description Language (HDL). Simulation results and performance comparison of architectures are the subsequent topics of discussion.

6.1 HDL Model

Verilog is a hardware description language capable of describing digital design as a set of modules which can become building blocks forming a complete system. This hierarchical design methodology was followed in modelling the proposed fixed-width multiplier architectures.

All Verilog codes were synthesized for Stratix EP1S10F484C5 FPGA using Altera Quartus II software. As an example, RTL schematic of a "32-bit fixed-width recursive multiplier using Proposal #4 truncation scheme (16 correction bits)" is shown in Figure 6.1. Example Verilog code for this design has been provided in Appendix B, and important synthesis and simulation reports are in Appendix C. The four main components of the architecture are the base multipliers, which provide intermediary products, the data-dependent correction block, and the reduction block, which provides the final fixed-width product.

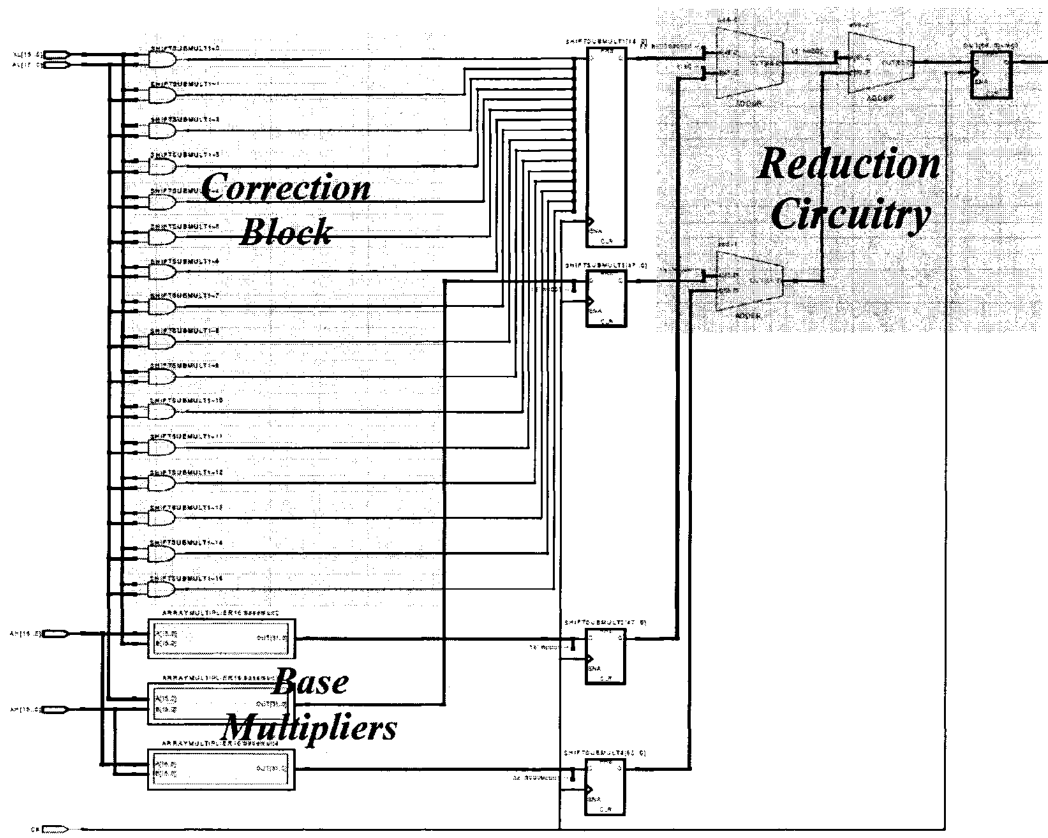


Figure 6.1: RTL Schematic Diagram of a "32-bit Fixed-Width Recursive Multiplier using Proposal #4 (16 correction bits)"

6.2 Simulation Results

Hardware simulation was carried out to measure propagation delay, dynamic power consumption, and complexity. Timing Analyzer and PowerPlay Analyzer tools in Altera Quartus II were utilized. Results for these metrics have been tabulated (Table 6.1). Reductions in delay, power and complexity with respect to the original recursive multiplier have been calculated. Additionally, dynamic power consumption in terms of mW/MHz has been calculated, which is essentially a power-delay-product (PDP) metric.

Table 6.1: FPGA Simulation Results

2*n*	Correcton Method	Delay (ns)	[redacted]	Dynamic Power (mW)	[redacted]	PDP (mW/MHz)	Complexity (LEs)	[redacted]
16	Original	4.245	—	375.76	—	1.595	698	—
	Remove $A_L x_L$	4.234	0.259	323.55	13.89	1.370	517	25.9
	Proposal #1	4.388	-3.369	338.23	9.98	1.484	534	23.5
	Proposal #2	4.643	-9.375	339.45	9.66	1.576	565	19.1
	Proposal #3	4.256	0.259	332.44	11.53	1.415	524	24.9
	Proposal #4 with $n = 8$ corr. bits	4.258	0.306	336.65	10.41	1.433	536	23.2
32	Original	5.116	—	529.79	—	2.710	2941	—
	Remove $A_L x_L$	4.686	8.405	432.27	17.31	2.026	2202	25.13
	Proposal #1	5.139	-0.450	470.55	10.57	2.418	2250	23.50
	Proposal #2	5.389	-5.336	472.86	10.11	2.548	2346	20.23
	Proposal #3	4.750	7.154	466.16	12.01	2.214	2210	24.86
	Proposal #4 with $n = 16$ corr. bits	4.770	6.763	470.36	11.60	2.244	2248	23.56

From Table 6.1, it can be seen that all designs achieved a reduction in propagation delay, with the exception of multipliers using Proposal #1 and #2 truncation schemes. For larger multipliers, using Proposal #3 and Proposal #4 can result in a delay reduction of almost 7%. At the same time, reduction in dynamic power consumption can reach 12%. Multipliers with Proposals #3 and #4 exhibit the lowest PDP (mW/MHz). Complexity savings incurred in FPGA implementation match well with the architectural estimates made earlier. Savings can potentially reach 25% as n increases.

CHAPTER 7

FIXED-WIDTH MULTI-LEVEL RECURSIVE MULTIPLIERS

As seen in Chapters 4 and 5, the recursive multiplier has an inherent hierarchical structure that consists of several sub-multipliers, making it suitable for fixed-width applications. Rather than modifying the sub-multipliers' structure, a truncation scheme simply removes one sub-multiplier and replaces it with a data-dependent correction term to minimize computational error due to truncation. An apparent advantage of using a fixed-width recursive multiplier is that no design change is needed for the structure's sub-multiplier components. In this chapter previous work is extended to multi-level recursive architectures and new truncation schemes for multi-level recursive multipliers are presented. Error analysis and complexity savings for the multi-level recursive structure are also discussed.

7.1 Multi-Level Recursive Multiplication

Single-level recursive multiplication may be further broken down into smaller sub-multipliers, which compute in parallel. The relationship between a positive integer number of levels of recursion, k, the overall size of the multiplier, a, and the size of the sub-multipliers, b, may be given by:

$$k = \log_2\left(\frac{a}{b}\right)$$

For example, a 64-bit multiplier may be composed of four 32-bit sub-multipliers ($k = 1$), sixteen 16-bit sub-multipliers ($k = 2$), etc. For convenience in describing multi-level recursive multiplication, let two unsigned $\left(2^k \times n\right)$-bit operands be used for a k-level recursive structure. Consider the case of a two-level recursive multiplier where the operands can be given by:

$$A = (a_{4n-1}a_{4n-2}\cdots a_0) = A_1^{(1)}\cdot 2^{2n} + A_0^{(1)} = A_3^{(2)}\cdot 2^{3n} + A_2^{(2)}\cdot 2^{2n} + A_1^{(2)}\cdot 2^{n} + A_0^{(2)}, \text{ and}$$

$$X = (x_{4n-1}x_{4n-2}\cdots x_0) = X_1^{(1)}\cdot 2^{2n} + X_0^{(1)} = X_3^{(2)}\cdot 2^{3n} + X_2^{(2)}\cdot 2^{2n} + X_1^{(2)}\cdot 2^{n} + X_0^{(2)},$$

where $A_i^{(1)}$ and $X_i^{(1)}$, $i = 0,1$, are components of $2n$ bits each and $A_i^{(2)}$ and $X_i^{(2)}$, $i = 0,1,2,3,$ are components of n bits each. The superscript of a term indicates at which recursive level it is generated. It follows that the resultant product of A and X with two levels of recursion is:

$$\begin{aligned}
Y &= A\cdot X \\
&= A_1^{(1)}X_1^{(1)}\cdot 2^{4n} + (A_1^{(1)}X_0^{(1)} + A_0^{(1)}X_1^{(1)})\cdot 2^{2n} + A_0^{(1)}X_0^{(1)} \\
&= \left(A_3^{(2)}X_3^{(2)}\cdot 2^{2n} + (A_3^{(2)}X_2^{(2)} + A_2^{(2)}X_3^{(2)})\cdot 2^{n} + A_2^{(2)}X_2^{(2)}\right)\cdot 2^{4n} \\
&\quad + \left(A_3^{(2)}X_1^{(2)}\cdot 2^{2n} + (A_3^{(2)}X_0^{(2)} + A_2^{(2)}X_1^{(2)})\cdot 2^{n} + A_2^{(2)}X_0^{(2)}\right)\cdot 2^{2n} \\
&\quad + \left(A_1^{(2)}X_3^{(2)}\cdot 2^{2n} + (A_1^{(2)}X_2^{(2)} + A_0^{(2)}X_3^{(2)})\cdot 2^{n} + A_0^{(2)}X_2^{(2)}\right)\cdot 2^{2n} \\
&\quad + \left(A_1^{(2)}X_1^{(2)}\cdot 2^{2n} + (A_1^{(2)}X_0^{(2)} + A_0^{(2)}X_1^{(2)})\cdot 2^{n} + A_0^{(2)}X_0^{(2)}\right).
\end{aligned}$$

7.2 Proposed Truncation Scheme for Multi-Level Recursive Multipliers

For fixed-width multiplication, the product must remain as $4n$ bits, which is the size of the input operands. Thus, the truncated components should include all the terms in the above equation whose most significant bit has a weight of less than 2^{4n}. For convenience, the above equation can be re-written as:

$$
\begin{aligned}
Y &= A \cdot X \\
&= A_1^{(1)} X_1^{(1)} \cdot 2^{4n} + (A_1^{(1)} X_0^{(1)} + A_0^{(1)} X_1^{(1)}) \cdot 2^{2n} + A_0^{(1)} X_0^{(1)} \\
&= A_1^{(1)} X_1^{(1)} \cdot 2^{4n} + \left(A_3^{(2)} X_1^{(2)} \cdot 2^{2n} + (A_3^{(2)} X_0^{(2)} + A_2^{(2)} X_1^{(2)}) \cdot 2^{n} + A_2^{(2)} X_0^{(2)} \right) \cdot 2^{2n} \\
&\quad + \left(A_1^{(2)} X_3^{(2)} \cdot 2^{2n} + (A_1^{(2)} X_2^{(2)} + A_0^{(2)} X_3^{(2)}) \cdot 2^{n} + A_0^{(2)} X_2^{(2)} \right) \cdot 2^{2n} + A_0^{(1)} X_0^{(1)} \\
&= A_1^{(1)} X_1^{(1)} \cdot 2^{4n} + (A_3^{(2)} X_1^{(2)} + A_1^{(2)} X_3^{(2)}) \cdot 2^{4n} + (A_3^{(2)} X_0^{(2)} + A_2^{(2)} X_1^{(2)} + A_1^{(2)} X_2^{(2)} + A_0^{(2)} X_3^{(2)}) \cdot 2^{3n} \\
&\quad + \mathbf{A_2^{(2)} X_0^{(2)} \times 2^{2n} + A_0^{(2)} X_2^{(2)} \times 2^{2n} + A_0^{(1)} X_0^{(1)}}.
\end{aligned}
$$

Clearly, the last three terms in bold, $A_2^{(2)} X_0^{(2)} \cdot 2^{2n}, A_0^{(2)} X_2^{(2)} \cdot 2^{2n}$, and $A_0^{(1)} X_0^{(1)}$, should be truncated, as shown graphically in Figure 7.1. For error correction, any of the truncation schemes proposed earlier may be applied. Error simulation and analysis have been carried out with Proposal #4 truncation scheme applied to a two-level recursive multiplier, as shown in the next sub-section.

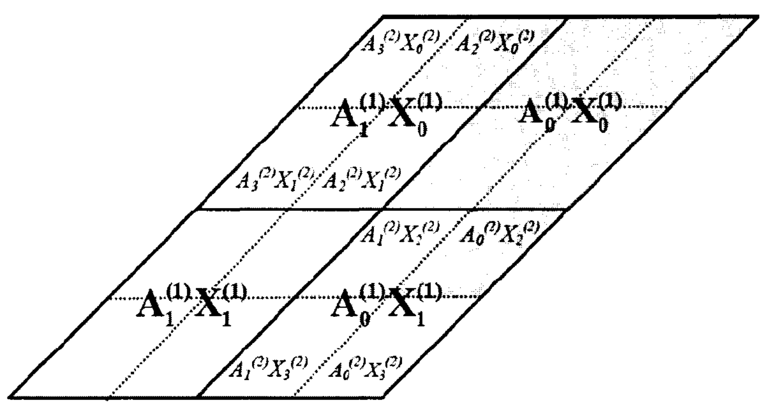


Figure 7.1: Graphical Representation of Truncation in a Two-Level Recursive Multiplier

7.3 Error Simulation and Complexity Analysis

Table 7.1 shows exhaustive error simulation results for fixed-width two-level recursive multipliers of sizes $4n$ = 4, 8 and 16, with no correction (only truncation), 1 correction bit (for each truncated sub-multiplier) and then the maximum number of correction bits. As expected, addition of correction bits reduces error. Overall error is also greater than in the single-level case. Generally, maximum errors become more prominent with larger multipliers, while average error and variance of error decrease. Mathematical analysis describing maximum positive error has been carried out.

Table 7.1: Error Simulation Results for Fixed-Width Two-Level Recursive Multipliers

$4n$	Number of Correction Bits	E_{avg}	E^{+}_{max}	E^{-}_{max}	σ^2_E
4	0	-0.266	0.500	-1.313	0.165
	1	0.086	0.938	-0.688	0.137
	Max.	0.072	0.938	-0.625	0.140
8	0	-0.501	0.500	-2.504	0.208
	1	-0.128	1.109	-1.285	0.136
	Max.	0.094	1.109	-0.961	0.122
16	0	-0.017	0.500	-3.250	0.131
	1	-0.008	1.220	-1.814	0.128
	Max.	0.007	1.220	-1.115	0.116

For the truncated component, $A_0^{(1)}X_0^{(1)} = \sum_{i=0}^{2n-1}\sum_{j=0}^{2n-1} a_i x_j \cdot 2^{i+j}$, define an error correction term (from Proposal #4) $C(d,2n)$ as $C(d,2n) = \sum_{k=1}^{d} a_{2n-k}x_{2n-k} \cdot 2^{4n-k}$, where d, $1 \leq d \leq 2n$, determines the number of bits to be used in the correction term, and $2n$ is the size of the truncated component. $C(d,2n)$ is obviously simpler to compute than carrying out the full multiplication for $A_0^{(1)}X_0^{(1)}$. The normalized error function $e(d,2n)$ for the error due to replacing $A_0^{(1)}X_0^{(1)}$ by $C(d,2n)$ can be defined as follows:

$$e(d,2n) = \frac{1}{2^{4n}}\left(C(d,2n) - A_0^{(1)}X_0^{(1)}\right) = \sum_{k=1}^{d} a_{2n-k}x_{2n-k} \cdot 2^{-k} - \sum_{i=0}^{2n-1}\sum_{j=0}^{2n-1} a_i x_j \cdot 2^{i+j-4n},$$

For the other two truncated terms (smaller multipliers), their error correction terms and error functions are given by:

$$e(d,n)=\frac{1}{2^{4n}}\left(C(d,n)\cdot 2^{2n}-A_2^{(2)}X_0^{(2)}\cdot 2^{2n}\right)=\sum_{k=1}^{d}a_{3n-k}x_{n-k}\cdot 2^{-k}-\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}a_{2n+i}x_j\cdot 2^{i+j-2n}$$

and

$$e(d,n)=\frac{1}{2^{4n}}\left(C(d,n)\cdot 2^{2n}-A_0^{(2)}X_2^{(2)}\cdot 2^{2n}\right)=\sum_{k=1}^{d}a_{n-k}x_{3n-k}\cdot 2^{-k}-\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}a_i x_{2n+j}\cdot 2^{i+j-2n},$$

respectively.

Lemma: The maximal positive value of the error function $e(d,n)$ is not greater than 0.25, or $e(d,n)\le 0.25$ for $1\le d\le n$.

A proof of the lemma follows by expanding $e(d,n)=\sum_{k=1}^{d}a_{n-k}x_{n-k}2^{-k}-\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}a_i x_j 2^{i+j-2n}$.

This was seen previously in error analysis presented in Chapter 5. It can be shown that for k-level recursive structure, the total error due to replacing the truncated terms by the corresponding correction terms is:

$$\sum_{i=1}^{k}2^{i-1}e(d_i,2^{k-i}n)+0.5\le\frac{1}{4}(2^k+1).$$

Maximum post rounding error, ± 0.5, is taken into consideration in the expression. In determining the expression it is assumed that each partial product bit has equal probability of being a one. This maximum error bound is tabulated for different values of k in Table 7.2. Exhaustive simulation results from earlier show that this upper bound is indeed approached as n becomes larger. Also, it is shown in Table 7.1 that absolute value of the maximum negative error can be reduced to a value below the maximum positive error bound for a sufficient number of correction bits, *d*.

Potential complexity savings for different levels of recursion are shown in Table 7.3. A two-level recursive fixed-width multiplier can offer more complexity savings than the single-level case with the expense of increased computation error. The percentage complexity savings for *k*-levels of recursion is found to be approximately:

$$50(1-\frac{1}{2^k}).$$

This expression can be determined intuitively from a diagram graphically showing the truncation pattern. Figure 7.2 graphically shows complexity savings for one, two and three levels of recursion. As the number of levels of recursion increases, it can be seen that the truncation pattern actually tends towards traditional truncation of partial product matrices for tree and array multipliers (approximately 50% complexity savings), shown in Figure 7.3.

Table 7.2: Maximum Positive Error for Different Levels of Recursion

Levels of Recursion (k)	1	2	3	4	5	k
Max. Pos. Error	0.75	1.25	2.25	4.25	8.25	$\frac{1}{4}(2^k+1)$

Table 7.3: Approximate Complexity Savings for Different Levels of Recursion

Levels of Recursion (k)	1	2	3	4	5	k
Approx. Complexity Savings (%)	25.0	37.5	43.8	46.9	48.4	$50(1-\frac{1}{2^k})$

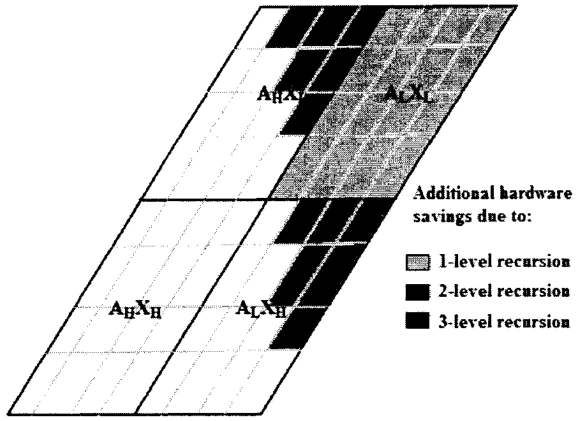


Figure 7.2: Graphical Representation of Complexity Savings for k = 1, 2, and 3

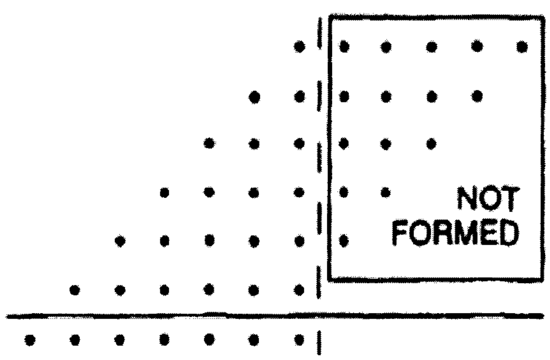


Figure 7.3: Partial Product Matrix Truncation for Tree Multipliers

CHAPTER 8

CONCLUSIONS

8.1 Summary of Contributions

The purpose of this study has been to explore the area of digital multiplication, and more specifically, fixed-width multipliers, which are important components of many DSP systems. This has lead to several contributions to fixed-width digital multiplication architecture and also to the field of computer arithmetic.

Traditionally, fixed-multiplication has been targeting only array and tree multipliers. This thesis has extended the concepts of truncation schemes to the high-performance recursive multiplier, whose inherent hierarchical structure makes it very suitable for fixed-width applications. Four novel fixed-width multipliers based on recursive architectures have been proposed. Error statistics for each design have been determined via exhaustive simulations, and compared with fixed-width multipliers in literature. Mathematical expressions for maximum negative and positive errors have been developed for one of the truncation schemes (Proposal #4). Complexity reduction estimates have been carried out at the architectural level. Additionally, all designs have been implemented in FPGA to determine reduction in delay, power and logic complexity savings with respect to the original full-width recursive multiplier.

Based on work done for fixed-width single-recursive multiplication, novel architectures for fixed-width multi-level recursive multiplication have also been developed. Error and complexity analysis have been carried out. New mathematical expressions describing maximum positive error and complexity savings for a general k-level fixed-width recursive multiplier have been derived.

8.2 Concluding Remarks

New architectures for fixed-width digital recursive multipliers have been developed. The designs have embodied many of the modern requirements of fixed-width multipliers such as low error, complexity, delay and power. A significant advantage of this work is that very little architectural change of the original recursive (single-level or multi-level) multiplier is needed when implementing any of the proposed truncation schemes

Simulation results have shown that the proposed fixed-width multipliers exhibit better error statistics than those found in literature, in terms of average error, maximum positive and negative errors, and variance of error. The designs have also been implemented in Stratix EP1S10F484C5 FPGA. Simulations have shown that delay, power and complexity can be reduced up to 7%, 12% and 25%, respectively, compared to the original full-width recursive multiplier. Proposal #4 truncation scheme has exhibited the best balance of error and performance characteristics. A performance summary of the proposed schemes has been given in Chapter 5 to aid one in determining the truncation scheme best suited for a certain application.

Fixed-width multiplication has also been extended to multi-level recursive multipliers, as shown in Chapter 7. A simple truncation scheme, based on previous schemes for single-level recursive multipliers has been presented. Generally, fixed-width multipliers with higher levels of recursion can offer more complexity savings at the expense of increased computation error. Mathematical expressions for maximum positive error as well as maximum potential complexity savings have been presented for k levels of recursion.

REFERENCES

[1] Michelle A. Hoyle, "The History of Computing Science", http://www.eingang.org /Lecture/index.html, 2002

[2] K. Biswas, P. Mokrian, H. Wu and M. Ahmadi, "Truncation Schemes for Recursive Multipliers", Asilomar Conference on Signals, Systems and Computers, pp. 1177-1180, 2005.

[3] A.N. Danysh and E.E. Swartzlander Jr., "A Recursive Fast Multiplier", Asilomar Conference on Signals, Systems and Computers, vol. 1, pp. 197-201, 1998.

[4] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata", Sov. Phys.-Dokl. (English Translation), vol. 7, pp. 595-596, 1963.

[5] P. Mokrian, "A Reconfigurable Digital Multiplier Architecture". A Master's Thesis, Department of Electrical and Computer Engineering, University of Windsor, April 2003.

[6] G. M. Howard, "Investigation into Arithmetic Sub-Cells for Digital Multiplication". A Master's Thesis, Department of Electrical and Computer Engineering, University of Windsor, April 2005.

[7] Gary W. Bewick, "Fast Multiplication: Algorithms and Implementation", A Ph.D Dissertation, Stanford University, 1994.

[8] Bickerstaff, Schulte and E.E. Swartzlander Jr., "Analysis of column compression multipliers", IEEE Symposium on Computer Arithmetic, pp. 33 -39, 2001

[9] M. Schulte and J. Stine, "Reduced Power Dissipation Through Truncated Multiplication", Technical Manuscript, Electrical Engineering and Computer Science Department, Lehigh University, 1999.

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

[11] L. Dadda, "The evolution of computer architectures", CompEuro '91, European Computer Conference on Advanced Computer Technology, Reliable Systems and Applications. pp. 9-16, 1991

[12] J. Rabaey, "Digital Integrated Circuits: A Design Perspective", Prentice Hall, New Jersey, 2003.

[13] IEEE Standard for Binary Floating-Point, IEEE Std. 754-1985.

[14] B. Parhami, "Computer Arithmetic: Algorithms and Hardware Designs", Oxford University Press, New York, 2000.

[15] A.G.M. Strollo, N. Petra and D. De Caro, "Dual-Tree Error Compensation for High Performance Fixed-Width Multipliers", IEEE Trans. Circuits and Systems—II, August 2005, vol. 52, No. 8, pp. 501-507.

[16] Earl E. Swartzlander, Jr., "Truncated multiplication with approximate rounding", Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, vol. 2, pp. 1480-1483, Oct. 1999.

[17] M. Ercegovac, "Digital Arithmetic", Morgan Kaufmann Publishers, 2003.

[18] M. J. Schulte and E. E. Swartzlander, Jr., "Truncated multiplication with correction Constant", VLSI Signal Processing, VI, New York, IEEE Press, 1993, pp. 1333-1336.

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

[20] E. J. King and E. E. Swartzlander, Jr., "Data-dependent truncation scheme for parallel multipliers", Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, pp. 1178-1182, 1997.

[21] J. M. Jou, S. R.Kuang, and R. D. Chen, "Design of low-error fixed-width multipliers for DSP applications," IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process., vol. 46, no. 6, pp. 836–842, Jun. 1999.

[22] L. Van, S. Wang, and W. Feng, "Design of the lower error fixed-width multiplier and its application," IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process., vol. 47, no. 10, pp. 1112–1118, Oct. 2000.

[23] A. G. M. Strollo, E. Napoli, and D. De Caro, "Direct digital frequency synthesizers using first-order polynomial Chebyshev approximation," in Proc. Eur. Solid-State Circuits Conf. (ESSCIRC'02), Florence, Italy, Sep. 24–26, 2002, pp. 527–530.

[24] Weisstein, Eric W, "Karatsuba Multiplication", From MathWorld--A Wolfram Web Resource, http://mathworld.wolfram.com/KaratsubaMultiplication.html.

[25] J. Kim, and E.E. Swartzlander, Jr., "Improving the Recursive Multiplier",Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, pp. 1320 -1324, 1999.

[26] TimeLogic Biocomputing Solutions, "FPGA Technology", http://www.timelogic.com/technology_fpga.html.

[27] A. Habibi and P.A. Wintz, "Fast Multipliers", IEEE Transactions on Computers, vol. C-19, pp. 153-157, 1970.

[28] M. Mehta, V. Parmar and E. Swartzlander, "High-speed multiplier design using multi-input counter and compressor circuits", IEEE Symposium on Computer Arithmetic, pp. 43-50, 1991.

[29] T. Rhyne and N.R. Strader, "A signed bit-sequential multiplier", IEEE Transactions on Computers, vol. C-35, no. 10, pp. 896-901, 1986.

[30] S. S. Kidambi, F. El-Guibaly, and A. Antoniou, "Area-efficient multipliers for digital signal processing applications," IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process., vol. 43, no. 2, pp. 90–95, Feb.1996.

APPENDICES

APPENDIX A

C Code for Error Simulation Programs

```
/**********************************
EXHAUSTIVE SIMULATION PROGRAM C CODE

Proposal #1 Error Statistics
**********************************/


#include <math.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <time.h>


int test();
int test1();

main ()
{

  float B,C,C1,C2,C3,C4,C5,C22,C33,C55;
  float E0,E1,E2,E3,AE0,AE1,AE2,AE3;
  float MAE0,MAE1,MAE2,MAE3;
  float MIE0,MIE1,MIE2,MIE3;
  float EE0,EE1,EE2,EE3,AEE0,AEE1,AEE2,AEE3;
  float FCI0,FCI1,FCI3,D1,D2,D3;
  int CI0,CI1,CI2,CI3,CII0,CII1;
  int n,i,j,k,ii,jj,kk;

/*
Case1:
      0 <= XH,XL,AH,AL <= 2^n-1
Case2:
      2^{n-1} <= XH,AH <= 2^n-1
            0 <= XL,AL <= 2^n-1
*/

for(n=8;n<9;n++)
{
printf("-------------n=%i---------------\n",n);
  B=1;
  for(i=0;i<n;i++)
    B=B*2;

AE0=0;AE1=0;AE2=0;
AEE0=0;AEE1=0;AEE2=0;
MAE0=0;MAE1=0;MAE2=0;
MIE0=0;MIE1=0;MIE2=0;
  for(XH=0;XH<B;XH++)
  for(AH=0;AH<B;AH++)
  for(XL=0;XL<B;XL++)
  for(AL=0;AL<B;AL++)
   {
C=AH*XH+(AH*XL+AL*XH)/B+AL*XL/B/B;
CI0=C;
CI1=AH*XH+(AH*XL+AL*XH)/B+AL*XL/B/B+0.5;
D1=C-CI0;
if(D1==0.5)
   {
    if(CI0/2*2==CI0) CI1=CI1-1;
```

```
   }
AL1=AL/(B/2);
XL1=XL/(B/2);


C2=AH*XH+(AH*XL+AL*XH)/B+(AH*XL)/B/B;


CI2=AH*XH+(AH*XL+AL*XH)/B+(AH*XL)/B/B + 0.5;
*/

D2=C2-CII0;
if(D2==0.5)
   {
    if(CII0/2*2==CII0) CI2=CI2-1;
   }
/*
printf("C=%f,CI0=%f,CI1=%f,CI2=%f\n",C,CI0,CI1,CI2);
*/
E0=CI0-C;
if(E0>MAE0) MAE0=E0;
if(E0<MIE0) MIE0=E0;
EE0=E0*E0;
E1=CI1-C;
if(E1>MAE1) MAE1=E1;
if(E1<MIE1) MIE1=E1;
EE1=E1*E1;
E2=CI2-C;
if(E2>MAE2) MAE2=E2;
if(E2<MIE2) MIE2=E2;
EE2=E2*E2;
AE0=AE0+E0;
AEE0=AEE0+EE0;
AE1=AE1+E1;
AEE1=AEE1+EE1;
AE2=AE2+E2;
AEE2=AEE2+EE2;
   }
AE0=AE0/B/B/B/B;
AEE0=AEE0/B/B/B/B;
AE1=AE1/B/B/B/B;
AEE1=AEE1/B/B/B/B;
AE2=AE2/B/B/B/B;
AEE2=AEE2/B/B/B/B;
printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var0=%f\n",AE0,MAE0,MIE0,AEE0-AE0*AE0);
printf("(True Round) AE1=%f, MaxE=%f, MinE=%f, Var1=%f\n",AE1,MAE1,MIE1,AEE1-AE1*AE1);
printf("(Trunc.Schm) AE2=%f, MaxE=%f, MinE=%f, Var2=%f\n",AE2,MAE2,MIE2,AEE2-AE2*AE2);
}

}


/*****************************************************************
EXHAUSTIVE SIMULATION PROGRAM EXAMPLE C CODE

Proposal #2 Error Statistics
*****************************************************************/

#include <math.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <time.h>

int test();
int test1();

main ()
{

  float B,C,C1,C2,C3,C4,C5,C22,C33,C55;
```

```
  float E0,E1,E2,E3,AE0,AE1,AE2,AE3;
  float MAE0,MAE1,MAE2,MAE3;
  float MIE0,MIE1,MIE2,MIE3;
  float EE0,EE1,EE2,EE3,AEE0,AEE1,AEE2,AEE3;
  float FCI0,FCI1,FCI3,D1,D2,D3;
  int CI0,CI1,CI2,CI3,CII0,CII1;
  int n,i,j,k,ii,jj,kk;

/*
Case1:
      0 <= XH,XL,AH,AL <= 2^n-1
Case2:
      2^{n-1} <= XH,AH <= 2^n-1
            0 <= XL,AL <= 2^n-1
*/

for(n=8;n<9;n++)
{
printf("-------------n=%i---------------\n",n);
  B=1;
  for(i=0;i<n;i++)
    B=B*2;

AE0=0;AE1=0;AE2=0;
AEE0=0;AEE1=0;AEE2=0;
MAE0=0;MAE1=0;MAE2=0;
MIE0=0;MIE1=0;MIE2=0;
  for(XH=0;XH<B;XH++)
  for(AH=0;AH<B;AH++)
  for(XL=0;XL<B;XL++)
  for(AL=0;AL<B;AL++)
   {
C=AH*XH+(AH*XL+AL*XH)/B+AL*XL/B/B;
CI0=C;
CI1=AH*XH+(AH*XL+AL*XH)/B+AL*XL/B/B+0.5;
D1=C-CI0;
if(D1==0.5)
   {
    if(CI0/2*2==CI0) CI1=CI1-1;
   }
AL1=AL/(B/2);
XL1=XL/(B/2);

C2=AH*XH+(AH*XL+AL*XH)/B+(AH*XL+AL*XH)/2/B/B;

CI2=AH*XH+(AH*XL+AL*XH)/B+(AH*XL+AL*XH)/2/B/B + 0.5;
*/

D2=C2-CII0;
if(D2==0.5)
   {
    if(CII0/2*2==CII0) CI2=CI2-1;
   }
/*
printf("C=%f,CI0=%f,CI1=%f,CI2=%f\n",C,CI0,CI1,CI2);
*/
E0=CI0-C;
if(E0>MAE0) MAE0=E0;
if(E0<MIE0) MIE0=E0;
EE0=E0*E0;
E1=CI1-C;
if(E1>MAE1) MAE1=E1;
if(E1<MIE1) MIE1=E1;
EE1=E1*E1;
E2=CI2-C;
if(E2>MAE2) MAE2=E2;
if(E2<MIE2) MIE2=E2;
EE2=E2*E2;
AE0=AE0+E0;
AEE0=AEE0+EE0;
AE1=AE1+E1;
```

```
AEE1=AEE1+EE1;
AE2=AE2+E2;
AEE2=AEE2+EE2;
   }
AE0=AE0/B/B/B/B;
AEE0=AEE0/B/B/B/B;
AE1=AE1/B/B/B/B;
AEE1=AEE1/B/B/B/B;
AE2=AE2/B/B/B/B;
AEE2=AEE2/B/B/B/B;
printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var0=%f\n",AE0,MAE0,MIE0,AEE0-AE0*AE0);
printf("(True Round) AE1=%f, MaxE=%f, MinE=%f, Var1=%f\n",AE1,MAE1,MIE1,AEE1-AE1*AE1);
printf("(Trunc.Schm) AE2=%f, MaxE=%f, MinE=%f, Var2=%f\n",AE2,MAE2,MIE2,AEE2-AE2*AE2);


/*****************************************************************
EXHAUSTIVE SIMULATION PROGRAM EXAMPLE C CODE

Proposal #3 Error Statistics
*****************************************************************/


#include <math.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <time.h>


int test();
int test1();

main ()
{

  float B,C,C1,C2,C3,C4,C5,C22,C33,C55;
  float E0,E1,E2,E3,AE0,AE1,AE2,AE3;
  float MAE0,MAE1,MAE2,MAE3;
  float MIE0,MIE1,MIE2,MIE3;
  float EE0,EE1,EE2,EE3,AEE0,AEE1,AEE2,AEE3;
  float FCI0,FCI1,FCI3,D1,D2,D3;
  int CI0,CI1,CI2,CI3,CII0,CII1;
  int n,i,j,k,ii,jj,kk;

/*
Case1:
      0 <= XH,XL,AH,AL <= 2^n-1
Case2:
      2^{n-1} <= XH,AH <= 2^n-1
            0 <= XL,AL <= 2^n-1
*/

for(n=8;n<9;n++)
{
printf("-------------n=%i---------------\n",n);
  B=1;
  for(i=0;i<n;i++)
    B=B*2;

AE0=0;AE1=0;AE2=0;
AEE0=0;AEE1=0;AEE2=0;
MAE0=0;MAE1=0;MAE2=0;
MIE0=0;MIE1=0;MIE2=0;
  for(XH=0;XH<B;XH++)
  for(AH=0;AH<B;AH++)
  for(XL=0;XL<B;XL++)
  for(AL=0;AL<B;AL++)
   {
C=AH*XH+(AH*XL+AL*XH)/B+AL*XL/B/B;
CI0=C;
CI1=AH*XH+(AH*XL+AL*XH)/B+AL*XL/B/B+0.5;
```

```
D1=C-CI0;
if(D1==0.5)
   {
    if(CI0/2*2==CI0) CI1=CI1-1;
   }
AL1=AL/(B/2);
XL1=XL/(B/2);

C2=AH*XH+(AH*XL+AL*XH)/B+AL1*XL1;


CI2=AH*XH+(AH*XL+AL*XH)/B+ AL1*XL1 + 0.5;

D2=C2-CII0;
if(D2==0.5)
   {
    if(CII0/2*2==CII0) CI2=CI2-1;
   }
E0=CI0-C;
if(E0>MAE0) MAE0=E0;
if(E0<MIE0) MIE0=E0;
EE0=E0*E0;
E1=CI1-C;
if(E1>MAE1) MAE1=E1;
if(E1<MIE1) MIE1=E1;
EE1=E1*E1;
E2=CI2-C;
if(E2>MAE2) MAE2=E2;
if(E2<MIE2) MIE2=E2;
EE2=E2*E2;
AE0=AE0+E0;
AEE0=AEE0+EE0;
AE1=AE1+E1;
AEE1=AEE1+EE1;
AE2=AE2+E2;
AEE2=AEE2+EE2;
   }
AE0=AE0/B/B/B/B;
AEE0=AEE0/B/B/B/B;
AE1=AE1/B/B/B/B;
AEE1=AEE1/B/B/B/B;
AE2=AE2/B/B/B/B;
AEE2=AEE2/B/B/B/B;
printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var0=%f\n",AE0,MAE0,MIE0,AEE0-AE0*AE0);
printf("(True Round) AE1=%f, MaxE=%f, MinE=%f, Var1=%f\n",AE1,MAE1,MIE1,AEE1-AE1*AE1);
printf("(Trunc.Schm) AE2=%f, MaxE=%f, MinE=%f, Var2=%f\n",AE2,MAE2,MIE2,AEE2-AE2*AE2);
}

}


/*****************************************************************
EXHAUSTIVE SIMULATION PROGRAM C CODE

Proposal #4 Error Statistics
*****************************************************************/


#include <math.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <time.h>


int test();
int test1();

main ()
{
```

```
  int XH,XL,AH,AL,AL1,XL1,AL2,XL2,AL3,XL3,AL4,XL4,AL5,XL5,AL6,XL6,AL7,XL7,AL8,XL8;

  float B,C,C1,C2,C3,C4,C5,C22,C33,C55;
  float E0,E1,E2,E3,AE0,AE1,AE2,AE3;
  float MAE0,MAE1,MAE2,MAE3;
  float MIE0,MIE1,MIE2,MIE3;
  float EE0,EE1,EE2,EE3,AEE0,AEE1,AEE2,AEE3;
  float FCI0,FCI1,FCI3,D1,D2,D3;
  int CI0,CI1,CI2,CI3,CII0,CII1;
  int n,i,j,k,ii,jj,kk;

/*
Case1:
      0 <= XH,XL,AH,AL <= 2^n-1
Case2:
      2^{n-1} <= XH,AH <= 2^n-1
            0 <= XL,AL <= 2^n-1
*/

for(n=8;n<9;n++)
{
printf("-------------n=%i---------------\n",n);
  B=1;
  for(i=0;i<n;i++)
    B=B*2;

AE0=0;AE1=0;AE2=0;
AEE0=0;AEE1=0;AEE2=0;
MAE0=0;MAE1=0;MAE2=0;
MIE0=0;MIE1=0;MIE2=0;
  for(XH=0;XH<B;XH++)
  for(AH=0;AH<B;AH++)
  for(XL=0;XL<B;XL++)
  for(AL=0;AL<B;AL++)
   {
C=AH*XH+(AH*XL+AL*XH)/B+AL*XL/B/B;
CI0=C;
CI1=AH*XH+(AH*XL+AL*XH)/B+AL*XL/B/B+0.5;
D1=C-CI0;
if(D1==0.5)
   {
    if(CI0/2*2==CI0) CI1=CI1-1;
   }
AL1=AL/(B/2);
XL1=XL/(B/2);


AL2=AL/(B/4);
XL2=XL/(B/4);


AL3=AL/(B/8);
XL3=XL/(B/8);


AL4=AL/(B/16);
XL4=XL/(B/16);

AL5=AL/(B/32);
XL5=XL/(B/32);

AL6=AL/(B/64);
XL6=XL/(B/64);

AL7=AL/(B/128);
XL7=XL/(B/128);

AL8=AL/(B/256);
XL8=XL/(B/256);
```

```
/*---------*/

AL2=AL2-AL1*2*2*2*2*2*2*2/(B/4);
XL2=XL2-XL1*2*2*2*2*2*2*2/(B/4);

AL3=AL3-AL1*2*2*2*2*2*2*2/(B/8)-AL2*2*2*2*2*2*2/(B/8);
XL3=XL3-XL1*2*2*2*2*2*2*2/(B/8)-XL2*2*2*2*2*2*2/(B/8);

AL4=AL4-AL1*2*2*2*2*2*2*2/(B/16)-AL2*2*2*2*2*2*2/(B/16)-AL3*2*2*2*2*2/(B/16);
XL4=XL4-XL1*2*2*2*2*2*2*2/(B/16)-XL2*2*2*2*2*2*2/(B/16)-XL3*2*2*2*2*2/(B/16);

AL5=AL5-AL1*2*2*2*2*2*2*2/(B/32)-AL2*2*2*2*2*2*2/(B/32)-AL3*2*2*2*2*2/(B/32)-
AL4*2*2*2*2/(B/32);
XL5=XL5-XL1*2*2*2*2*2*2*2/(B/32)-XL2*2*2*2*2*2*2/(B/32)-XL3*2*2*2*2*2/(B/32)-
XL4*2*2*2*2/(B/32);

AL6=AL6-AL1*2*2*2*2*2*2*2/(B/64)-AL2*2*2*2*2*2*2/(B/64)-AL3*2*2*2*2*2/(B/64)-
AL4*2*2*2*2/(B/64)-AL5*2*2*2/(B/64);
XL6=XL6-XL1*2*2*2*2*2*2*2/(B/64)-XL2*2*2*2*2*2*2/(B/64)-XL3*2*2*2*2*2/(B/64)-
XL4*2*2*2*2/(B/64)-XL5*2*2*2/(B/64);

AL7=AL7-AL1*2*2*2*2*2*2*2/(B/128)-AL2*2*2*2*2*2*2/(B/128)-AL3*2*2*2*2*2/(B/128)-
AL4*2*2*2*2/(B/128)-AL5*2*2*2/(B/128)-AL6*2*2/(B/128);
XL7=XL7-XL1*2*2*2*2*2*2*2/(B/128)-XL2*2*2*2*2*2*2/(B/128)-XL3*2*2*2*2*2/(B/128)-
XL4*2*2*2*2/(B/128)-XL5*2*2*2/(B/128)-XL6*2*2/(B/128);

AL8=AL8-AL1*2*2*2*2*2*2*2/(B/256)-AL2*2*2*2*2*2*2/(B/256)-AL3*2*2*2*2*2/(B/256)-
AL4*2*2*2*2/(B/256)-AL5*2*2*2/(B/256)-AL6*2*2/(B/256)-AL7*2/(B/256);
XL8=XL8-XL1*2*2*2*2*2*2*2/(B/256)-XL2*2*2*2*2*2*2/(B/256)-XL3*2*2*2*2*2/(B/256)-
XL4*2*2*2*2/(B/256)-XL5*2*2*2/(B/256)-XL6*2*2/(B/256)-XL7*2/(B/256);

/*---------*/

C2=AH*XH+(AH*XL+AL*XH)/B+(AH*XL+AL*XH)/2/B/B+

(AL1*XL1/2.0)+(AL2*XL2/2.0/2.0)+(AL3*XL3/2.0/2.0/2.0)+(AL4*XL4/2.0/2.0/2.0/2.0)+(AL5*XL5/
2.0/2.0/2.0/2.0/2.0)+(AL6*XL6/2.0/2.0/2.0/2.0/2.0/2.0)+(AL7*XL7/2.0/2.0/2.0/2.0/2.0/2.0/2
.0)+(AL8*XL8/2.0/2.0/2.0/2.0/2.0/2.0/2.0/2.0);
CII0=C2;

CI2=AH*XH+(AH*XL+AL*XH)/B+(AH*XL+AL*XH)/2/B/B+(AL1*XL1/2.0)+(AL2*XL2/2.0/2.0)+(AL3*XL3/2.
0/2.0/2.0)+(AL4*XL4/2.0/2.0/2.0/2.0)+(AL5*XL5/2.0/2.0/2.0/2.0/2.0)+(AL6*XL6/2.0/2.0/2.0/2
.0/2.0/2.0)+(AL7*XL7/2.0/2.0/2.0/2.0/2.0/2.0/2.0)+(AL8*XL8/2.0/2.0/2.0/2.0/2.0/2.0/2.0/2.
0)+0.5;
*/

D2=C2-CII0;
if(D2==0.5)
   {
    if(CII0/2*2==CII0) CI2=CI2-1;
   }
E0=CI0-C;
if(E0>MAE0) MAE0=E0;
if(E0<MIE0) MIE0=E0;
EE0=E0*E0;
E1=CI1-C;
if(E1>MAE1) MAE1=E1;
if(E1<MIE1) MIE1=E1;
EE1=E1*E1;
E2=CI2-C;
if(E2>MAE2) MAE2=E2;
if(E2<MIE2) MIE2=E2;
EE2=E2*E2;
AE0=AE0+E0;
AEE0=AEE0+EE0;
```

```
AE1=AE1+E1;
AEE1=AEE1+EE1;
AE2=AE2+E2;
AEE2=AEE2+EE2;
   }
AE0=AE0/B/B/B/B;
AEE0=AEE0/B/B/B/B;
AE1=AE1/B/B/B/B;
AEE1=AEE1/B/B/B/B;
AE2=AE2/B/B/B/B;
AEE2=AEE2/B/B/B/B;

printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var0=%f\n",AE0,MAE0,MIE0,AEE0-AE0*AE0);
printf("(True Round) AE1=%f, MaxE=%f, MinE=%f, Var1=%f\n",AE1,MAE1,MIE1,AEE1-AE1*AE1);
printf("(Trunc.Schm) AE2=%f, MaxE=%f, MinE=%f, Var2=%f\n",AE2,MAE2,MIE2,AEE2-AE2*AE2);
}

}



/*****************************************************************
EXHAUSTIVE SIMULATION PROGRAM EXAMPLE C CODE

Two-Level Fixed-Width Recursive Multiplier Error Statistics
*****************************************************************/

#include <math.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <time.h>


int test();
int test1();

main ()
{
  int XHH,XHL,XLH,XLL,AHH,AHL,ALH,ALL;
  int ALH1,XLH1,AHL1,XLL1,ALL1,XHL1;

  int AL1,XL1,AL2,XL2,AL3,XL3,AL4,XL4,AL5,XL5,AL6,XL6,AL7,XL7,AL8,XL8;

  float B,B2,C,C1,C2,C3,C4,C5,C22,C33,C55;
  float E0,E1,E2,E3,AE0,AE1,AE2,AE3;
  float MAE0,MAE1,MAE2,MAE3;
  float MIE0,MIE1,MIE2,MIE3;
  float EE0,EE1,EE2,EE3,AEE0,AEE1,AEE2,AEE3;
  float FCI0,FCI1,FCI3,D1,D2,D3;
  int CI0,CI1,CI2,CI3,CII0,CII1;
  int n,i,j,k,ii,jj,kk;

/*
Case1:
      0 <= XH,XL,AH,AL <= 2^n-1
Case2:
      2^{n-1} <= XH,AH <= 2^n-1
             0 <= XL,AL <= 2^n-1
*/

for(n=8;n<9;n++)
{
printf("-------------n=%i---------------\n",n);
  B2=1;
  B=1;

  for(i=0;i<(n/2);i++)
    B=B*2;

  for(i=0;i<n;i++)
```

```
    B2=B2*2;

AE0=0;AE1=0;AE2=0;
AEE0=0;AEE1=0;AEE2=0;
MAE0=0;MAE1=0;MAE2=0;
MIE0=0;MIE1=0;MIE2=0;


  for(XHH=0;XHH<B;XHH++)
  for(XHL=0;XHL<B;XHL++)
  for(XLH=0;XLH<B;XLH++)
  for(XLL=0;XLL<B;XLL++)

  for(AHH=0;AHH<B;AHH++)
  for(AHL=0;AHL<B;AHL++)
  for(ALH=0;ALH<B;ALH++)
  for(ALL=0;ALL<B;ALL++)
   {
C=(AHH*XHH*B*B+(AHH*XHL+AHL*XHH)*B+AHL*XHL)                              +
(AHH*XLH*B*B+(AHH*XLL+AHL*XLH)*B+AHL*XLL)/B2                             +
(ALH*XHH*B*B+(ALH*XHL+ALL*XHH)*B+ALL*XHL)/B2                             +
(ALH*XLH*B*B+(ALH*XLL+ALL*XLH)*B+ALL*XLL)/B2/B2;

CI0=C;

CI1=(AHH*XHH*B*B+(AHH*XHL+AHL*XHH)*B+AHL*XHL)                            +
(AHH*XLH*B*B+(AHH*XLL+AHL*XLH)*B+AHL*XLL)/B2                             +
(ALH*XHH*B*B+(ALH*XHL+ALL*XHH)*B+ALL*XHL)/B2                             +
(ALH*XLH*B*B+(ALH*XLL+ALL*XLH)*B+ALL*XLL)/B2/B2+0.5;
D1=C-CI0;
if(D1==0.5)
   {
    if(CI0/2*2==CI0) CI1=CI1-1;
   }

/*ALH1=ALH/(B/2);
XLH1=XLH/(B/2);

AHL1=AHL/(B/2);
XLL1=XLL/(B/2);

ALL1=ALL/(B/2);
XHL1=XHL/(B/2);*/


C2=(AHH*XHH*B*B+(AHH*XHL+AHL*XHH)*B+AHL*XHL)   +   (AHH*XLH*B*B+(AHH*XLL+AHL*XLH)*B)/B2   +
(ALH*XHH*B*B+(ALH*XHL+ALL*XHH)*B)/B2;

CII0=C2;

CI2=(AHH*XHH*B*B+(AHH*XHL+AHL*XHH)*B+AHL*XHL)   +   (AHH*XLH*B*B+(AHH*XLL+AHL*XLH)*B)/B2   +
(ALH*XHH*B*B+(ALH*XHL+ALL*XHH)*B)/B2+0.5;


D2=C2-CII0;
if(D2==0.5)
   {
    if(CII0/2*2==CII0) CI2=CI2-1;
   }
E0=CI0-C;
if(E0>MAE0) MAE0=E0;
if(E0<MIE0) MIE0=E0;
EE0=E0*E0;
E1=CI1-C;
if(E1>MAE1) MAE1=E1;
if(E1<MIE1) MIE1=E1;
EE1=E1*E1;
E2=CI2-C;
if(E2>MAE2) MAE2=E2;
if(E2<MIE2) MIE2=E2;
```

```
EE2=E2*E2;
AE0=AE0+E0;
AEE0=AEE0+EE0;
AE1=AE1+E1;
AEE1=AEE1+EE1;
AE2=AE2+E2;
AEE2=AEE2+EE2;
    }
AE0=AE0/B2/B2/B2/B2;
AEE0=AEE0/B2/B2/B2/B2;
AE1=AE1/B2/B2/B2/B2;
AEE1=AEE1/B2/B2/B2/B2;
AE2=AE2/B2/B2/B2/B2;
AEE2=AEE2/B2/B2/B2/B2;
printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var0=%f\n",AE0,MAE0,MIE0,AEE0-AE0*AE0);
printf("(True Round) AE1=%f, MaxE=%f, MinE=%f, Var1=%f\n",AE1,MAE1,MIE1,AEE1-AE1*AE1);
printf("(Trunc.Schm) AE2=%f, MaxE=%f, MinE=%f, Var2=%f\n",AE2,MAE2,MIE2,AEE2-AE2*AE2);
}

}
```

APPENDIX B

Verilog HDL Code

```
/****************************************
VERILOG HARDWARE DESCRIPTION LANGUAGE

Code for 32-bit Recursive Multiplier
using Truncation Scheme #4 (16 correction
bits)

****************************************/

module RECURSIVEMULTIPLIER (OUT,AH, AL,
XH, XL,CK);
        input CK;
        input [15:0] AH, AL, XH, XL;
        output [63:0] OUT;

        reg                          [0:0]
AL1,XL1,AL2,XL2,AL3,XL3,AL4,XL4,AL5,XL5,
AL6,XL6,AL7,XL7,AL8,XL8,AL9,XL9,AL10,
XL10,AL11,XL11,AL12,XL12,AL13,XL13,AL14,X
L14,AL15,XL15,AL16,XL16;



        wire [31:0] SUBMULT1;
        wire [31:0]    SUBMULT2,
SUBMULT3, SUBMULT4;
        reg [15:0] SHIFTSUBMULT1;
//      reg [31:0] SHIFTSUBMULT1;
        reg      [47:0]      SHIFTSUBMULT2,
SHIFTSUBMULT3;
        reg [63:0] SHIFTSUBMULT4;

        reg [63:0] OUT;//, OUT2;
//      reg [23:0] TEMP;

//      HIGHLOWBITS                    hlb0
(CK,A,B,AH,AL,XH,XL,AL1,XL1,AL2,XL2,AL3,X
L3,AL4,XL4,AL5,XL5,AL6,XL6,AL7,XL7,AL8,XL
8);



        //ARRAYMULTIPLIER16      basemult1
(SUBMULT1,AL,XL);  //remove ALXL
        //SUBMULT1[15:0] = 16'b0;

        ARRAYMULTIPLIER16        basemult2
(SUBMULT2,AH,XL);
        ARRAYMULTIPLIER16        basemult3
(SUBMULT3,AL,XH);
        ARRAYMULTIPLIER16        basemult4
(SUBMULT4,AH,XH);

        always @(posedge CK)
        begin

                AL1 = AL[15];
                AL2 = AL[14];
                AL3 = AL[13];
                AL4 = AL[12];
                AL5 = AL[11];
                AL6 = AL[10];
                AL7 = AL[9];
                AL8 = AL[8];
                AL9 = AL[7];
                AL10= AL[6];
                AL11= AL[5];
                AL12= AL[4];
                AL13= AL[3];
                AL14= AL[2];
                AL15= AL[1];
                AL16= AL[0];

                XL1 = XL[15];
                XL2 = XL[14];
                XL3 = XL[13];
                XL4 = XL[12];
                XL5 = XL[11];
                XL6 = XL[10];
                XL7 = XL[9];
                XL8 = XL[8];
                XL9 = XL[7];
                XL10= XL[6];
                XL11= XL[5];
                XL12= XL[4];
                XL13= XL[3];
                XL14= XL[2];
                XL15= XL[1];
                XL16= XL[0];


        SHIFTSUBMULT1[15] <= AL1&XL1;
        SHIFTSUBMULT1[14] <= AL2&XL2;
        SHIFTSUBMULT1[13] <= AL3&XL3;
        SHIFTSUBMULT1[12] <= AL4&XL4;
        SHIFTSUBMULT1[11] <= AL5&XL5;
        SHIFTSUBMULT1[10] <= AL6&XL6;
        SHIFTSUBMULT1[9] <= AL7&XL7;
        SHIFTSUBMULT1[8] <= AL8&XL8;
        SHIFTSUBMULT1[7] <= AL9&XL9;
        SHIFTSUBMULT1[6] <= AL10&XL10;
        SHIFTSUBMULT1[5] <= AL11&XL11;
        SHIFTSUBMULT1[4] <= AL12&XL12;

        SHIFTSUBMULT1[3] <= AL13&XL13;
        SHIFTSUBMULT1[2] <= AL14&XL14;
        SHIFTSUBMULT1[1] <= AL15&XL15;
        SHIFTSUBMULT1[0] <= AL16&XL16;



        SHIFTSUBMULT2 <= SUBMULT2 << 16;



        SHIFTSUBMULT3 <= SUBMULT3 <<16;
        SHIFTSUBMULT4 <= SUBMULT4 <<32;


        OUT     <=     (SHIFTSUBMULT1     +
SHIFTSUBMULT2)    +    (SHIFTSUBMULT3     +
SHIFTSUBMULT4);


                //OUT2 <= A*B;
        end
```

```
        //ANDGATE      andtemp      (temp,
SHIFTSUBMULT1[2], SHIFTSUBMULT3[4]);



endmodule //RECURSIVEMULTIPLIER


/*module                          HIGHLOWBITS
(CK,A,B,AH,AL,XH,XL,AL1,XL1,AL2,XL2,AL3,X
L3,AL4,XL4,AL5,XL5,AL6,XL6,AL7,XL7,AL8,XL
8);
        input CK;
        input [15:0] A,B;
        output [7:0] AH, AL, XH, XL;
        output                      [0:0]
AL1,XL1,AL2,XL2,AL3,XL3,AL4,XL4,AL5,XL5,A
L6,XL6,AL7,XL7,AL8,XL8;
        reg [7:0] AH, AL, XH, XL;
        reg                         [0:0]
AL1,XL1,AL2,XL2,AL3,XL3,AL4,XL4,AL5,XL5,A
L6,XL6,AL7,XL7,AL8,XL8;

        always @(posedge CK)
        begin
                AH <= { A[15:8] };
                AL <= { A[7:0] };

                AL1 <= { AL[7] };
                AL2 <= { AL[6] };
                AL3 <= { AL[5] };
                AL4 <= { AL[4] };
                AL5 <= { AL[3] };
                AL6 <= { AL[2] };
                AL7 <= { AL[1] };
                AL8 <= { AL[0] };



                XH <= { B[15:8] };
                XL <= { B[7:0] };

                XL1 <= { XL[7] };
                XL2 <= { XL[6] };
                XL3 <= { XL[5] };
                XL4 <= { XL[4] };
                XL5 <= { XL[3] };
                XL6 <= { XL[2] };
                XL7 <= { XL[1] };
                XL8 <= { XL[0] };


        end
endmodule //HIGHLOWBITS */


module ARRAYMULTIPLIER8 (OUT,A,B);

        input [7:0] A,B;
        output [15:0] OUT;
        //wire X;


        wire
        WCOUT00,WCOUT01,WCOUT02,WCOUT03,W
COUT04,WCOUT05,WCOUT06,WCOUT07,
        WCOUT10,WCOUT11,WCOUT12,WCOUT13,W
COUT14,WCOUT15,WCOUT16,WCOUT17,
        WCOUT20,WCOUT21,WCOUT22,WCOUT23,W
COUT24,WCOUT25,WCOUT26,WCOUT27,
        WCOUT30,WCOUT31,WCOUT32,WCOUT33,W
COUT34,WCOUT35,WCOUT36,WCOUT37,

        WCOUT40,WCOUT41,WCOUT42,WCOUT43,W
COUT44,WCOUT45,WCOUT46,WCOUT47,
        WCOUT50,WCOUT51,WCOUT52,WCOUT53,W
COUT54,WCOUT55,WCOUT56,WCOUT57,
        WCOUT60,WCOUT61,WCOUT62,WCOUT63,W
COUT64,WCOUT65,WCOUT66,WCOUT67,
        WCOUT70,WCOUT71,WCOUT72,WCOUT73,W
COUT74,WCOUT75,WCOUT76,WCOUT77;


        wire
        WSOUT00,WSOUT01,WSOUT02,WSOUT03,W
SOUT04,WSOUT05,WSOUT06,WSOUT07,
        WSOUT10,WSOUT11,WSOUT12,WSOUT13,W
SOUT14,WSOUT15,WSOUT16,WSOUT17,
        WSOUT20,WSOUT21,WSOUT22,WSOUT23,W
SOUT24,WSOUT25,WSOUT26,WSOUT27,
        WSOUT30,WSOUT31,WSOUT32,WSOUT33,W
SOUT34,WSOUT35,WSOUT36,WSOUT37,

        WSOUT40,WSOUT41,WSOUT42,WSOUT43,W
SOUT44,WSOUT45,WSOUT46,WSOUT47,
        WSOUT50,WSOUT51,WSOUT52,WSOUT53,W
SOUT54,WSOUT55,WSOUT56,WSOUT57,
        WSOUT60,WSOUT61,WSOUT62,WSOUT63,W
SOUT64,WSOUT65,WSOUT66,WSOUT67,
        WSOUT70,WSOUT71,WSOUT72,WSOUT73,W
SOUT74,WSOUT75,WSOUT76,WSOUT77;



        wire
        WCOUTFA1,    WCOUTFA2,    WCOUTFA3,
WCOUTFA4, WCOUTFA5, WCOUTFA6;

        //assign OUT2 = A*B;

        //ANDGATE andA0 (X, A[0], B[0]);

        //ARRAY MULTIPLIER ARCHITECTURE

        //ROW 0
        ARRAYCELL                 arraycell00
(A[0],B[0],1'b0,1'b0,WCOUT00,OUT[0]);
        ARRAYCELL                 arraycell01
(A[1],B[0],1'b0,1'b0,WCOUT01,WSOUT01);
        ARRAYCELL                 arraycell02
(A[2],B[0],1'b0,1'b0,WCOUT02,WSOUT02);
        ARRAYCELL                 arraycell03
(A[3],B[0],1'b0,1'b0,WCOUT03,WSOUT03);
        ARRAYCELL                 arraycell04
(A[4],B[0],1'b0,1'b0,WCOUT04,WSOUT04);
        ARRAYCELL                 arraycell05
(A[5],B[0],1'b0,1'b0,WCOUT05,WSOUT05);
        ARRAYCELL                 arraycell06
(A[6],B[0],1'b0,1'b0,WCOUT06,WSOUT06);
        ARRAYCELL                 arraycell07
(A[7],B[0],1'b0,1'b0,WCOUT07,WSOUT07);

        //ROW 1
        ARRAYCELL                 arraycell10
(A[0],B[1],WCOUT00,WSOUT01,WCOUT10,OUT[1]
);
        ARRAYCELL                 arraycell11
(A[1],B[1],WCOUT01,WSOUT02,WCOUT11,WSOUT1
1);
```

```
        ARRAYCELL                  arraycell12
(A[2],B[1],WCOUT02,WSOUT03,WCOUT12,WSOUT1
2);
        ARRAYCELL                  arraycell13
(A[3],B[1],WCOUT03,WSOUT04,WCOUT13,WSOUT1
3);
        ARRAYCELL                  arraycell14
(A[4],B[1],WCOUT04,WSOUT05,WCOUT14,WSOUT1
4);
        ARRAYCELL                  arraycell15
(A[5],B[1],WCOUT05,WSOUT06,WCOUT15,WSOUT1
5);
        ARRAYCELL                  arraycell16
(A[6],B[1],WCOUT06,WSOUT07,WCOUT16,WSOUT1
6);
        ARRAYCELL                  arraycell17
(A[7],B[1],WCOUT07,1'b0,WCOUT17,WSOUT17);

        //ROW 2
        ARRAYCELL                  arraycell20
(A[0],B[2],WCOUT10,WSOUT11,WCOUT20,OUT[2]
);
        ARRAYCELL                  arraycell21
(A[1],B[2],WCOUT11,WSOUT12,WCOUT21,WSOUT2
1);
        ARRAYCELL                  arraycell22
(A[2],B[2],WCOUT12,WSOUT13,WCOUT22,WSOUT2
2);
        ARRAYCELL                  arraycell23
(A[3],B[2],WCOUT13,WSOUT14,WCOUT23,WSOUT2
3);
        ARRAYCELL                  arraycell24
(A[4],B[2],WCOUT14,WSOUT15,WCOUT24,WSOUT2
4);
        ARRAYCELL                  arraycell25
(A[5],B[2],WCOUT15,WSOUT16,WCOUT25,WSOUT2
5);
        ARRAYCELL                  arraycell26
(A[6],B[2],WCOUT16,WSOUT17,WCOUT26,WSOUT2
6);
        ARRAYCELL                  arraycell27
(A[7],B[2],WCOUT17,1'b0,WCOUT27,WSOUT27);


        //ROW 3

        ARRAYCELL                  arraycell30
(A[0],B[3],WCOUT20,WSOUT21,WCOUT30,OUT[3]
);
        ARRAYCELL                  arraycell31
(A[1],B[3],WCOUT21,WSOUT22,WCOUT31,WSOUT3
1);
        ARRAYCELL                  arraycell32
(A[2],B[3],WCOUT22,WSOUT23,WCOUT32,WSOUT3
2);
        ARRAYCELL                  arraycell33
(A[3],B[3],WCOUT23,WSOUT24,WCOUT33,WSOUT3
3);
        ARRAYCELL                  arraycell34
(A[4],B[3],WCOUT24,WSOUT25,WCOUT34,WSOUT3
4);
        ARRAYCELL                  arraycell35
(A[5],B[3],WCOUT25,WSOUT26,WCOUT35,WSOUT3
5);
        ARRAYCELL                  arraycell36
(A[6],B[3],WCOUT26,WSOUT27,WCOUT36,WSOUT3
6);
        ARRAYCELL                  arraycell37
(A[7],B[3],WCOUT27,1'b0,WCOUT37,WSOUT37);

        //ROW 4

        ARRAYCELL                  arraycell40
(A[0],B[4],WCOUT30,WSOUT31,WCOUT40,OUT[4]
);
        ARRAYCELL                  arraycell41
(A[1],B[4],WCOUT31,WSOUT32,WCOUT41,WSOUT4
1);
        ARRAYCELL                  arraycell42
(A[2],B[4],WCOUT32,WSOUT33,WCOUT42,WSOUT4
2);
        ARRAYCELL                  arraycell43
(A[3],B[4],WCOUT33,WSOUT34,WCOUT43,WSOUT4
3);
        ARRAYCELL                  arraycell44
(A[4],B[4],WCOUT34,WSOUT35,WCOUT44,WSOUT4
4);
        ARRAYCELL                  arraycell45
(A[5],B[4],WCOUT35,WSOUT36,WCOUT45,WSOUT4
5);
        ARRAYCELL                  arraycell46
(A[6],B[4],WCOUT36,WSOUT37,WCOUT46,WSOUT4
6);
        ARRAYCELL                  arraycell47
(A[7],B[4],WCOUT37,1'b0,WCOUT47,WSOUT47);

        //ROW 5

        ARRAYCELL                  arraycell50
(A[0],B[5],WCOUT40,WSOUT41,WCOUT50,OUT[5]
);
        ARRAYCELL                  arraycell51
(A[1],B[5],WCOUT41,WSOUT42,WCOUT51,WSOUT5
1);
        ARRAYCELL                  arraycell52
(A[2],B[5],WCOUT42,WSOUT43,WCOUT52,WSOUT5
2);
        ARRAYCELL                  arraycell53
(A[3],B[5],WCOUT43,WSOUT44,WCOUT53,WSOUT5
3);
        ARRAYCELL                  arraycell54
(A[4],B[5],WCOUT44,WSOUT45,WCOUT54,WSOUT5
4);
        ARRAYCELL                  arraycell55
(A[5],B[5],WCOUT45,WSOUT46,WCOUT55,WSOUT5
5);
        ARRAYCELL                  arraycell56
(A[6],B[5],WCOUT46,WSOUT47,WCOUT56,WSOUT5
6);
        ARRAYCELL                  arraycell57
(A[7],B[5],WCOUT47,1'b0,WCOUT57,WSOUT57);



        //ROW 6

        ARRAYCELL                  arraycell60
(A[0],B[6],WCOUT50,WSOUT51,WCOUT60,OUT[6]
);
        ARRAYCELL                  arraycell61
(A[1],B[6],WCOUT51,WSOUT52,WCOUT61,WSOUT6
1);
        ARRAYCELL                  arraycell62
(A[2],B[6],WCOUT52,WSOUT53,WCOUT62,WSOUT6
2);
```

```
        ARRAYCELL                   arraycell63
(A[3],B[6],WCOUT53,WSOUT54,WCOUT63,WSOUT6
3);
        ARRAYCELL                   arraycell64
(A[4],B[6],WCOUT54,WSOUT55,WCOUT64,WSOUT6
4);
        ARRAYCELL                   arraycell65
(A[5],B[6],WCOUT55,WSOUT56,WCOUT65,WSOUT6
5);
        ARRAYCELL                   arraycell66
(A[6],B[6],WCOUT56,WSOUT57,WCOUT66,WSOUT6
6);
        ARRAYCELL                   arraycell67
(A[7],B[6],WCOUT57,1'b0,WCOUT67,WSOUT67);

        //ROW 7


        ARRAYCELL                   arraycell70
(A[0],B[7],WCOUT60,WSOUT61,WCOUT70,OUT[7]
);
        ARRAYCELL                   arraycell71
(A[1],B[7],WCOUT61,WSOUT62,WCOUT71,WSOUT7
1);
        ARRAYCELL                   arraycell72
(A[2],B[7],WCOUT62,WSOUT63,WCOUT72,WSOUT7
2);
        ARRAYCELL                   arraycell73
(A[3],B[7],WCOUT63,WSOUT64,WCOUT73,WSOUT7
3);
        ARRAYCELL                   arraycell74
(A[4],B[7],WCOUT64,WSOUT65,WCOUT74,WSOUT7
4);
        ARRAYCELL                   arraycell75
(A[5],B[7],WCOUT65,WSOUT66,WCOUT75,WSOUT7
5);
        ARRAYCELL                   arraycell76
(A[6],B[7],WCOUT66,WSOUT67,WCOUT76,WSOUT7
6);
        ARRAYCELL                   arraycell77
(A[7],B[7],WCOUT67,1'b0,WCOUT77,WSOUT77);



        //FULLADDER ROW

        FULLADDER                           fa1
(WSOUT71,WCOUT70,1'b0,WCOUTFA1,OUT[8]);
        FULLADDER                           fa2
(WSOUT72,WCOUT71,WCOUTFA1,WCOUTFA2,OUT[9]
);
        FULLADDER                           fa3
(WSOUT73,WCOUT72,WCOUTFA2,WCOUTFA3,OUT[10
]);
        FULLADDER                           fa4
(WSOUT74,WCOUT73,WCOUTFA3,WCOUTFA4,OUT[11
]);
        FULLADDER                           fa5
(WSOUT75,WCOUT74,WCOUTFA4,WCOUTFA5,OUT[12
]);
        FULLADDER                           fa6
(WSOUT76,WCOUT75,WCOUTFA5,WCOUTFA6,OUT[13
]);
        FULLADDER                           fa7
(WSOUT77,WCOUT76,WCOUTFA6,OUT[15],OUT[14]
);


endmodule//ARRAYMULTIPLIER8

module ARRAYMULTIPLIER16 (OUT,A,B);

        input [15:0] A,B;
        output [31:0] OUT;
        //wire X;


        wire
        WCOUT0000,
        WCOUT0001,
        WCOUT0002,
        WCOUT0003,
        WCOUT0004,
        WCOUT0005,
        WCOUT0006,
        WCOUT0007,
        WCOUT0008,
        WCOUT0009,
        WCOUT0010,
        WCOUT0011,
        WCOUT0012,
        WCOUT0013,
        WCOUT0014,
        WCOUT0015,

        WCOUT0100,
        WCOUT0101,
        WCOUT0102,
        WCOUT0103,
        WCOUT0104,
        WCOUT0105,
        WCOUT0106,
        WCOUT0107,
        WCOUT0108,
        WCOUT0109,
        WCOUT0110,
        WCOUT0111,
        WCOUT0112,
        WCOUT0113,
        WCOUT0114,
        WCOUT0115,

        WCOUT0200,
        WCOUT0201,
        WCOUT0202,
        WCOUT0203,
        WCOUT0204,
        WCOUT0205,
        WCOUT0206,
        WCOUT0207,
        WCOUT0208,
        WCOUT0209,
        WCOUT0210,
        WCOUT0211,
        WCOUT0212,
        WCOUT0213,
        WCOUT0214,
        WCOUT0215,

        WCOUT0300,
        WCOUT0301,
        WCOUT0302,
        WCOUT0303,
        WCOUT0304,
        WCOUT0305,
        WCOUT0306,
        WCOUT0307,
        WCOUT0308,
        WCOUT0309,
        WCOUT0310,
        WCOUT0311,
        WCOUT0312,
```

```
WCOUT0313,
WCOUT0314,
WCOUT0315,

WCOUT0400,
WCOUT0401,
WCOUT0402,
WCOUT0403,
WCOUT0404,
WCOUT0405,
WCOUT0406,
WCOUT0407,
WCOUT0408,
WCOUT0409,
WCOUT0410,
WCOUT0411,
WCOUT0412,
WCOUT0413,
WCOUT0414,
WCOUT0415,

WCOUT0500,
WCOUT0501,
WCOUT0502,
WCOUT0503,
WCOUT0504,
WCOUT0505,
WCOUT0506,
WCOUT0507,
WCOUT0508,
WCOUT0509,
WCOUT0510,
WCOUT0511,
WCOUT0512,
WCOUT0513,
WCOUT0514,
WCOUT0515,

WCOUT0600,
WCOUT0601,
WCOUT0602,
WCOUT0603,
WCOUT0604,
WCOUT0605,
WCOUT0606,
WCOUT0607,
WCOUT0608,
WCOUT0609,
WCOUT0610,
WCOUT0611,
WCOUT0612,
WCOUT0613,
WCOUT0614,
WCOUT0615,

WCOUT0700,
WCOUT0701,
WCOUT0702,
WCOUT0703,
WCOUT0704,
WCOUT0705,
WCOUT0706,
WCOUT0707,
WCOUT0708,
WCOUT0709,
WCOUT0710,
WCOUT0711,
WCOUT0712,
WCOUT0713,
WCOUT0714,
WCOUT0715,

WCOUT0800,
WCOUT0801,
WCOUT0802,
WCOUT0803,
WCOUT0804,
WCOUT0805,
WCOUT0806,
WCOUT0807,
WCOUT0808,
WCOUT0809,
WCOUT0810,
WCOUT0811,
WCOUT0812,
WCOUT0813,
WCOUT0814,
WCOUT0815,

WCOUT0900,
WCOUT0901,
WCOUT0902,
WCOUT0903,
WCOUT0904,
WCOUT0905,
WCOUT0906,
WCOUT0907,
WCOUT0908,
WCOUT0909,
WCOUT0910,
WCOUT0911,
WCOUT0912,
WCOUT0913,
WCOUT0914,
WCOUT0915,

WCOUT1000,
WCOUT1001,
WCOUT1002,
WCOUT1003,
WCOUT1004,
WCOUT1005,
WCOUT1006,
WCOUT1007,
WCOUT1008,
WCOUT1009,
WCOUT1010,
WCOUT1011,
WCOUT1012,
WCOUT1013,
WCOUT1014,
WCOUT1015,

WCOUT1100,
WCOUT1101,
WCOUT1102,
WCOUT1103,
WCOUT1104,
WCOUT1105,
WCOUT1106,
WCOUT1107,
WCOUT1108,
WCOUT1109,
WCOUT1110,
WCOUT1111,
WCOUT1112,
WCOUT1113,
WCOUT1114,
WCOUT1115,

WCOUT1200,
WCOUT1201,
```

```
WCOUT1202,
WCOUT1203,
WCOUT1204,
WCOUT1205,
WCOUT1206,
WCOUT1207,
WCOUT1208,
WCOUT1209,
WCOUT1210,
WCOUT1211,
WCOUT1212,
WCOUT1213,
WCOUT1214,
WCOUT1215,

WCOUT1300,
WCOUT1301,
WCOUT1302,
WCOUT1303,
WCOUT1304,
WCOUT1305,
WCOUT1306,
WCOUT1307,
WCOUT1308,
WCOUT1309,
WCOUT1310,
WCOUT1311,
WCOUT1312,
WCOUT1313,
WCOUT1314,
WCOUT1315,

WCOUT1400,
WCOUT1401,
WCOUT1402,
WCOUT1403,
WCOUT1404,
WCOUT1405,
WCOUT1406,
WCOUT1407,
WCOUT1408,
WCOUT1409,
WCOUT1410,
WCOUT1411,
WCOUT1412,
WCOUT1413,
WCOUT1414,
WCOUT1415,

WCOUT1500,
WCOUT1501,
WCOUT1502,
WCOUT1503,
WCOUT1504,
WCOUT1505,
WCOUT1506,
WCOUT1507,
WCOUT1508,
WCOUT1509,
WCOUT1510,
WCOUT1511,
WCOUT1512,
WCOUT1513,
WCOUT1514,
WCOUT1515;

wire
WSOUT0000,
WSOUT0001,
WSOUT0002,
WSOUT0003,
WSOUT0004,
WSOUT0005,
WSOUT0006,
WSOUT0007,
WSOUT0008,
WSOUT0009,
WSOUT0010,
WSOUT0011,
WSOUT0012,
WSOUT0013,
WSOUT0014,
WSOUT0015,

WSOUT0100,
WSOUT0101,
WSOUT0102,
WSOUT0103,
WSOUT0104,
WSOUT0105,
WSOUT0106,
WSOUT0107,

WSOUT0108,
WSOUT0109,
WSOUT0110,
WSOUT0111,
WSOUT0112,
WSOUT0113,
WSOUT0114,
WSOUT0115,

WSOUT0200,
WSOUT0201,
WSOUT0202,
WSOUT0203,
WSOUT0204,
WSOUT0205,
WSOUT0206,
WSOUT0207,
WSOUT0208,
WSOUT0209,
WSOUT0210,
WSOUT0211,
WSOUT0212,
WSOUT0213,
WSOUT0214,
WSOUT0215,

WSOUT0300,
WSOUT0301,
WSOUT0302,
WSOUT0303,
WSOUT0304,
WSOUT0305,
WSOUT0306,
WSOUT0307,
WSOUT0308,
WSOUT0309,
WSOUT0310,
WSOUT0311,
WSOUT0312,
WSOUT0313,
WSOUT0314,

WSOUT0315,

WSOUT0400,
WSOUT0401,
```

```
WSOUT0402,
WSOUT0403,
WSOUT0404,
WSOUT0405,
WSOUT0406,
WSOUT0407,
WSOUT0408,
WSOUT0409,
WSOUT0410,
WSOUT0411,
WSOUT0412,
WSOUT0413,
WSOUT0414,
WSOUT0415,

WSOUT0500,
WSOUT0501,
WSOUT0502,
WSOUT0503,
WSOUT0504,
WSOUT0505,
WSOUT0506,
WSOUT0507,
WSOUT0508,
WSOUT0509,
WSOUT0510,
WSOUT0511,
WSOUT0512,
WSOUT0513,
WSOUT0514,
WSOUT0515,

WSOUT0600,
WSOUT0601,
WSOUT0602,
WSOUT0603,
WSOUT0604,
WSOUT0605,
WSOUT0606,
WSOUT0607,
WSOUT0608,
WSOUT0609,
WSOUT0610,
WSOUT0611,
WSOUT0612,
WSOUT0613,
WSOUT0614,
WSOUT0615,

WSOUT0700,
WSOUT0701,
WSOUT0702,
WSOUT0703,
WSOUT0704,
WSOUT0705,
WSOUT0706,
WSOUT0707,
WSOUT0708,
WSOUT0709,
WSOUT0710,
WSOUT0711,
WSOUT0712,
WSOUT0713,
WSOUT0714,
WSOUT0715,

WSOUT0800,
WSOUT0801,
WSOUT0802,
WSOUT0803,
WSOUT0804,
WSOUT0805,
WSOUT0806,
WSOUT0807,
WSOUT0808,
WSOUT0809,
WSOUT0810,
WSOUT0811,
WSOUT0812,
WSOUT0813,
WSOUT0814,
WSOUT0815,

WSOUT0900,
WSOUT0901,
WSOUT0902,
WSOUT0903,
WSOUT0904,
WSOUT0905,
WSOUT0906,
WSOUT0907,
WSOUT0908,
WSOUT0909,
WSOUT0910,
WSOUT0911,
WSOUT0912,
WSOUT0913,
WSOUT0914,
WSOUT0915,

WSOUT1000,
WSOUT1001,
WSOUT1002,
WSOUT1003,
WSOUT1004,
WSOUT1005,
WSOUT1006,
WSOUT1007,
WSOUT1008,
WSOUT1009,
WSOUT1010,
WSOUT1011,
WSOUT1012,
WSOUT1013,
WSOUT1014,
WSOUT1015,

WSOUT1100,
WSOUT1101,
WSOUT1102,
WSOUT1103,
WSOUT1104,
WSOUT1105,
WSOUT1106,
WSOUT1107,
WSOUT1108,
WSOUT1109,
WSOUT1110,
WSOUT1111,
WSOUT1112,
WSOUT1113,
WSOUT1114,
WSOUT1115,

WSOUT1200,
WSOUT1201,
WSOUT1202,
WSOUT1203,
WSOUT1204,
WSOUT1205,
WSOUT1206,
WSOUT1207,
```

```
        WSOUT1208,
        WSOUT1209,
        WSOUT1210,
        WSOUT1211,
        WSOUT1212,
        WSOUT1213,
        WSOUT1214,
        WSOUT1215,

        WSOUT1300,
        WSOUT1301,
        WSOUT1302,
        WSOUT1303,
        WSOUT1304,
        WSOUT1305,
        WSOUT1306,
        WSOUT1307,
        WSOUT1308,
        WSOUT1309,
        WSOUT1310,
        WSOUT1311,
        WSOUT1312,
        WSOUT1313,
        WSOUT1314,
        WSOUT1315,

        WSOUT1400,
        WSOUT1401,
        WSOUT1402,
        WSOUT1403,
        WSOUT1404,
        WSOUT1405,
        WSOUT1406,
        WSOUT1407,
        WSOUT1408,
        WSOUT1409,
        WSOUT1410,
        WSOUT1411,
        WSOUT1412,
        WSOUT1413,
        WSOUT1414,
        WSOUT1415,

        WSOUT1500,
        WSOUT1501,
        WSOUT1502,
        WSOUT1503,
        WSOUT1504,
        WSOUT1505,
        WSOUT1506,
        WSOUT1507,
        WSOUT1508,
        WSOUT1509,
        WSOUT1510,
        WSOUT1511,
        WSOUT1512,
        WSOUT1513,
        WSOUT1514,
        WSOUT1515;


        wire
        WCOUTFA1,   WCOUTFA2,   WCOUTFA3,
WCOUTFA4, WCOUTFA5, WCOUTFA6,
        WCOUTFA7,   WCOUTFA8,   WCOUTFA9,
WCOUTFA10, WCOUTFA11, WCOUTFA12,
        WCOUTFA13, WCOUTFA14, WCOUTFA15;

        //ROW 0
        ARRAYCELL            arraycell0000
(A[0],B[0],1'b0,1'b0,WCOUT0000,OUT[0]);
        ARRAYCELL            arraycell0001
(A[1],B[0],1'b0,1'b0,WCOUT0001,WSOUT0001)
;
        ARRAYCELL            arraycell0002
(A[2],B[0],1'b0,1'b0,WCOUT0002,WSOUT0002)
;
        ARRAYCELL            arraycell0003
(A[3],B[0],1'b0,1'b0,WCOUT0003,WSOUT0003)
;
        ARRAYCELL            arraycell0004
(A[4],B[0],1'b0,1'b0,WCOUT0004,WSOUT0004)
;
        ARRAYCELL            arraycell0005
(A[5],B[0],1'b0,1'b0,WCOUT0005,WSOUT0005)
;
        ARRAYCELL            arraycell0006
(A[6],B[0],1'b0,1'b0,WCOUT0006,WSOUT0006)
;
        ARRAYCELL            arraycell0007
(A[7],B[0],1'b0,1'b0,WCOUT0007,WSOUT0007)
;
        ARRAYCELL            arraycell0008
(A[8],B[0],1'b0,1'b0,WCOUT0008,WSOUT0008)
;
        ARRAYCELL            arraycell0009
(A[9],B[0],1'b0,1'b0,WCOUT0009,WSOUT0009)
;
        ARRAYCELL            arraycell0010
(A[10],B[0],1'b0,1'b0,WCOUT0010,WSOUT0010
);
        ARRAYCELL            arraycell0011
(A[11],B[0],1'b0,1'b0,WCOUT0011,WSOUT0011
);
        ARRAYCELL            arraycell0012
(A[12],B[0],1'b0,1'b0,WCOUT0012,WSOUT0012
);
        ARRAYCELL            arraycell0013
(A[13],B[0],1'b0,1'b0,WCOUT0013,WSOUT0013
);
        ARRAYCELL            arraycell0014
(A[14],B[0],1'b0,1'b0,WCOUT0014,WSOUT0014
);
        ARRAYCELL            arraycell0015
(A[15],B[0],1'b0,1'b0,WCOUT0015,WSOUT0015
);


        //ROW 1
        ARRAYCELL            arraycell0100
(A[0],B[1],WCOUT0000,WSOUT0001,WCOUT0100,
OUT[1]);
        ARRAYCELL            arraycell0101
(A[1],B[1],WCOUT0001,WSOUT0002,WCOUT0101,
WSOUT0101);
        ARRAYCELL            arraycell0102
(A[2],B[1],WCOUT0002,WSOUT0003,WCOUT0102,
WSOUT0102);
        ARRAYCELL            arraycell0103
(A[3],B[1],WCOUT0003,WSOUT0004,WCOUT0103,
WSOUT0103);
        ARRAYCELL            arraycell0104
(A[4],B[1],WCOUT0004,WSOUT0005,WCOUT0104,
WSOUT0104);
        ARRAYCELL            arraycell0105
(A[5],B[1],WCOUT0005,WSOUT0006,WCOUT0105,
WSOUT0105);
        ARRAYCELL            arraycell0106
(A[6],B[1],WCOUT0006,WSOUT0007,WCOUT0106,
WSOUT0106);
```

```
        ARRAYCELL               arraycell0107
(A[7],B[1],WCOUT0007,WSOUT0008,WCOUT0107,
WSOUT0107);
        ARRAYCELL               arraycell0108
(A[8],B[1],WCOUT0008,WSOUT0009,WCOUT0108,
WSOUT0108);
        ARRAYCELL               arraycell0109
(A[9],B[1],WCOUT0009,WSOUT0010,WCOUT0109,
WSOUT0109);
        ARRAYCELL               arraycell0110
(A[10],B[1],WCOUT0010,WSOUT0011,WCOUT0110
,WSOUT0110);
        ARRAYCELL               arraycell0111
(A[11],B[1],WCOUT0011,WSOUT0012,WCOUT0111
,WSOUT0111);
        ARRAYCELL               arraycell0112
(A[12],B[1],WCOUT0012,WSOUT0013,WCOUT0112
,WSOUT0112);
        ARRAYCELL               arraycell0113
(A[13],B[1],WCOUT0013,WSOUT0014,WCOUT0113
,WSOUT0113);
        ARRAYCELL               arraycell0114
(A[14],B[1],WCOUT0014,WSOUT0015,WCOUT0114
,WSOUT0114);
        ARRAYCELL               arraycell0115
(A[15],B[1],WCOUT0015,1'b0,WCOUT0115,WSOU
T0115);

//ROW 2
        ARRAYCELL               arraycell0200
(A[0],B[2],WCOUT0100,WSOUT0101,WCOUT0200,
OUT[2]);
        ARRAYCELL               arraycell0201
(A[1],B[2],WCOUT0101,WSOUT0102,WCOUT0201,
WSOUT0201);
        ARRAYCELL               arraycell0202
(A[2],B[2],WCOUT0102,WSOUT0103,WCOUT0202,
WSOUT0202);
        ARRAYCELL               arraycell0203
(A[3],B[2],WCOUT0103,WSOUT0104,WCOUT0203,
WSOUT0203);
        ARRAYCELL               arraycell0204
(A[4],B[2],WCOUT0104,WSOUT0105,WCOUT0204,
WSOUT0204);
        ARRAYCELL               arraycell0205
(A[5],B[2],WCOUT0105,WSOUT0106,WCOUT0205,
WSOUT0205);
        ARRAYCELL               arraycell0206
(A[6],B[2],WCOUT0106,WSOUT0107,WCOUT0206,
WSOUT0206);
        ARRAYCELL               arraycell0207
(A[7],B[2],WCOUT0107,WSOUT0108,WCOUT0207,
WSOUT0207);
        ARRAYCELL               arraycell0208
(A[8],B[2],WCOUT0108,WSOUT0109,WCOUT0208,
WSOUT0208);
        ARRAYCELL               arraycell0209
(A[9],B[2],WCOUT0109,WSOUT0110,WCOUT0209,
WSOUT0209);
        ARRAYCELL               arraycell0210
(A[10],B[2],WCOUT0110,WSOUT0111,WCOUT0210
,WSOUT0210);
        ARRAYCELL               arraycell0211
(A[11],B[2],WCOUT0111,WSOUT0112,WCOUT0211
,WSOUT0211);
        ARRAYCELL               arraycell0212
(A[12],B[2],WCOUT0112,WSOUT0113,WCOUT0212
,WSOUT0212);
        ARRAYCELL               arraycell0213
(A[13],B[2],WCOUT0113,WSOUT0114,WCOUT0213
,WSOUT0213);
        ARRAYCELL               arraycell0214
(A[14],B[2],WCOUT0114,WSOUT0115,WCOUT0214
,WSOUT0214);
        ARRAYCELL               arraycell0215
(A[15],B[2],WCOUT0115,1'b0,WCOUT0215,WSOU
T0215);


        //ROW 3
        ARRAYCELL               arraycell0300
(A[0],B[3],WCOUT0200,WSOUT0201,WCOUT0300,
OUT[3]);
        ARRAYCELL               arraycell0301
(A[1],B[3],WCOUT0201,WSOUT0202,WCOUT0301,
WSOUT0301);
        ARRAYCELL               arraycell0302
(A[2],B[3],WCOUT0202,WSOUT0203,WCOUT0302,
WSOUT0302);
        ARRAYCELL               arraycell0303
(A[3],B[3],WCOUT0203,WSOUT0204,WCOUT0303,
WSOUT0303);
        ARRAYCELL               arraycell0304
(A[4],B[3],WCOUT0204,WSOUT0205,WCOUT0304,
WSOUT0304);
        ARRAYCELL               arraycell0305
(A[5],B[3],WCOUT0205,WSOUT0206,WCOUT0305,
WSOUT0305);
        ARRAYCELL               arraycell0306
(A[6],B[3],WCOUT0206,WSOUT0207,WCOUT0306,
WSOUT0306);
        ARRAYCELL               arraycell0307
(A[7],B[3],WCOUT0207,WSOUT0208,WCOUT0307,
WSOUT0307);
        ARRAYCELL               arraycell0308
(A[8],B[3],WCOUT0208,WSOUT0209,WCOUT0308,
WSOUT0308);
        ARRAYCELL               arraycell0309
(A[9],B[3],WCOUT0209,WSOUT0210,WCOUT0309,
WSOUT0309);
        ARRAYCELL               arraycell0310
(A[10],B[3],WCOUT0210,WSOUT0211,WCOUT0310
,WSOUT0310);
        ARRAYCELL               arraycell0311
(A[11],B[3],WCOUT0211,WSOUT0212,WCOUT0311
,WSOUT0311);
        ARRAYCELL               arraycell0312
(A[12],B[3],WCOUT0212,WSOUT0213,WCOUT0312
,WSOUT0312);
        ARRAYCELL               arraycell0313
(A[13],B[3],WCOUT0213,WSOUT0214,WCOUT0313
,WSOUT0313);
        ARRAYCELL               arraycell0314
(A[14],B[3],WCOUT0214,WSOUT0215,WCOUT0314
,WSOUT0314);
        ARRAYCELL               arraycell0315
(A[15],B[3],WCOUT0215,1'b0,WCOUT0315,WSOU
T0315);


//ROW 4
        ARRAYCELL               arraycell0400
(A[0],B[4],WCOUT0300,WSOUT0301,WCOUT0400,
OUT[4]);
        ARRAYCELL               arraycell0401
(A[1],B[4],WCOUT0301,WSOUT0302,WCOUT0401,
WSOUT0401);
        ARRAYCELL               arraycell0402
(A[2],B[4],WCOUT0302,WSOUT0303,WCOUT0402,
WSOUT0402);
```

```
        ARRAYCELL               arraycell0403
(A[3],B[4],WCOUT0303,WSOUT0304,WCOUT0403,
WSOUT0403);
        ARRAYCELL               arraycell0404
(A[4],B[4],WCOUT0304,WSOUT0305,WCOUT0404,
WSOUT0404);
        ARRAYCELL               arraycell0405
(A[5],B[4],WCOUT0305,WSOUT0306,WCOUT0405,
WSOUT0405);
        ARRAYCELL               arraycell0406
(A[6],B[4],WCOUT0306,WSOUT0307,WCOUT0406,
WSOUT0406);
        ARRAYCELL               arraycell0407
(A[7],B[4],WCOUT0307,WSOUT0308,WCOUT0407,
WSOUT0407);
        ARRAYCELL               arraycell0408
(A[8],B[4],WCOUT0308,WSOUT0309,WCOUT0408,
WSOUT0408);
        ARRAYCELL               arraycell0409
(A[9],B[4],WCOUT0309,WSOUT0310,WCOUT0409,
WSOUT0409);
        ARRAYCELL               arraycell0410
(A[10],B[4],WCOUT0310,WSOUT0311,WCOUT0410
,WSOUT0410);
        ARRAYCELL               arraycell0411
(A[11],B[4],WCOUT0311,WSOUT0312,WCOUT0411
,WSOUT0411);
        ARRAYCELL               arraycell0412
(A[12],B[4],WCOUT0312,WSOUT0313,WCOUT0412
,WSOUT0412);
        ARRAYCELL               arraycell0413
(A[13],B[4],WCOUT0313,WSOUT0314,WCOUT0413
,WSOUT0413);
        ARRAYCELL               arraycell0414
(A[14],B[4],WCOUT0314,WSOUT0315,WCOUT0414
,WSOUT0414);
        ARRAYCELL               arraycell0415
(A[15],B[4],WCOUT0315,1'b0,WCOUT0415,WSOU
T0415);

        //ROW 5
        ARRAYCELL               arraycell0500
(A[0],B[5],WCOUT0400,WSOUT0401,WCOUT0500,
OUT[5]);
        ARRAYCELL               arraycell0501
(A[1],B[5],WCOUT0401,WSOUT0402,WCOUT0501,
WSOUT0501);
        ARRAYCELL               arraycell0502
(A[2],B[5],WCOUT0402,WSOUT0403,WCOUT0502,
WSOUT0502);
        ARRAYCELL               arraycell0503
(A[3],B[5],WCOUT0403,WSOUT0404,WCOUT0503,
WSOUT0503);
        ARRAYCELL               arraycell0504
(A[4],B[5],WCOUT0404,WSOUT0405,WCOUT0504,
WSOUT0504);
        ARRAYCELL               arraycell0505
(A[5],B[5],WCOUT0405,WSOUT0406,WCOUT0505,
WSOUT0505);
        ARRAYCELL               arraycell0506
(A[6],B[5],WCOUT0406,WSOUT0407,WCOUT0506,
WSOUT0506);
        ARRAYCELL               arraycell0507
(A[7],B[5],WCOUT0407,WSOUT0408,WCOUT0507,
WSOUT0507);
        ARRAYCELL               arraycell0508
(A[8],B[5],WCOUT0408,WSOUT0409,WCOUT0508,
WSOUT0508);
        ARRAYCELL               arraycell0509
(A[9],B[5],WCOUT0409,WSOUT0410,WCOUT0509,
WSOUT0509);
        ARRAYCELL               arraycell0510
(A[10],B[5],WCOUT0410,WSOUT0411,WCOUT0510
,WSOUT0510);
        ARRAYCELL               arraycell0511
(A[11],B[5],WCOUT0411,WSOUT0412,WCOUT0511
,WSOUT0511);
        ARRAYCELL               arraycell0512
(A[12],B[5],WCOUT0412,WSOUT0413,WCOUT0512
,WSOUT0512);
        ARRAYCELL               arraycell0513
(A[13],B[5],WCOUT0413,WSOUT0414,WCOUT0513
,WSOUT0513);
        ARRAYCELL               arraycell0514
(A[14],B[5],WCOUT0414,WSOUT0415,WCOUT0514
,WSOUT0514);
        ARRAYCELL               arraycell0515
(A[15],B[5],WCOUT0415,1'b0,WCOUT0515,WSOU
T0515);

        //ROW 6
        ARRAYCELL               arraycell0600
(A[0],B[6],WCOUT0500,WSOUT0501,WCOUT0600,
OUT[6]);
        ARRAYCELL               arraycell0601
(A[1],B[6],WCOUT0501,WSOUT0502,WCOUT0601,
WSOUT0601);
        ARRAYCELL               arraycell0602
(A[2],B[6],WCOUT0502,WSOUT0503,WCOUT0602,
WSOUT0602);
        ARRAYCELL               arraycell0603
(A[3],B[6],WCOUT0503,WSOUT0504,WCOUT0603,
WSOUT0603);
        ARRAYCELL               arraycell0604
(A[4],B[6],WCOUT0504,WSOUT0505,WCOUT0604,
WSOUT0604);
        ARRAYCELL               arraycell0605
(A[5],B[6],WCOUT0505,WSOUT0506,WCOUT0605,
WSOUT0605);
        ARRAYCELL               arraycell0606
(A[6],B[6],WCOUT0506,WSOUT0507,WCOUT0606,
WSOUT0606);
        ARRAYCELL               arraycell0607
(A[7],B[6],WCOUT0507,WSOUT0508,WCOUT0607,
WSOUT0607);
        ARRAYCELL               arraycell0608
(A[8],B[6],WCOUT0508,WSOUT0509,WCOUT0608,
WSOUT0608);
        ARRAYCELL               arraycell0609
(A[9],B[6],WCOUT0509,WSOUT0510,WCOUT0609,
WSOUT0609);
        ARRAYCELL               arraycell0610
(A[10],B[6],WCOUT0510,WSOUT0511,WCOUT0610
,WSOUT0610);
        ARRAYCELL               arraycell0611
(A[11],B[6],WCOUT0511,WSOUT0512,WCOUT0611
,WSOUT0611);
        ARRAYCELL               arraycell0612
(A[12],B[6],WCOUT0512,WSOUT0513,WCOUT0612
,WSOUT0612);
        ARRAYCELL               arraycell0613
(A[13],B[6],WCOUT0513,WSOUT0514,WCOUT0613
,WSOUT0613);
        ARRAYCELL               arraycell0614
(A[14],B[6],WCOUT0514,WSOUT0515,WCOUT0614
,WSOUT0614);
        ARRAYCELL               arraycell0615
(A[15],B[6],WCOUT0515,1'b0,WCOUT0615,WSOU
T0615);

        //ROW 7
```

```
ARRAYCELL arraycell0700
(A[0],B[7],WCOUT0600,WSOUT0601,WCOUT0700,
OUT[7]);
ARRAYCELL arraycell0701
(A[1],B[7],WCOUT0601,WSOUT0602,WCOUT0701,
WSOUT0701);
ARRAYCELL arraycell0702
(A[2],B[7],WCOUT0602,WSOUT0603,WCOUT0702,
WSOUT0702);
ARRAYCELL arraycell0703
(A[3],B[7],WCOUT0603,WSOUT0604,WCOUT0703,
WSOUT0703);
ARRAYCELL arraycell0704
(A[4],B[7],WCOUT0604,WSOUT0605,WCOUT0704,
WSOUT0704);
ARRAYCELL arraycell0705
(A[5],B[7],WCOUT0605,WSOUT0606,WCOUT0705,
WSOUT0705);
ARRAYCELL arraycell0706
(A[6],B[7],WCOUT0606,WSOUT0607,WCOUT0706,
WSOUT0706);
ARRAYCELL arraycell0707
(A[7],B[7],WCOUT0607,WSOUT0608,WCOUT0707,
WSOUT0707);
ARRAYCELL arraycell0708
(A[8],B[7],WCOUT0608,WSOUT0609,WCOUT0708,
WSOUT0708);
ARRAYCELL arraycell0709
(A[9],B[7],WCOUT0609,WSOUT0610,WCOUT0709,
WSOUT0709);
ARRAYCELL arraycell0710
(A[10],B[7],WCOUT0610,WSOUT0611,WCOUT0710
,WSOUT0710);
ARRAYCELL arraycell0711
(A[11],B[7],WCOUT0611,WSOUT0612,WCOUT0711
,WSOUT0711);
ARRAYCELL arraycell0712
(A[12],B[7],WCOUT0612,WSOUT0613,WCOUT0712
,WSOUT0712);
ARRAYCELL arraycell0713
(A[13],B[7],WCOUT0613,WSOUT0614,WCOUT0713
,WSOUT0713);
ARRAYCELL arraycell0714
(A[14],B[7],WCOUT0614,WSOUT0615,WCOUT0714
,WSOUT0714);
ARRAYCELL arraycell0715
(A[15],B[7],WCOUT0615,1'b0,WCOUT0715,WSOU
T0715);

//ROW 8
ARRAYCELL arraycell0800
(A[0],B[8],WCOUT0700,WSOUT0701,WCOUT0800,
OUT[8]);
ARRAYCELL arraycell0801
(A[1],B[8],WCOUT0701,WSOUT0702,WCOUT0801,
WSOUT0801);
ARRAYCELL arraycell0802
(A[2],B[8],WCOUT0702,WSOUT0703,WCOUT0802,
WSOUT0802);
ARRAYCELL arraycell0803
(A[3],B[8],WCOUT0703,WSOUT0704,WCOUT0803,
WSOUT0803);
ARRAYCELL arraycell0804
(A[4],B[8],WCOUT0704,WSOUT0705,WCOUT0804,
WSOUT0804);
ARRAYCELL arraycell0805
(A[5],B[8],WCOUT0705,WSOUT0706,WCOUT0805,
WSOUT0805);
ARRAYCELL arraycell0806
(A[6],B[8],WCOUT0706,WSOUT0707,WCOUT0806,
WSOUT0806);
ARRAYCELL arraycell0807
(A[7],B[8],WCOUT0707,WSOUT0708,WCOUT0807,
WSOUT0807);
ARRAYCELL arraycell0808
(A[8],B[8],WCOUT0708,WSOUT0709,WCOUT0808,
WSOUT0808);
ARRAYCELL arraycell0809
(A[9],B[8],WCOUT0709,WSOUT0710,WCOUT0809,
WSOUT0809);
ARRAYCELL arraycell0810
(A[10],B[8],WCOUT0710,WSOUT0711,WCOUT0810
,WSOUT0810);
ARRAYCELL arraycell0811
(A[11],B[8],WCOUT0711,WSOUT0712,WCOUT0811
,WSOUT0811);
ARRAYCELL arraycell0812
(A[12],B[8],WCOUT0712,WSOUT0713,WCOUT0812
,WSOUT0812);
ARRAYCELL arraycell0813
(A[13],B[8],WCOUT0713,WSOUT0714,WCOUT0813
,WSOUT0813);
ARRAYCELL arraycell0814
(A[14],B[8],WCOUT0714,WSOUT0715,WCOUT0814
,WSOUT0814);
ARRAYCELL arraycell0815
(A[15],B[8],WCOUT0715,1'b0,WCOUT0815,WSOU
T0815);

//ROW 9
ARRAYCELL arraycell0900
(A[0],B[9],WCOUT0800,WSOUT0801,WCOUT0900,
OUT[9]);
ARRAYCELL arraycell0901
(A[1],B[9],WCOUT0801,WSOUT0802,WCOUT0901,
WSOUT0901);
ARRAYCELL arraycell0902
(A[2],B[9],WCOUT0802,WSOUT0803,WCOUT0902,
WSOUT0902);
ARRAYCELL arraycell0903
(A[3],B[9],WCOUT0803,WSOUT0804,WCOUT0903,
WSOUT0903);
ARRAYCELL arraycell0904
(A[4],B[9],WCOUT0804,WSOUT0805,WCOUT0904,
WSOUT0904);
ARRAYCELL arraycell0905
(A[5],B[9],WCOUT0805,WSOUT0806,WCOUT0905,
WSOUT0905);
ARRAYCELL arraycell0906
(A[6],B[9],WCOUT0806,WSOUT0807,WCOUT0906,
WSOUT0906);
ARRAYCELL arraycell0907
(A[7],B[9],WCOUT0807,WSOUT0808,WCOUT0907,
WSOUT0907);
ARRAYCELL arraycell0908
(A[8],B[9],WCOUT0808,WSOUT0809,WCOUT0908,
WSOUT0908);
ARRAYCELL arraycell0909
(A[9],B[9],WCOUT0809,WSOUT0810,WCOUT0909,
WSOUT0909);
ARRAYCELL arraycell0910
(A[10],B[9],WCOUT0810,WSOUT0811,WCOUT0910
,WSOUT0910);
ARRAYCELL arraycell0911
(A[11],B[9],WCOUT0811,WSOUT0812,WCOUT0911
,WSOUT0911);
ARRAYCELL arraycell0912
(A[12],B[9],WCOUT0812,WSOUT0813,WCOUT0912
,WSOUT0912);
ARRAYCELL arraycell0913
(A[13],B[9],WCOUT0813,WSOUT0814,WCOUT0913
,WSOUT0913);
```

```
ARRAYCELL arraycell0914
(A[14],B[9],WCOUT0814,WSOUT0815,WCOUT0914
,WSOUT0914);
ARRAYCELL arraycell0915
(A[15],B[9],WCOUT0815,1'b0,WCOUT0915,WSOU
T0915);

//ROW 10
ARRAYCELL arraycell1000
(A[0],B[10],WCOUT0900,WSOUT0901,WCOUT1000
,OUT[10]);
ARRAYCELL arraycell1001
(A[1],B[10],WCOUT0901,WSOUT0902,WCOUT1001
,WSOUT1001);
ARRAYCELL arraycell1002
(A[2],B[10],WCOUT0902,WSOUT0903,WCOUT1002
,WSOUT1002);
ARRAYCELL arraycell1003
(A[3],B[10],WCOUT0903,WSOUT0904,WCOUT1003
,WSOUT1003);
ARRAYCELL arraycell1004
(A[4],B[10],WCOUT0904,WSOUT0905,WCOUT1004
,WSOUT1004);
ARRAYCELL arraycell1005
(A[5],B[10],WCOUT0905,WSOUT0906,WCOUT1005
,WSOUT1005);
ARRAYCELL arraycell1006
(A[6],B[10],WCOUT0906,WSOUT0907,WCOUT1006
,WSOUT1006);
ARRAYCELL arraycell1007
(A[7],B[10],WCOUT0907,WSOUT0908,WCOUT1007
,WSOUT1007);
ARRAYCELL arraycell1008
(A[8],B[10],WCOUT0908,WSOUT0909,WCOUT1008
,WSOUT1008);
ARRAYCELL arraycell1009
(A[9],B[10],WCOUT0909,WSOUT0910,WCOUT1009
,WSOUT1009);
ARRAYCELL arraycell1010
(A[10],B[10],WCOUT0910,WSOUT0911,WCOUT101
0,WSOUT1010);
ARRAYCELL arraycell1011
(A[11],B[10],WCOUT0911,WSOUT0912,WCOUT101
1,WSOUT1011);
ARRAYCELL arraycell1012
(A[12],B[10],WCOUT0912,WSOUT0913,WCOUT101
2,WSOUT1012);
ARRAYCELL arraycell1013
(A[13],B[10],WCOUT0913,WSOUT0914,WCOUT101
3,WSOUT1013);
ARRAYCELL arraycell1014
(A[14],B[10],WCOUT0914,WSOUT0915,WCOUT101
4,WSOUT1014);
ARRAYCELL arraycell1015
(A[15],B[10],WCOUT0915,1'b0,WCOUT1015,WSO
UT1015);

//ROW 11

ARRAYCELL arraycell1100
(A[0],B[11],WCOUT1000,WSOUT1001,WCOUT1100
,OUT[11]);
ARRAYCELL arraycell1101
(A[1],B[11],WCOUT1001,WSOUT1002,WCOUT1101
,WSOUT1101);
ARRAYCELL arraycell1102
(A[2],B[11],WCOUT1002,WSOUT1003,WCOUT1102
,WSOUT1102);
ARRAYCELL arraycell1103
(A[3],B[11],WCOUT1003,WSOUT1004,WCOUT1103
,WSOUT1103);
ARRAYCELL arraycell1104
(A[4],B[11],WCOUT1004,WSOUT1005,WCOUT1104
,WSOUT1104);
ARRAYCELL arraycell1105
(A[5],B[11],WCOUT1005,WSOUT1006,WCOUT1105
,WSOUT1105);
ARRAYCELL arraycell1106
(A[6],B[11],WCOUT1006,WSOUT1007,WCOUT1106
,WSOUT1106);
ARRAYCELL arraycell1107
(A[7],B[11],WCOUT1007,WSOUT1008,WCOUT1107
,WSOUT1107);
ARRAYCELL arraycell1108
(A[8],B[11],WCOUT1008,WSOUT1009,WCOUT1108
,WSOUT1108);
ARRAYCELL arraycell1109
(A[9],B[11],WCOUT1009,WSOUT1010,WCOUT1109
,WSOUT1109);
ARRAYCELL arraycell1110
(A[10],B[11],WCOUT1010,WSOUT1011,WCOUT111
0,WSOUT1110);
ARRAYCELL arraycell1111
(A[11],B[11],WCOUT1011,WSOUT1012,WCOUT111
1,WSOUT1111);
ARRAYCELL arraycell1112
(A[12],B[11],WCOUT1012,WSOUT1013,WCOUT111
2,WSOUT1112);
ARRAYCELL arraycell1113
(A[13],B[11],WCOUT1013,WSOUT1014,WCOUT111
3,WSOUT1113);
ARRAYCELL arraycell1114
(A[14],B[11],WCOUT1014,WSOUT1015,WCOUT111
4,WSOUT1114);
ARRAYCELL arraycell1115
(A[15],B[11],WCOUT1015,1'b0,WCOUT1115,WSO
UT1115);

//ROW 12
ARRAYCELL arraycell1200
(A[0],B[12],WCOUT1100,WSOUT1101,WCOUT1200
,OUT[12]);
ARRAYCELL arraycell1201
(A[1],B[12],WCOUT1101,WSOUT1102,WCOUT1201
,WSOUT1201);
ARRAYCELL arraycell1202
(A[2],B[12],WCOUT1102,WSOUT1103,WCOUT1202
,WSOUT1202);
ARRAYCELL arraycell1203
(A[3],B[12],WCOUT1103,WSOUT1104,WCOUT1203
,WSOUT1203);
ARRAYCELL arraycell1204
(A[4],B[12],WCOUT1104,WSOUT1105,WCOUT1204
,WSOUT1204);
ARRAYCELL arraycell1205
(A[5],B[12],WCOUT1105,WSOUT1106,WCOUT1205
,WSOUT1205);
ARRAYCELL arraycell1206
(A[6],B[12],WCOUT1106,WSOUT1107,WCOUT1206
,WSOUT1206);
ARRAYCELL arraycell1207
(A[7],B[12],WCOUT1107,WSOUT1108,WCOUT1207
,WSOUT1207);
ARRAYCELL arraycell1208
(A[8],B[12],WCOUT1108,WSOUT1109,WCOUT1208
,WSOUT1208);
ARRAYCELL arraycell1209
(A[9],B[12],WCOUT1109,WSOUT1110,WCOUT1209
,WSOUT1209);
ARRAYCELL arraycell1210
(A[10],B[12],WCOUT1110,WSOUT1111,WCOUT121
0,WSOUT1210);
```

```
        ARRAYCELL              arraycell1211
(A[11],B[12],WCOUT1111,WSOUT1112,WCOUT121
1,WSOUT1211);
        ARRAYCELL              arraycell1212
(A[12],B[12],WCOUT1112,WSOUT1113,WCOUT121
2,WSOUT1212);
        ARRAYCELL              arraycell1213
(A[13],B[12],WCOUT1113,WSOUT1114,WCOUT121
3,WSOUT1213);
        ARRAYCELL              arraycell1214
(A[14],B[12],WCOUT1114,WSOUT1115,WCOUT121
4,WSOUT1214);
        ARRAYCELL              arraycell1215
(A[15],B[12],WCOUT1115,1'b0,WCOUT1215,WSO
UT1215);

        //ROW 13
        ARRAYCELL              arraycell1300
(A[0],B[13],WCOUT1200,WSOUT1201,WCOUT1300
,OUT[13]);
        ARRAYCELL              arraycell1301
(A[1],B[13],WCOUT1201,WSOUT1202,WCOUT1301
,WSOUT1301);
        ARRAYCELL              arraycell1302
(A[2],B[13],WCOUT1202,WSOUT1203,WCOUT1302
,WSOUT1302);
        ARRAYCELL              arraycell1303
(A[3],B[13],WCOUT1203,WSOUT1204,WCOUT1303
,WSOUT1303);
        ARRAYCELL              arraycell1304
(A[4],B[13],WCOUT1204,WSOUT1205,WCOUT1304
,WSOUT1304);
        ARRAYCELL              arraycell1305
(A[5],B[13],WCOUT1205,WSOUT1206,WCOUT1305
,WSOUT1305);
        ARRAYCELL              arraycell1306
(A[6],B[13],WCOUT1206,WSOUT1207,WCOUT1306
,WSOUT1306);
        ARRAYCELL              arraycell1307
(A[7],B[13],WCOUT1207,WSOUT1208,WCOUT1307
,WSOUT1307);
        ARRAYCELL              arraycell1308
(A[8],B[13],WCOUT1208,WSOUT1209,WCOUT1308
,WSOUT1308);
        ARRAYCELL              arraycell1309
(A[9],B[13],WCOUT1209,WSOUT1210,WCOUT1309
,WSOUT1309);
        ARRAYCELL              arraycell1310
(A[10],B[13],WCOUT1210,WSOUT1211,WCOUT131
0,WSOUT1310);
        ARRAYCELL              arraycell1311
(A[11],B[13],WCOUT1211,WSOUT1212,WCOUT131
1,WSOUT1311);
        ARRAYCELL              arraycell1312
(A[12],B[13],WCOUT1212,WSOUT1213,WCOUT131
2,WSOUT1312);
        ARRAYCELL              arraycell1313
(A[13],B[13],WCOUT1213,WSOUT1214,WCOUT131
3,WSOUT1313);
        ARRAYCELL              arraycell1314
(A[14],B[13],WCOUT1214,WSOUT1215,WCOUT131
4,WSOUT1314);
        ARRAYCELL              arraycell1315
(A[15],B[13],WCOUT1215,1'b0,WCOUT1315,WSO
UT1315);

        //ROW 14
        ARRAYCELL              arraycell1400
(A[0],B[14],WCOUT1300,WSOUT1301,WCOUT1400
,OUT[14]);
        ARRAYCELL              arraycell1401
(A[1],B[14],WCOUT1301,WSOUT1302,WCOUT1401
,WSOUT1401);
        ARRAYCELL              arraycell1402
(A[2],B[14],WCOUT1302,WSOUT1303,WCOUT1402
,WSOUT1402);
        ARRAYCELL              arraycell1403
(A[3],B[14],WCOUT1303,WSOUT1304,WCOUT1403
,WSOUT1403);
        ARRAYCELL              arraycell1404
(A[4],B[14],WCOUT1304,WSOUT1305,WCOUT1404
,WSOUT1404);
        ARRAYCELL              arraycell1405
(A[5],B[14],WCOUT1305,WSOUT1306,WCOUT1405
,WSOUT1405);
        ARRAYCELL              arraycell1406
(A[6],B[14],WCOUT1306,WSOUT1307,WCOUT1406
,WSOUT1406);
        ARRAYCELL              arraycell1407
(A[7],B[14],WCOUT1307,WSOUT1308,WCOUT1407
,WSOUT1407);
        ARRAYCELL              arraycell1408
(A[8],B[14],WCOUT1308,WSOUT1309,WCOUT1408
,WSOUT1408);
        ARRAYCELL              arraycell1409
(A[9],B[14],WCOUT1309,WSOUT1310,WCOUT1409
,WSOUT1409);
        ARRAYCELL              arraycell1410
(A[10],B[14],WCOUT1310,WSOUT1311,WCOUT141
0,WSOUT1410);
        ARRAYCELL              arraycell1411
(A[11],B[14],WCOUT1311,WSOUT1312,WCOUT141
1,WSOUT1411);
        ARRAYCELL              arraycell1412
(A[12],B[14],WCOUT1312,WSOUT1313,WCOUT141
2,WSOUT1412);
        ARRAYCELL              arraycell1413
(A[13],B[14],WCOUT1313,WSOUT1314,WCOUT141
3,WSOUT1413);
        ARRAYCELL              arraycell1414
(A[14],B[14],WCOUT1314,WSOUT1315,WCOUT141
4,WSOUT1414);
        ARRAYCELL              arraycell1415
(A[15],B[14],WCOUT1315,1'b0,WCOUT1415,WSO
UT1415);

        //ROW 15
        ARRAYCELL              arraycell1500
(A[0],B[15],WCOUT1400,WSOUT1401,WCOUT1500
,OUT[15]);
        ARRAYCELL              arraycell1501
(A[1],B[15],WCOUT1401,WSOUT1402,WCOUT1501
,WSOUT1501);
        ARRAYCELL              arraycell1502
(A[2],B[15],WCOUT1402,WSOUT1403,WCOUT1502
,WSOUT1502);
        ARRAYCELL              arraycell1503
(A[3],B[15],WCOUT1403,WSOUT1404,WCOUT1503
,WSOUT1503);
        ARRAYCELL              arraycell1504
(A[4],B[15],WCOUT1404,WSOUT1405,WCOUT1504
,WSOUT1504);
        ARRAYCELL              arraycell1505
(A[5],B[15],WCOUT1405,WSOUT1406,WCOUT1505
,WSOUT1505);
        ARRAYCELL              arraycell1506
(A[6],B[15],WCOUT1406,WSOUT1407,WCOUT1506
,WSOUT1506);
        ARRAYCELL              arraycell1507
(A[7],B[15],WCOUT1407,WSOUT1408,WCOUT1507
,WSOUT1507);
```

```
        ARRAYCELL                   arraycell1508
(A[8],B[15],WCOUT1408,WSOUT1409,WCOUT1508
,WSOUT1508);
        ARRAYCELL                   arraycell1509
(A[9],B[15],WCOUT1409,WSOUT1410,WCOUT1509
,WSOUT1509);
        ARRAYCELL                   arraycell1510
(A[10],B[15],WCOUT1410,WSOUT1411,WCOUT151
0,WSOUT1510);
        ARRAYCELL                   arraycell1511
(A[11],B[15],WCOUT1411,WSOUT1412,WCOUT151
1,WSOUT1511);
        ARRAYCELL                   arraycell1512
(A[12],B[15],WCOUT1412,WSOUT1413,WCOUT151
2,WSOUT1512);
        ARRAYCELL                   arraycell1513
(A[13],B[15],WCOUT1413,WSOUT1414,WCOUT151
3,WSOUT1513);
        ARRAYCELL                   arraycell1514
(A[14],B[15],WCOUT1414,WSOUT1415,WCOUT151
4,WSOUT1514);
        ARRAYCELL                   arraycell1515
(A[15],B[15],WCOUT1415,1'b0,WCOUT1515,WSO
UT1515);


        //FULLADDER ROW

        FULLADDER                             fa1
(WSOUT1501,WCOUT1500,1'b0,WCOUTFA1,OUT[16
]);
        FULLADDER                             fa2
(WSOUT1502,WCOUT1501,WCOUTFA1,WCOUTFA2,OU
T[17]);
        FULLADDER                             fa3
(WSOUT1503,WCOUT1502,WCOUTFA2,WCOUTFA3,OU
T[18]);
        FULLADDER                             fa4
(WSOUT1504,WCOUT1503,WCOUTFA3,WCOUTFA4,OU
T[19]);
        FULLADDER                             fa5
(WSOUT1505,WCOUT1504,WCOUTFA4,WCOUTFA5,OU
T[20]);
        FULLADDER                             fa6
(WSOUT1506,WCOUT1505,WCOUTFA5,WCOUTFA6,OU
T[21]);
        FULLADDER                             fa7
(WSOUT1507,WCOUT1506,WCOUTFA6,WCOUTFA7,OU
T[22]);
        FULLADDER                             fa8
(WSOUT1508,WCOUT1507,WCOUTFA7,WCOUTFA8,OU
T[23]);
        FULLADDER                             fa9
(WSOUT1509,WCOUT1508,WCOUTFA8,WCOUTFA9,OU
T[24]);
        FULLADDER                            fa10
(WSOUT1510,WCOUT1509,WCOUTFA9,WCOUTFA10,O
UT[25]);
        FULLADDER                            fa11
(WSOUT1511,WCOUT1510,WCOUTFA10,WCOUTFA11,
OUT[26]);
        FULLADDER                            fa12
(WSOUT1512,WCOUT1511,WCOUTFA11,WCOUTFA12,
OUT[27]);
        FULLADDER                            fa13
(WSOUT1513,WCOUT1512,WCOUTFA12,WCOUTFA13,
OUT[28]);
        FULLADDER                            fa14
(WSOUT1514,WCOUT1513,WCOUTFA13,WCOUTFA14,
OUT[29]);
        FULLADDER                            fa15
(WSOUT1515,WCOUT1514,WCOUTFA14,OUT[31],OU
T[30]);


endmodule//ARRAYMULTIPLIER 16


module ARRAYCELL (A,B,CIN,SIN,COUT,SOUT);

        input A,B,SIN,CIN;
        output COUT,SOUT;
        wire PP;

        ANDGATE and0 (PP,A,B);
        FULLADDER                      fulladder0
(PP,SIN,CIN,COUT,SOUT);


endmodule //ARRAYCELL


module ANDGATE (OUT,A,B);

        input A,B;
        output OUT;
        assign OUT = A&B;

endmodule //ANDGATE

module HALFADDER (A,B,COUT,SUM);

        input A,B;
        output COUT,SUM;

        assign SUM = A^B;
        assign COUT = A&B;

endmodule //HALFADDER

module FULLADDER (A,B,CIN,COUT,SUM);

        input A,B,CIN;
        output COUT,SUM;

        assign SUM = A^B^CIN;
        assign COUT = A&B|A&CIN|B&CIN;

endmodule //FULLADDER
```

APPENDIX C

Simulation Reports and Logs from Altera Quartus II

"32 BIT RECURSIVE MULTIPLIER WITH PROPOSAL #4 TRUNCATION SCHEME (16 CORR. BITS)"

RECURSIVEMULTIPLIER Analysis & Synthesis Source Files Read

File Name with User-Entered Path   Used in Netlist   File Type   File Name with Absolute Path
RECURSIVEMULTIPLIER.v yes   User   Verilog   HDL   File
C:/altera/quartus51/bin/Thesis/RECURSIVEMULTIPLIER.v

RECURSIVEMULTIPLIER Analysis & Synthesis Resource Usage Summary

Resource   Usage
Total logic elements   2233
-- Combinational with no register   2057
-- Register only   16
-- Combinational with a register   160

Logic element usage by number of LUT inputs
-- 4 input functions   1389
-- 3 input functions   142
-- 2 input functions   686
-- 1 input functions   0
-- 0 input functions   0
-- Combinational cells for routing   0

Logic elements by mode
-- normal mode 2155
-- arithmetic mode   78
-- qfbk mode   0
-- register cascade mode   0
-- synchronous clear/load mode   0
-- asynchronous clear/load mode   0

Total registers   176
Total logic cells in carry chains   80
I/O pins   129
Maximum fan-out node   CK
Maximum fan-out   176
Total fan-out   7610
Average fan-out   3.22

without limitation, that your use is for the sole purpose of programming logic devices manufactured by Altera and sold by Altera or its authorized distributors. Please refer to the applicable agreement for further details.

RECURSIVEMULTIPLIER Interconnect Usage Summary

Interconnect Resource Type	Usage
C16 interconnects	125 / 2,286 (5 %)
C4 interconnects	1,203 / 31,320 (4 %)
C8 interconnects	473 / 7,272 (7 %)
DIFFIOCLKs	0 / 16 (0 %)
DQS bus muxes	0 / 56 (0 %)
DQS-32 I/O buses	0 / 4 (0 %)
DQS-8 I/O buses	0 / 16 (0 %)
Direct links	238 / 44,740 (< 1 %)
Fast regional clocks	0 / 8 (0 %)
Global clocks	1 / 16 (6 %)
I/O buses	11 / 208 (5 %)
LUT chains	110 / 9,513 (1 %)
Local routing interconnects	965 / 10,570 (9 %)
R24 interconnects	76 / 2,280 (3 %)
R4 interconnects	1,102 / 62,520 (2 %)
R8 interconnects	476 / 10,410 (5 %)
Regional clocks	0 / 16 (0 %)

RECURSIVEMULTIPLIER Analysis & Synthesis Settings

Option	Setting	Default Value
Top-level entity name	RECURSIVEMULTIPLIER	RECURSIVEMULTIPLIER
Family name	Stratix	Stratix
Use smart compilation	Off	Off
Restructure Multiplexers	Auto	Auto
Create Debugging Nodes for IP Cores	Off	Off
Preserve fewer node names	On	On
Disable OpenCore Plus hardware evaluation	Off	Off
Verilog Version	Verilog_2001	Verilog_2001
VHDL Version	VHDL93	VHDL93
State Machine Processing	Auto	Auto
Extract Verilog State Machines	On	On
Extract VHDL State Machines	On	On
Add Pass-Through Logic to Inferred RAMs	On	On
DSP Block Balancing	Auto	Auto
Maximum DSP Block Usage	-1	-1
NOT Gate Push-Back	On	On
Power-Up Don't Care	On	On
Remove Redundant Logic Cells	Off	Off
Remove Duplicate Registers	On	On
Ignore CARRY Buffers	Off	Off
Ignore CASCADE Buffers	Off	Off
Ignore GLOBAL Buffers	Off	Off
Ignore ROW GLOBAL Buffers	Off	Off
Ignore LCELL Buffers	Off	Off
Ignore SOFT Buffers	On	On
Limit AHDL Integers to 32 Bits	Off	Off
Optimization Technique -- Stratix/Stratix GX	Balanced	Balanced
Carry Chain Length -- Stratix/Stratix GX/Cyclone/MAX II/Cyclone II	70	70
Auto Carry Chains	On	On
Auto Open-Drain Pins	On	On
Remove Duplicate Logic	On	On
Perform WYSIWYG Primitive Resynthesis	Off	Off
Perform gate-level register retiming	Off	Off
Allow register retiming to trade off Tsu/Tco with Fmax	On	On
Auto ROM Replacement	On	On
Auto RAM Replacement	On	On
Auto DSP Block Replacement	On	On
Auto Shift Register Replacement	On	On
Auto Clock Enable Replacement	On	On
Allow Synchronous Control Signals	On	On
Force Use of Synchronous Clear Signals	Off	Off
Auto RAM Block Balancing	On	On

```
Auto Resource Sharing Off     Off
Allow Any RAM Size For Recognition     Off     Off
Allow Any ROM Size For Recognition     Off     Off
Allow Any Shift Register Size For Recognition        Off     Off
Maximum Number of M512 Memory Blocks  -1      -1
Maximum Number of M4K Memory Blocks   -1      -1
Maximum Number of M-RAM Memory Blocks -1      -1
Ignore translate_off and translate_on Synthesis Directives  Off     Off
Show Parameter Settings Tables in Synthesis Report  On      On
Ignore Maximum Fan-Out Assignments    Off     Off
Retiming Meta-Stability Register Sequence Length    2       2
PowerPlay Power Optimization  Normal compilation    Normal compilation
HDL message level      Level2  Level2


RECURSIVEMULTIPLIER Fitter Settings

Option  Setting Default Value
Device  AUTO
SignalProbe signals routed during normal compilation Off     Off
Use smart compilation  Off     Off
Router Timing Optimization Level      Normal  Normal
Placement Effort Multiplier   1.0     1.0
Router Effort Multiplier      1.0     1.0
Optimize Hold Timing  IO Paths and Minimum TPD Paths       IO   Paths   and   Minimum   TPD
Paths
Optimize Fast-Corner Timing   Off     Off
Optimize Timing       Normal compilation    Normal compilation
Optimize IOC Register Placement for Timing   On      On
Limit to One Fitting Attempt  Off     Off
Final Placement Optimizations Automatically  Automatically
Fitter Aggressive Routability Optimizations  Automatically  Automatically
Fitter Initial Placement Seed 1      1
Slow Slew Rate Off     Off
PCI I/O Off    Off
Weak Pull-Up Resistor  Off     Off
Enable Bus-Hold Circuitry     Off     Off
Auto Global Memory Control Signals    Off     Off
Auto Packed Registers -- Stratix/Stratix GX  Auto    Auto
Auto Delay Chains      On      On
Auto Merge PLLs        On      On
Perform Physical Synthesis for Combinational Logic   Off     Off
Perform Register Duplication  Off     Off
Perform Register Retiming     Off     Off
Perform Asynchronous Signal Pipelining       Off     Off
Fitter Effort  Auto Fit      Auto Fit
Physical Synthesis Effort Level      Normal  Normal
Logic Cell Insertion - Logic Duplication     Auto    Auto
Auto Register Duplication     Off     Off
Auto Global Clock      On      On
Auto Global Register Control Signals  On      On


RECURSIVEMULTIPLIER Fitter Settings

Option  Setting Default Value
Device  AUTO
SignalProbe signals routed during normal compilation Off     Off
Use smart compilation  Off     Off
Router Timing Optimization Level      Normal  Normal
Placement Effort Multiplier   1.0     1.0
Router Effort Multiplier      1.0     1.0
Optimize Hold Timing  IO Paths and Minimum TPD Paths       IO   Paths   and   Minimum   TPD
Paths
Optimize Fast-Corner Timing   Off     Off
Optimize Timing       Normal compilation    Normal compilation
Optimize IOC Register Placement for Timing   On      On
Limit to One Fitting Attempt  Off     Off
Final Placement Optimizations Automatically  Automatically
Fitter Aggressive Routability Optimizations  Automatically  Automatically
Fitter Initial Placement Seed 1      1
```

```
Slow Slew Rate Off     Off
PCI I/O Off     Off
Weak Pull-Up Resistor  Off     Off
Enable Bus-Hold Circuitry     Off     Off
Auto Global Memory Control Signals    Off     Off
Auto Packed Registers -- Stratix/Stratix GX  Auto    Auto
Auto Delay Chains      On      On
Auto Merge PLLs        On      On
Perform Physical Synthesis for Combinational Logic  Off     Off
Perform Register Duplication  Off     Off
Perform Register Retiming     Off     Off
Perform Asynchronous Signal Pipelining       Off     Off
Fitter Effort  Auto Fit       Auto Fit
Physical Synthesis Effort Level       Normal  Normal
Logic Cell Insertion - Logic Duplication     Auto    Auto
Auto Register Duplication     Off     Off
Auto Global Clock      On      On
Auto Global Register Control Signals  On      On


Date: 08/04/2006 09:47:32
Analysis Type: slack
Compiler Settings: RECURSIVEMULTIPLIER
Device: EP1S10F484C5

Timing Analyzer Summary
------------------------------------------------------------------------------------
Timing Analyzer Summary
------------------------------------------------------------------------------------
Path Number    : 1
Type           : Worst-case tsu
Slack          : N/A
Required Time  : None
Actual Time    : 33.769 ns
From           : AH[6]
To             : SHIFTSUBMULT2[47]
From Clock     : --
To Clock       : CK
Failed Paths   : 0

Path Number    : 2
Type           : Worst-case tco
Slack          : N/A
Required Time  : None
Actual Time    : 8.064 ns
From           : OUT[50]~reg0
To             : OUT[50]
From Clock     : CK
To Clock       : --
Failed Paths   : 0

Path Number    : 3
Type           : Worst-case th
Slack          : N/A
Required Time  : None
Actual Time    : -2.470 ns
From           : XH[1]
To             : SHIFTSUBMULT3[17]
From Clock     : --
To Clock       : CK
Failed Paths   : 0

Path Number    : 4
Type           : Clock Setup: 'CK'
Slack          : N/A
Required Time  : None
Actual Time    : 209.64 MHz ( period = 4.770 ns )
From           : SHIFTSUBMULT3[34]
To             : OUT[60]~reg0
From Clock     : CK
To Clock       : CK
```

```
Failed Paths    : 0

Path Number     : 5
Type            : Total number of failed paths
Slack           :
Required Time   :
Actual Time     :
From            :
To              :
From Clock      :
To Clock        :
Failed Paths    : 0

-------------------------------------------------------------------------------------------
```

```
 -----------------------------------------------------------------------------------
 -- NC              : No Connect. This pin has no internal connection to the device.
 -- VCCINT          : Dedicated power pin, which MUST be connected to VCC  (1.5V).
 -- VCCIO           : Dedicated power pin, which MUST be connected to VCC
 --                   of its bank.
 --                                  Bank 1:        3.3V
 --                                  Bank 2:        3.3V
 --                                  Bank 3:        3.3V
 --                                  Bank 4:        3.3V
 --                                  Bank 5:        3.3V
 --                                  Bank 6:        3.3V
 --                                  Bank 7:        3.3V
 --                                  Bank 8:        3.3V
 --                                  Bank 9:        3.3V
 --                                  Bank 10:       3.3V
 --                                  Bank 11:       3.3V
 --                                  Bank 12:       3.3V
 -- GND             : Dedicated ground pin. Dedicated GND pins MUST be connected to GND.
 --                                  It can also be used to report unused dedicated pins.
The connection
 --                                  on the board for unused dedicated pins depends on
whether this will
 --                                  be used in a future design. One example is device
migration. When
 --                                  using device migration, refer to the device pin-
tables. If it is a
 --                                  GND pin in the pin table or if it will not be used
in a future design
 --                                  for another purpose the it MUST be connected to GND.
If it is an unused
 --                                  dedicated pin, then it can be connected to a valid
signal on the board
 --                                  (low, high, or toggling) if that signal is required
for a different
 --                                  revision of the design.
 -- GND+            : Unused input pin. It can also be used to report unused dual-purpose
pins.
 --                                  This pin should be connected to GND. It may also be
connected  to a
 --                                  valid signal  on the board  (low, high, or toggling)
if that signal
 --                                  is required for a different revision of the design.
 -- GND*            : Unused  I/O  pin.   This pin can either be left unconnected or
 --                   connected to GND.  Connecting this pin to GND will improve the
 --                   device's immunity to noise.
 -- RESERVED        : Unused I/O pin, which MUST be left unconnected.
 -- RESERVED_INPUT     : Pin is tri-stated and should be connected to the board.
 --  RESERVED_INPUT_WITH_WEAK_PULLUP      : Pin is tri-stated with internal weak pull-up
resistor.
 -----------------------------------------------------------------------------------

Quartus II Version 5.1 Build 176 10/26/2005 SJ Web Edition
CHIP  "RECURSIVEMULTIPLIER"  ASSIGNED TO AN: EP1S10F484C5
```

Pin Name/Usage	Location	Dir.	I/O Standard	Voltage	I/O Bank	User Assignment
VCCINT	A1	power		1.5V		
GND	A2	gnd				
VCCIO4	A3	power		3.3V	4	
GND*	A4				4	
GND*	A5				4	
GND*	A6				4	
OUT[8]	A7	output	LVTTL		4	N
XL[11]	A8	input	LVTTL		4	N
GND	A9	gnd				
VCCIO4	A10	power		3.3V	4	
AH[0]	A11	input	LVTTL		4	N
XH[5]	A12	input	LVTTL		9	N
VCCIO3	A13	power		3.3V	3	
GND	A14	gnd				
AL[15]	A15	input	LVTTL		3	N
OUT[28]	A16	output	LVTTL		3	N
OUT[40]	A17	output	LVTTL		3	N
OUT[1]	A18	output	LVTTL		3	N
GND*	A19				3	
VCCIO3	A20	power		3.3V	3	
GND	A21	gnd				
VCCINT	A22	power		1.5V		
GND	AA1	gnd				
GND*	AA2				7	
GND*	AA3				7	
GND*	AA4				7	
GND*	AA5				7	
GND*	AA6				7	
GND*	AA7				7	
AH[15]	AA8	input	LVTTL		7	N
NC	AA9					
NC	AA10					
GND+	AA11				7	

OUT[61]	AA12	output	LVTTL		11	N
OUT[50]	AA13	output	LVTTL		11	N
GND+	AA14				8	
GND*	AA15				8	
GND*	AA16				8	
GND*	AA17				8	
GND*	AA18				8	
GND*	AA19				8	
GND*	AA20				8	
GND*	AA21				8	
GND	AA22	gnd				
VCCINT	AB1	power		1.5V		
GND	AB2	gnd				
VCCIO7	AB3	power		3.3V	7	
GND*	AB4				7	
GND*	AB5				7	
GND*	AB6				7	
GND*	AB7				7	
GND*	AB8				7	
GND	AB9	gnd				
VCCIO7	AB10	power		3.3V	7	
AH[6]	AB11	input	LVTTL		7	N
GND*	AB12				11	
VCCIO8	AB13	power		3.3V	8	
GND	AB14	gnd				
GND*	AB15				8	
GND*	AB16				8	
GND*	AB17				8	
GND*	AB18				8	
GND*	AB19				8	
VCCIO8	AB20	power		3.3V	8	
GND	AB21	gnd				
VCCINT	AB22	power		1.5V		
GND	B1	gnd				
GND*	B2				4	

GND*	B3				4	
GND*	B4				4	
GND*	B5				4	
GND*	B6				4	
OUT[4]	B7	output	LVTTL		4	N
AH[9]	B8	input	LVTTL		4	N
NC	B9					
NC	B10					
AH[2]	B11	input	LVTTL		4	N
AL[6]	B12	input	LVTTL		9	N
AL[1]	B13	input	LVTTL		9	N
AL[13]	B14	input	LVTTL		3	N
OUT[18]	B15	output	LVTTL		3	N
OUT[25]	B16	output	LVTTL		3	N
OUT[32]	B17	output	LVTTL		3	N
OUT[44]	B18	output	LVTTL		3	N
OUT[16]	B19	output	LVTTL		3	N
GND*	B20				3	
GND*	B21				3	
GND	B22	gnd				
VCCIO5	C1	power		3.3V	5	
GND*	C2				4	
GND*	C3				4	
GND*	C4				4	
GND*	C5				4	
GND*	C6				4	
XL[3]	C7	input	LVTTL		4	N
XL[8]	C8	input	LVTTL		4	N
XH[7]	C9	input	LVTTL		4	N
NC	C10					
NC	C11					
XH[3]	C12	input	LVTTL		9	N
AH[1]	C13	input	LVTTL		9	N
OUT[19]	C14	output	LVTTL		3	N
XH[8]	C15	input	LVTTL		3	N

AL[4]	C16	input	LVTTL		3	N
OUT[33]	C17	output	LVTTL		3	N
GND*	C18				3	
GND*	C19				3	
GND*	C20				3	
GND*	C21				3	
VCCIO2	C22	power		3.3V	2	
GND*	D1				5	
GND*	D2				5	
GND*	D3				4	
GND*	D4				4	
GND*	D5				4	
GND*	D6				4	
XL[13]	D7	input	LVTTL		4	N
AH[5]	D8	input	LVTTL		4	N
AH[3]	D9	input	LVTTL		4	N
NC	D10					
NC	D11					
XH[15]	D12	input	LVTTL		9	N
XH[14]	D13	input	LVTTL		3	N
XH[11]	D14	input	LVTTL		3	N
OUT[20]	D15	output	LVTTL		3	N
OUT[24]	D16	output	LVTTL		3	N
GND*	D17				3	
OUT[34]	D18	output	LVTTL		3	N
GND*	D19				3	
GND*	D20				3	
GND*	D21				2	
GND*	D22				2	
GND*	E1				5	
GND*	E2				5	
NC	E3					
NC	E4					
GND*	E5				4	
GND*	E6				4	

AH[12]	E7	input	LVTTL		4	N
AH[11]	E8	input	LVTTL		4	N
XH[0]	E9	input	LVTTL		4	N
NC	E10					
NC	E11					
NC	E12					
XH[2]	E13	input	LVTTL		3	N
OUT[37]	E14	output	LVTTL		3	N
OUT[58]	E15	output	LVTTL		3	N
OUT[63]	E16	output	LVTTL		3	N
OUT[38]	E17	output	LVTTL		3	N
GND*	E18				3	
GND*	E19				2	
GND*	E20				2	
GND*	E21				2	
GND*	E22				2	
GND*	F1				5	
GND*	F2				5	
GND*	F3				5	
GND*	F4				5	
GND*	F5				5	
XL[5]	F6	input	LVTTL		4	N
XL[6]	F7	input	LVTTL		4	N
AH[13]	F8	input	LVTTL		4	N
XH[4]	F9	input	LVTTL		4	N
XH[6]	F10	input	LVTTL		4	N
VCCG_PLL5	F11	power		1.5V	1	
GNDA_PLL5	F12	gnd				
VCC_PLL5_OUTA	F13	power		3.3V	9	
XH[9]	F14	input	LVTTL		3	N
AL[8]	F15	input	LVTTL		3	N
OUT[48]	F16	output	LVTTL		3	N
OUT[31]	F17	output	LVTTL		3	N
GND*	F18				2	
NC	F19					

NC	F20					
OUT[43]	F21	output	LVTTL		2	N
OUT[39]	F22	output	LVTTL		2	N
OUT[54]	G1	output	LVTTL		5	N
XL[15]	G2	input	LVTTL		5	N
GND*	G3				5	
GND*	G4				5	
GND*	G5				5	
GND	G6	gnd				
GND*	G7				4	
AH[14]	G8	input	LVTTL		4	N
AH[8]	G9	input	LVTTL		4	N
TMS	G10	input			4	
GNDG_PLL5	G11	gnd				
TEMPDIODEp	G12					
VCCA_PLL5	G13	power		1.5V		
AL[5]	G14	input	LVTTL		3	N
GND	G15	gnd				
OUT[30]	G16	output	LVTTL		3	N
GND	G17	gnd				
GND*	G18				2	
AL[0]	G19	input	LVTTL		2	N
GND*	G20				2	
OUT[57]	G21	output	LVTTL		2	N
OUT[47]	G22	output	LVTTL		2	N
OUT[59]	H1	output	LVTTL		5	N
XL[2]	H2	input	LVTTL		5	N
OUT[29]	H3	output	LVTTL		5	N
GND*	H4				5	
GND	H5	gnd				
GND	H6	gnd				
GND	H7	gnd				
AH[10]	H8	input	LVTTL		4	N
NC	H9					
AL[14]	H10	input	LVTTL		4	N

TDO	H11	output			4	
TEMPDIODEn	H12					
nCONFIG	H13				3	
GND	H14	gnd				
GND	H15	gnd				
NC	H16					
OUT[22]	H17	output	LVTTL		2	N
GND	H18	gnd				
OUT[36]	H19	output	LVTTL		2	N
OUT[26]	H20	output	LVTTL		2	N
OUT[46]	H21	output	LVTTL		2	N
OUT[6]	H22	output	LVTTL		2	N
GND	J1	gnd				
OUT[3]	J2	output	LVTTL		5	N
OUT[7]	J3	output	LVTTL		5	N
OUT[21]	J4	output	LVTTL		5	N
GND	J5	gnd				
GND*	J6				5	
GND*	J7				4	
AL[11]	J8	input	LVTTL		4	N
XL[14]	J9	input	LVTTL		4	N
TRST	J10	input			4	
TDI	J11	input			4	
nSTATUS	J12				3	
DCLK	J13				3	
GND*	J14				3	
XH[12]	J15	input	LVTTL		3	N
OUT[0]	J16	output	LVTTL		3	N
OUT[52]	J17	output	LVTTL		2	N
GND	J18	gnd				
XH[13]	J19	input	LVTTL		2	N
OUT[60]	J20	output	LVTTL		2	N
GND*	J21				2	
GND	J22	gnd				
VCCIO5	K1	power		3.3V	5	

```
GND*                          : K2        :          :                       :          : 5
:
XL[10]                        : K3        : input    : LVTTL                 :          : 5
: N
VCCA_PLL4                     : K4        : power    :                       : 1.5V     :
:
VCCG_PLL4                     : K5        : power    :                       : 1.5V     : 1
:
GND*                          : K6        :          :                       :          : 5
:
GND*                          : K7        :          :                       :          : 4
:
XH[10]                        : K8        : input    : LVTTL                 :          : 4
: N
TCK                           : K9        : input    :                       :          : 4
:
XL[1]                         : K10       : input    : LVTTL                 :          : 4
: N
GND                           : K11       : gnd      :                       :          :
:
VCCINT                        : K12       : power    :                       : 1.5V     :
:
CONF_DONE                     : K13       :          :                       :          : 3
:
XH[1]                         : K14       : input    : LVTTL                 :          : 3
: N
AL[9]                         : K15       : input    : LVTTL                 :          : 3
: N
OUT[49]                       : K16       : output   : LVTTL                 :          : 3
: N
OUT[35]                       : K17       : output   : LVTTL                 :          : 2
: N
VCCG_PLL1                     : K18       : power    :                       : 1.5V     : 1
:
VCCA_PLL1                     : K19       : power    :                       : 1.5V     :
:
OUT[10]                       : K20       : output   : LVTTL                 :          : 2
: N
OUT[13]                       : K21       : output   : LVTTL                 :          : 2
: N
VCCIO2                        : K22       : power    :                       : 3.3V     : 2
:
GND+                          : L1        :          :                       :          : 5
:
CK                            : L2        : input    : LVTTL                 :          : 5
: N
GND+                          : L3        :          :                       :          : 5
:
GNDA_PLL4                     : L4        : gnd      :                       :          :
:
GNDG_PLL4                     : L5        : gnd      :                       :          :
:
GND*                          : L6        :          :                       :          : 5
:
XL[7]                         : L7        : input    : LVTTL                 :          : 4
: N
~DATA0~ / RESERVED_INPUT      : L8        : input    : LVTTL                 :          : 4
: N
VCCINT                        : L9        : power    :                       : 1.5V     :
:
GND                           : L10       : gnd      :                       :          :
:
VCCINT                        : L11       : power    :                       : 1.5V     :
:
GND                           : L12       : gnd      :                       :          :
:
VCCINT                        : L13       : power    :                       : 1.5V     :
:
GND                           : L14       : gnd      :                       :          :
:
```

AL[12]	L15	input	LVTTL		3	N
AL[2]	L16	input	LVTTL		3	N
OUT[15]	L17	output	LVTTL		2	N
GNDG_PLL1	L18	gnd				
GNDA_PLL1	L19	gnd				
GND+	L20				2	
GND+	L21				2	
GND+	L22				2	
GND+	M1				6	
GND+	M2				6	
GND+	M3				6	
GNDA_PLL3	M4	gnd				
VCCA_PLL3	M5	power		1.5V		
GND*	M6				6	
XL[4]	M7	input	LVTTL		7	N
OUT[11]	M8	output	LVTTL		7	N
GND	M9	gnd				
VCCINT	M10	power		1.5V		
GND	M11	gnd				
VCCINT	M12	power		1.5V		
GND	M13	gnd				
VCCINT	M14	power		1.5V		
AL[3]	M15	input	LVTTL		3	N
GND*	M16				3	
GND*	M17				1	
VCCA_PLL2	M18	power		1.5V		
GNDA_PLL2	M19	gnd				
GND+	M20				1	
GND+	M21				1	
GND+	M22				1	
VCCIO6	N1	power		3.3V	6	
GND*	N2				6	
GND*	N3				6	
GNDG_PLL3	N4	gnd				
VCCG_PLL3	N5	power		1.5V	1	

GND*	N6				6	
GND*	N7				7	
AH[7]	N8	input	LVTTL		7	N
nIO_PULLUP	N9				7	
OUT[5]	N10	output	LVTTL		7	N
VCCINT	N11	power		1.5V		
GND	N12	gnd				
GND*	N13				8	
GND*	N14				8	
OUT[55]	N15	output	LVTTL		8	N
GND*	N16				8	
GND*	N17				1	
VCCG_PLL2	N18	power		1.5V	1	
GNDG_PLL2	N19	gnd				
GND*	N20				1	
GND*	N21				1	
VCCIO1	N22	power		3.3V	1	
GND	P1	gnd				
GND*	P2				6	
GND*	P3				6	
GND*	P4				6	
GND	P5	gnd				
GND*	P6				6	
GND*	P7				7	
XL[12]	P8	input	LVTTL		7	N
OUT[12]	P9	output	LVTTL		7	N
XL[0]	P10	input	LVTTL		7	N
nCEO	P11				7	
MSEL1	P12				8	
OUT[45]	P13	output	LVTTL		8	N
OUT[27]	P14	output	LVTTL		8	N
OUT[51]	P15	output	LVTTL		8	N
GND*	P16				8	
GND*	P17				1	
GND	P18	gnd				

```
GND*                              : P19        :          :                      :          : 1
:
GND*                              : P20        :          :                      :          : 1
:
GND*                              : P21        :          :                      :          : 1
:
GND                                : P22        : gnd      :                      :          :
:
GND*                              : R1         :          :                      :          : 6
:
GND*                              : R2         :          :                      :          : 6
:
GND*                              : R3         :          :                      :          : 6
:
GND*                              : R4         :          :                      :          : 6
:
NC                                 : R5         :          :                      :          :
:
GND                                : R6         : gnd      :                      :          :
:
GND                                : R7         : gnd      :                      :          :
:
AL[7]                             : R8         : input    : LVTTL                :          : 7
: N
GND                                : R9         : gnd      :                      :          :
:
VCCSEL                            : R10        :          :                      :          : 7
:
nCE                               : R11        :          :                      :          : 7
:
MSEL2                             : R12        :          :                      :          : 8
:
PLL_ENA                           : R13        :          :                      :          : 8
:
NC                                 : R14        :          :                      :          :
:
GND*                              : R15        :          :                      :          : 8
:
GND                                : R16        : gnd      :                      :          :
:
GND                                : R17        : gnd      :                      :          :
:
GND                                : R18        : gnd      :                      :          :
:
GND*                              : R19        :          :                      :          : 1
:
GND*                              : R20        :          :                      :          : 1
:
GND*                              : R21        :          :                      :          : 1
:
GND*                              : R22        :          :                      :          : 1
:
GND*                              : T1         :          :                      :          : 6
:
GND*                              : T2         :          :                      :          : 6
:
GND*                              : T3         :          :                      :          : 6
:
GND*                              : T4         :          :                      :          : 6
:
GND*                              : T5         :          :                      :          : 6
:
GND                                : T6         : gnd      :                      :          :
:
GND*                              : T7         :          :                      :          : 7
:
GND*                              : T8         :          :                      :          : 7
:
OUT[14]                           : T9         : output   : LVTTL                :          : 7
: N
```

```
OUT[9]                    : T10      : output : LVTTL           :         : 7
: N
GNDG_PLL6                 : T11      : gnd    :                 :         :
:
VCCA_PLL6                 : T12      : power  :                 : 1.5V    :
:
MSEL0                     : T13      :        :                 :         : 8
:
OUT[23]                   : T14      : output : LVTTL           :         : 8
: N
GND*                      : T15      :        :                 :         : 8
:
GND*                      : T16      :        :                 :         : 8
:
GND                       : T17      : gnd    :                 :         :
:
GND*                      : T18      :        :                 :         : 1
:
GND*                      : T19      :        :                 :         : 1
:
GND*                      : T20      :        :                 :         : 1
:
GND*                      : T21      :        :                 :         : 1
:
GND*                      : T22      :        :                 :         : 1
:
GND*                      : U1       :        :                 :         : 6
:
GND*                      : U2       :        :                 :         : 6
:
NC                        : U3       :        :                 :         :
:
NC                        : U4       :        :                 :         :
:
GND*                      : U5       :        :                 :         : 6
:
GND*                      : U6       :        :                 :         : 7
:
GND*                      : U7       :        :                 :         : 7
:
XL[9]                     : U8       : input  : LVTTL           :         : 7
: N
AH[4]                     : U9       : input  : LVTTL           :         : 7
: N
PORSEL                    : U10      :        :                 :         : 7
:
VCCG_PLL6                 : U11      : power  :                 : 1.5V    : 1
:
GNDA_PLL6                 : U12      : gnd    :                 :         :
:
VCC_PLL6_OUTA             : U13      : power  :                 : 3.3V    : 11
:
GND*                      : U14      :        :                 :         : 8
:
OUT[17]                   : U15      : output : LVTTL           :         : 8
: N
GND*                      : U16      :        :                 :         : 8
:
GND*                      : U17      :        :                 :         : 8
:
GND*                      : U18      :        :                 :         : 1
:
GND*                      : U19      :        :                 :         : 1
:
GND*                      : U20      :        :                 :         : 1
:
GND*                      : U21      :        :                 :         : 1
:
GND*                      : U22      :        :                 :         : 1
:
```

```
GND*                    : V1     :        :              :       : 6
:
GND*                    : V2     :        :              :       : 6
:
GND*                    : V3     :        :              :       : 6
:
GND*                    : V4     :        :              :       : 6
:
GND*                    : V5     :        :              :       : 7
:
GND*                    : V6     :        :              :       : 7
:
GND*                    : V7     :        :              :       : 7
:
GND*                    : V8     :        :              :       : 7
:
GND*                    : V9     :        :              :       : 7
:
NC                      : V10    :        :              :       :
:
NC                      : V11    :        :              :       :
:
NC                      : V12    :        :              :       :
:
AL[10]                  : V13    : input  : LVTTL        :       : 8
: N
GND*                    : V14    :        :              :       : 8
:
GND*                    : V15    :        :              :       : 8
:
GND*                    : V16    :        :              :       : 8
:
GND*                    : V17    :        :              :       : 8
:
GND*                    : V18    :        :              :       : 8
:
NC                      : V19    :        :              :       :
:
NC                      : V20    :        :              :       :
:
GND*                    : V21    :        :              :       : 1
:
GND*                    : V22    :        :              :       : 1
:
GND*                    : W1     :        :              :       : 6
:
GND*                    : W2     :        :              :       : 6
:
GND*                    : W3     :        :              :       : 7
:
GND*                    : W4     :        :              :       : 7
:
GND*                    : W5     :        :              :       : 7
:
GND*                    : W6     :        :              :       : 7
:
GND*                    : W7     :        :              :       : 7
:
GND*                    : W8     :        :              :       : 7
:
GND*                    : W9     :        :              :       : 7
:
NC                      : W10    :        :              :       :
:
NC                      : W11    :        :              :       :
:
OUT[2]                  : W12    : output : LVTTL        :       : 11
: N
OUT[42]                 : W13    : output : LVTTL        :       : 8
: N
```

```
OUT[53]                         : W14     : output : LVTTL          :        : 8
: N
GND*                            : W15     :        :                :        : 8
:
GND*                            : W16     :        :                :        : 8
:
GND*                            : W17     :        :                :        : 8
:
GND*                            : W18     :        :                :        : 8
:
GND*                            : W19     :        :                :        : 8
:
GND*                            : W20     :        :                :        : 8
:
GND*                            : W21     :        :                :        : 1
:
GND*                            : W22     :        :                :        : 1
:
VCCIO6                          : Y1      : power  :                : 3.3V   : 6
:
GND*                            : Y2      :        :                :        : 7
:
GND*                            : Y3      :        :                :        : 7
:
GND*                            : Y4      :        :                :        : 7
:
GND*                            : Y5      :        :                :        : 7
:
GND*                            : Y6      :        :                :        : 7
:
GND*                            : Y7      :        :                :        : 7
:
GND*                            : Y8      :        :                :        : 7
:
GND*                            : Y9      :        :                :        : 7
:
NC                              : Y10     :        :                :        :
:
NC                              : Y11     :        :                :        :
:
OUT[62]                         : Y12     : output : LVTTL          :        : 11
: N
OUT[41]                         : Y13     : output : LVTTL          :        : 11
: N
OUT[56]                         : Y14     : output : LVTTL          :        : 8
: N
GND*                            : Y15     :        :                :        : 8
:
GND*                            : Y16     :        :                :        : 8
:
GND*                            : Y17     :        :                :        : 8
:
GND*                            : Y18     :        :                :        : 8
:
GND*                            : Y19     :        :                :        : 8
:
GND*                            : Y20     :        :                :        : 8
:
GND*                            : Y21     :        :                :        : 8
:
VCCIO1                          : Y22     : power  :                : 3.3V   : 1
:
```

VITA AUCTORIS

Kevin Biswas was born on February 1, 1981 in Ottawa, Ontario, Canada. At a young age, he moved to Windsor, Ontario. He attended Vincent Massey Secondary School where he was enrolled in the enriched mathematics and science program.

In 2000, he enrolled in Electrical Engineering at the University of Windsor under a Yves Landry Memorial Scholarship. He received the B.A.Sc. degree in Electrical Engineering in 2004, graduating with Great Distinction (12.28/13.0), and was accorded positions on the Dean's and President's Honour Rolls every year. Kevin also gained invaluable industrial experience through his enrolment in the co-operative education program. His professional employment includes electrical engineering research positions at the Ford Powertrain NVH Research and Development group for three semesters, under the supervision of Dr. Jimi Tjong. In 2003, he was awarded a Natural Sciences and Engineering Research Council of Canada (NSERC) Undergraduate Student Research Award to work at the University of Windsor DSP Research Group under the supervision of Dr. Majid Ahmadi.

In 2004, Kevin pursued the M.A.Sc. degree in Electrical Engineering under the supervision of Dr. Majid Ahmadi in the Research Centre for Integrated Microsystems at the University of Windsor, and was funded by an NSERC Canada Graduate Scholarship. His areas of specialization have been computer arithmetic and VLSI design. In 2006, Kevin was awarded an NSERC Canada Graduate Scholarship for doctoral studies.

He intends on commencing his studies towards the Ph.D. degree in Electrical and Computer Engineering in the fall of 2006.