Fixed-width digital multipliers based on recursive architectures.

Kevin Biswas
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/7094

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
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

© 2006 Kevin Biswas
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
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

ABSTRACT .................................................................................................................................. iii
DEDICATION ............................................................................................................................ iv
ACKNOWLEDGEMENTS ......................................................................................................... v
LIST OF TABLES ................................................................................................................... viii
LIST OF FIGURES ................................................................................................................... ix

CHAPTER

1. INTRODUCTION TO COMPUTER ARITHMETIC
   1.1 Overview of Computer Arithmetic ................................................................. 1
   1.2 Thesis Highlights ......................................................................................... 1
   1.3 Thesis Organization .................................................................................. 2

2. DIGITAL MULTIPLICATION OVERVIEW
   2.1 Basics of Digital Multiplication .................................................................. 5
   2.2 Sequential Multiplication .......................................................................... 6
   2.3 Parallel Multiplication ............................................................................. 8
   2.4 Floating Point Number System and Multiplication .................................. 12

3. FIXED-WIDTH MULTIPLICATION
   3.1 Overview of Fixed-Width Multiplication .................................................. 15
   3.2 Truncated Multipliers ............................................................................. 16
   3.3 Truncation Schemes for Fixed-Width Multipliers .................................... 18

4. THE RECURSIVE MULTIPLIER
   4.1 Overview of the Recursive Multiplication Algorithm .............................. 25
   4.2 Recursive Multiplication Architecture .................................................. 27

5. FIXED-WIDTH RECURSIVE MULTIPLIER ARCHITECTURES
   5.1 Proposed Truncation Schemes for Recursive Multipliers ...................... 31
   5.2 Error Simulation and Analysis ............................................................... 35
   5.3 Complexity Analysis .............................................................................. 39
LIST OF TABLES

Table 3.1: Error Statistics for Constant Correction Multipliers............................... 20
Table 3.2: Complexity Savings From Truncation of Array and Dadda Multipliers....... 21
Table 5.1: Error Statistics for Proposed Fixed-Width Recursive Multipliers.............. 36
Table 5.2: Error Statistics for Different Fixed-Width Multipliers............................. 37
Table 5.3: Complexity Savings Comparison of Each Scheme (2n = 8)....................... 40
Table 5.4: Complexity Savings Comparison for Larger Multipliers (2n = 16, 32, 64).... 40
Table 5.5: Overall Performance Comparison of Proposed Truncation Schemes.......... 41
Table 6.1: FPGA Simulation Results............................................................................ 44
Table 7.1: Error Simulation Results for Fixed-Width Two-Level Recursive Multipliers 49
Table 7.2: Maximum Positive Error for Different Levels of Recursion..................... 51
Table 7.3: Approximate Complexity Savings for Different Levels of Recursion........ 51
LIST OF FIGURES

Figure 2.1: Example of Pen and Paper Multiplication........................................................... 5
Figure 2.2: Partial Product Array for a 16-bit Multiplication............................................... 6
Figure 2.3: Sequential Right-Shift Multiplier.......................................................................... 7
Figure 2.4: Multiplication Performed Using Radix-4............................................................. 8
Figure 2.5: Standard Layout of an Array Multiplier............................................................... 9
Figure 2.6: Flow Diagram of a Column Compression Multiplier ........................................... 10
Figure 2.7: Dot Diagrams of Dadda and Wallace Multipliers................................................ 11
Figure 2.8: IEEE Floating Point Standard Word Widths ....................................................... 13
Figure 2.9: A Floating Point Multiplication Scheme............................................................. 13
Figure 3.1: Standard and Truncated Array Multipliers........................................................ 17
Figure 3.2: Standard and Truncated Tree (Dadda) Multipliers............................................ 17
Figure 3.3: Truncated Partial Products Matrix with Constant Correction......................... 19
Figure 3.4: Truncated Partial Products Matrix with Data-Dependent Correction.............. 22
Figure 4.1: Block Diagram of Recursive Multiplier Architecture...................................... 27
Figure 4.2: Another Block Diagram of Recursive Multiplier Architecture......................... 27
Figure 4.3: Full Dot Diagram of a Recursive Multiplier with n-bit Onput Operands............ 28
Figure 4.4: Delay Comparison of Array, Dadda and Recursive Multipliers....................... 29
Figure 5.1: Fixed-Width Recursive Multiplier ..................................................................... 32
Figure 5.2: Proposal #1 ....................................................................................................... 33
Figure 5.3: Proposal #2 ....................................................................................................... 34
Figure 5.4: Proposal #3 ....................................................................................................... 34
Figure 5.5: Proposal #4

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

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

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 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 $M \times N$ bit digital multiplication is determined similarly to traditional pen and paper decimal multiplication. For example, multiplication of multiplier $X = x_n x_{n-1} ... x_2 x_1 x_0$ and multiplicand $A = a_m a_{m-1} ... a_2 a_1 a_0$ yields the final product $(n+m)$-bit product:

$$P = [p_{n+m}, p_{n+m-2}, p_{n+m-3}, ... p_2, p_0] = x_n(a_m, a_{m-1}, ... a_2, a_1, a_0) + x_{n-1}(a_m, a_{m-1}, ... , a_2, a_1, a_0) + ... + x_0(a_m, a_{m-1}, ... , a_2, a_1, a_0)$$

This multiplication can be illustrated in Figure 2.1, below.

![Figure 2.1: Example of Pen and Paper Multiplication](image)

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].

![Partial Product Array for a 16-bit Multiplication](image)

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.
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.
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

![Figure 2.4: Multiplication Performed Using Radix-4](image-url)
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 theadders 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.

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.

![Flow Diagram of a Column Compression Multiplier](image)

**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

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
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 \( n \times n \) 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:

\[
P_{\text{trunc}} = \sum_{k=n}^{2n-1} p_k \cdot 2^k
\]

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.
Figure 3.1: Standard and Truncated Array Multipliers

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.

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

\[
P' = P + E_{\text{reduct}} + E_{\text{round}} + C,
\]

where \( P \) is the true product, \( E_{\text{reduct}} \) and \( E_{\text{round}} \) 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]:

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

where \( E_{\text{total}} = -\frac{1}{4} \sum_{q=0}^{n-k-1} (q+1) \cdot 2^{-(2^n-q)} - 2^{-(n+1)} \cdot (1 - 2^{-k}) \).
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>
<thead>
<tr>
<th>( n )</th>
<th>( k )</th>
<th>( E_{\text{avg}} )</th>
<th>Variance</th>
<th>( E_{\text{max}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>(-9.766 \times 10^{-4})</td>
<td>0.1667</td>
<td>2.5039</td>
</tr>
<tr>
<td>2</td>
<td>(6.152 \times 10^{-2})</td>
<td>0.1040</td>
<td>1.2539</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(6.152 \times 10^{-2})</td>
<td>0.0903</td>
<td>0.7539</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(-1.660 \times 10^{-2})</td>
<td>0.0842</td>
<td>0.6289</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(-9.766 \times 10^{-4})</td>
<td>0.0834</td>
<td>0.5352</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(-1.953 \times 10^{-3})</td>
<td>0.0833</td>
<td>0.5000</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>(-3.815 \times 10^{-6})</td>
<td>0.2917</td>
<td>5.5000</td>
</tr>
<tr>
<td>2</td>
<td>(6.250 \times 10^{-2})</td>
<td>0.1354</td>
<td>2.7500</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(6.250 \times 10^{-2})</td>
<td>0.0983</td>
<td>1.5000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(-1.563 \times 10^{-4})</td>
<td>0.0861</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(-3.815 \times 10^{-6})</td>
<td>0.0839</td>
<td>0.7188</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>(7.629 \times 10^{-6})</td>
<td>0.0833</td>
<td>0.5000</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>$k$</th>
<th>% Complexity Savings (Array)</th>
<th>% Complexity Savings (Dadda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>35.4</td>
<td>41.8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>23.9</td>
<td>28.8</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>15.2</td>
<td>18.6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9.28</td>
<td>11.9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.36</td>
<td>6.14</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>42.6</td>
<td>46.6</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>36.6</td>
<td>40.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>31.0</td>
<td>34.2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>26.2</td>
<td>29.3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>21.7</td>
<td>24.2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

_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](image)

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:

\[
P = N_1 \cdot N_2 \\
= a_0 b_0 + (a_0 b_1 + a_1 b_0)X + a_1 b_1 X^2 \\
= p_0 + p_1 X + p_2 X^2
\]

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 $p_0, p_1, \text{and } p_2$:

$$q_1 = p_1 + p_0 + p_2$$

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

$$p_0 = q_0$$
$$p_1 = q_1 - q_0 - q_2$$
$$p_2 = q_2$$

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 $2n \times 2n$ bit multiplication may be carried out through four $n \times n$ 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:

$$Y = A \cdot X$$
$$= (A_H \times 2^n + A_L) \cdot (X_H \times 2^n + X_L)$$
$$= A_H \times X_H \times 2^{2n} + (A_L \cdot X_H + A_H \cdot X_L) \times 2^n + A_L \cdot X_L,$$

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.

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-multippliers 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 Output Operands [5]](image)

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]:

\[
D_{\text{Array}} = 1 + 3(n-1) + 4\log_2(n-1)
\]

\[
D_{\text{Dadda}} = 1 + 3(2\log_2(n-1)) + 3\log_2(n+1)
\]

\[
D_{\text{Recursive}} = 7 + 9\log_2(n-2) + 3\log_2(n+1)
\]
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. It will 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:
\[ Y = A \cdot X \]
\[ = \left( A_H \times 2^n + A_L \right) \left( X_H \times 2^n + X_L \right) \]
\[ = A_H \cdot X_H \times 2^{2n} + (A_L \cdot X_H + A_H \cdot X_L) \times 2^n + A_L \cdot X_L. \]

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.

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 $A_H X_L$ or $A_L X_H$ to replace the least significant truncated term $A_L X_L$. 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.

In Proposal #2, the average of the two blocks, $A_H X_L$ and $A_L X_H$, is placed in the block of $A_L X_L$ 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 $A_H X_L / 2$ and $A_L X_H / 2$ 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, $A_L X_L$, is provided.
In Proposal #3, the most significant partial product bit, namely $a_{n-1}x_{n-1}$, generated by the block $A_LX_L$, is added at the least significant bit position of block $A_HX_H$. 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.

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, $A_LX_L$. Additional
correction bits, \( a_{n-2}x_{n-2}, a_{n-3}x_{n-3}, \ldots, a_0x_0 \), are added to positions right of the first bit. Mathematically, the correction term with \( d \) correction bits, \( 1 \leq d \leq n \), can be defined as:

\[
C_d = c_{2n-1}^{(d)} c_{2n-2}^{(d)} \cdots c_0^{(d)},
\]

where \( c_i^{(d)} = \begin{cases} 
2n-d \leq i \leq 2n-1 & \text{if } i = 0,
0 & \text{otherwise,}
\end{cases} \quad 0 \leq i \leq 2n-d-1.
\]

For example, when \( d = 2 \), the correction term \( C_2 \) contains only two bits and has a value of \( C_2 = a_{n-1}x_{n-1}2^{n-1} + a_{n-2}x_{n-2}2^{n-2} \). 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](image)

### 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 $2n = 6, 8 \text{ and } 10$. 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>
<thead>
<tr>
<th>$2n$</th>
<th>Correction Method</th>
<th>$E_{avg}$</th>
<th>$E_{max}^+$</th>
<th>$E_{max}^-$</th>
<th>$\sigma^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Full-width (With $A_1X_L$)</td>
<td>0.000</td>
<td>0.500</td>
<td>-0.500</td>
<td>0.083</td>
</tr>
<tr>
<td></td>
<td>Removal of $A_1X_L$</td>
<td>-0.191</td>
<td>0.500</td>
<td>-1.266</td>
<td>0.128</td>
</tr>
<tr>
<td></td>
<td>Proposal #1</td>
<td>-0.000</td>
<td>1.141</td>
<td>-1.156</td>
<td>0.128</td>
</tr>
<tr>
<td></td>
<td>Proposal #2</td>
<td>0.037</td>
<td>0.875</td>
<td>-0.906</td>
<td>0.109</td>
</tr>
<tr>
<td></td>
<td>Proposal #3</td>
<td>0.059</td>
<td>1.250</td>
<td>-0.828</td>
<td>0.173</td>
</tr>
<tr>
<td></td>
<td>Proposal #4 w/ 1 correction bit</td>
<td>-0.067</td>
<td>0.750</td>
<td>-0.828</td>
<td>0.102</td>
</tr>
<tr>
<td></td>
<td>Proposal #4 w/ n=3 correction bits</td>
<td>0.025</td>
<td>0.750</td>
<td>-0.688</td>
<td>0.098</td>
</tr>
<tr>
<td>8</td>
<td>Full-width (With $A_1X_L$)</td>
<td>0.000</td>
<td>0.500</td>
<td>-0.500</td>
<td>0.083</td>
</tr>
<tr>
<td></td>
<td>Removal of $A_1X_L$</td>
<td>-0.220</td>
<td>0.500</td>
<td>-1.379</td>
<td>0.128</td>
</tr>
<tr>
<td></td>
<td>Proposal #1</td>
<td>-0.004</td>
<td>1.316</td>
<td>-1.320</td>
<td>0.133</td>
</tr>
<tr>
<td></td>
<td>Proposal #2</td>
<td>0.014</td>
<td>0.938</td>
<td>-1.137</td>
<td>0.113</td>
</tr>
<tr>
<td></td>
<td>Proposal #3</td>
<td>0.030</td>
<td>1.250</td>
<td>-0.910</td>
<td>0.167</td>
</tr>
<tr>
<td></td>
<td>Proposed w/ 1 correction bit</td>
<td>-0.095</td>
<td>0.750</td>
<td>-0.910</td>
<td>0.101</td>
</tr>
<tr>
<td></td>
<td>Proposed w/ n=4 correction bits</td>
<td>0.014</td>
<td>0.750</td>
<td>-0.719</td>
<td>0.095</td>
</tr>
<tr>
<td>10</td>
<td>Full-width (With $A_1X_L$)</td>
<td>0.000</td>
<td>0.500</td>
<td>-0.500</td>
<td>0.083</td>
</tr>
<tr>
<td></td>
<td>Removal of $A_1X_L$</td>
<td>-0.235</td>
<td>0.500</td>
<td>-1.438</td>
<td>0.130</td>
</tr>
<tr>
<td></td>
<td>Proposal #1</td>
<td>-0.001</td>
<td>1.407</td>
<td>-1.408</td>
<td>0.136</td>
</tr>
<tr>
<td></td>
<td>Proposal #2</td>
<td>0.005</td>
<td>0.967</td>
<td>-1.253</td>
<td>0.114</td>
</tr>
<tr>
<td></td>
<td>Proposal #3</td>
<td>0.015</td>
<td>1.250</td>
<td>-0.954</td>
<td>0.165</td>
</tr>
<tr>
<td></td>
<td>Proposed w/ 1 correction bit</td>
<td>-0.110</td>
<td>0.750</td>
<td>-0.954</td>
<td>0.101</td>
</tr>
<tr>
<td></td>
<td>Proposed w/ n=5 correction bits</td>
<td>0.008</td>
<td>0.750</td>
<td>-0.734</td>
<td>0.094</td>
</tr>
</tbody>
</table>

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

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table 5.2: Error Statistics for Different Fixed-Width Multipliers

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

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 $n$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
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 \( C_d \):

\[
A_l X_L = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (a_i x_j) \cdot 2^{i+j} \approx C_d = \sum_{k=1}^{d} a_n \cdot x_{n-k} 2^{2n-k}.
\]

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

\[
e(n,d) = \frac{1}{2^{2n}} (C_d - A_l X_L) = \sum_{k=1}^{d} a_n \cdot x_{n-k} 2^{2k} - \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (a_i x_j) \cdot 2^{i+j-2n}.
\]

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:

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

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 \( \pm 0.5 \) 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 $2^n = 8, 16$ 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 $2^n = 8, 16$ 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>
<thead>
<tr>
<th>Correction Method</th>
<th>Complexity (# of gates)</th>
<th>Complexity Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (With $A_LX_i$)</td>
<td>596</td>
<td>—</td>
</tr>
<tr>
<td>Removal of $A_LX_i$</td>
<td>452</td>
<td>24.16</td>
</tr>
<tr>
<td>Proposal #1</td>
<td>496</td>
<td>16.78</td>
</tr>
<tr>
<td>Proposal #2</td>
<td>584</td>
<td>2.01</td>
</tr>
<tr>
<td>Proposal #3</td>
<td>461</td>
<td>22.65</td>
</tr>
<tr>
<td>Proposal #4 w/ 1 correction bit</td>
<td>465</td>
<td>21.98</td>
</tr>
<tr>
<td>Proposal #4 w/ $n = 4$ correction bits</td>
<td>500</td>
<td>16.11</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$2n$</th>
<th>Original</th>
<th>Proposal #1</th>
<th>Proposal #2</th>
<th>Proposal #3</th>
<th>Proposal #4 w/ $n$ correction bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>3196</td>
<td>2600</td>
<td>18.65</td>
<td>2884</td>
<td>9.76</td>
</tr>
<tr>
<td>32</td>
<td>12956</td>
<td>10120</td>
<td>21.89</td>
<td>10692</td>
<td>17.47</td>
</tr>
<tr>
<td>64</td>
<td>52444</td>
<td>40136</td>
<td>23.47</td>
<td>41284</td>
<td>21.28</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Truncation Scheme</th>
<th>Error Statistics Performance Score</th>
<th>Complexity Savings Performance Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposal #1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Proposal #2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Proposal #3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Proposal #4 /w 1 correction bit</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Proposal #4 /w n correction bits</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

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)"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
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>
<thead>
<tr>
<th>(2^n)</th>
<th>Correction Method</th>
<th>Delay (ns)</th>
<th>Dynamic Power (mW)</th>
<th>PDP (mW/MHz)</th>
<th>Complexity (LEs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>4.245</td>
<td>—</td>
<td>375.76</td>
<td>—</td>
<td>1.595</td>
</tr>
<tr>
<td>Remove (A_XL)</td>
<td>4.234</td>
<td>0.259</td>
<td>323.55</td>
<td>13.89</td>
<td>1.370</td>
</tr>
<tr>
<td>Proposal #1</td>
<td>4.388</td>
<td>-3.369</td>
<td>338.23</td>
<td>9.98</td>
<td>1.484</td>
</tr>
<tr>
<td>Proposal #3</td>
<td>4.256</td>
<td>0.259</td>
<td>332.44</td>
<td>11.53</td>
<td>1.415</td>
</tr>
<tr>
<td>Proposal #4 with (n = 8) corr. bits</td>
<td>4.258</td>
<td>0.306</td>
<td>336.65</td>
<td>10.41</td>
<td>1.433</td>
</tr>
<tr>
<td>Original</td>
<td>5.116</td>
<td>—</td>
<td>529.79</td>
<td>—</td>
<td>2.710</td>
</tr>
<tr>
<td>Remove (A_XL)</td>
<td>4.686</td>
<td>8.405</td>
<td>432.27</td>
<td>17.31</td>
<td>2.026</td>
</tr>
<tr>
<td>Proposal #1</td>
<td>5.139</td>
<td>-0.450</td>
<td>470.55</td>
<td>10.57</td>
<td>2.418</td>
</tr>
<tr>
<td>Proposal #2</td>
<td>5.389</td>
<td>-5.336</td>
<td>472.86</td>
<td>10.11</td>
<td>2.548</td>
</tr>
<tr>
<td>Proposal #3</td>
<td>4.750</td>
<td>7.154</td>
<td>466.16</td>
<td>12.01</td>
<td>2.214</td>
</tr>
<tr>
<td>Proposal #4 with (n = 16) corr. bits</td>
<td>4.770</td>
<td>6.763</td>
<td>470.36</td>
<td>11.60</td>
<td>2.244</td>
</tr>
</tbody>
</table>
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 \((2^k \times n)\)-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:

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

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

where \(A_i^{(i)}\) and \(X_i^{(i)}\), \(i = 0, 1\), 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:

\[
Y = A \cdot X
= A_1^{(1)} X_1^{(1)} \cdot 2^{2n} + (A_1^{(1)} X_0^{(1)} + A_0^{(1)} X_1^{(1)}) \cdot 2^{2n} + A_0^{(1)} X_0^{(1)}
= \left( A_3^{(2)} X_3^{(2)} \cdot 2^{2n} + (A_2^{(2)} X_2^{(2)} + A_3^{(2)} X_3^{(2)}) \cdot 2^{2n} + A_2^{(2)} X_2^{(2)} \right) \cdot 2^{4n}
+ \left( A_3^{(2)} X_2^{(2)} \cdot 2^{2n} + (A_2^{(2)} X_1^{(2)} + A_3^{(2)} X_2^{(2)}) \cdot 2^{2n} + A_2^{(2)} X_1^{(2)} \right) \cdot 2^{2n}
+ \left( A_2^{(2)} X_1^{(2)} \cdot 2^{2n} + (A_1^{(2)} X_0^{(2)} + A_2^{(2)} X_1^{(2)}) \cdot 2^{2n} + A_1^{(2)} X_0^{(2)} \right) \cdot 2^{2n}
+ \left( A_1^{(2)} X_0^{(2)} \cdot 2^{2n} + (A_0^{(2)} X_0^{(2)}) \cdot 2^{2n} + A_0^{(2)} X_0^{(2)} \right).
\]

### 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:
\[ Y = A \cdot X \]
\[ = A^{(1)}_0 X^{(1)}_0 \cdot 2^{4n} + (A^{(1)}_1 X^{(1)}_1 + A^{(1)}_2 X^{(1)}_2) \cdot 2^{2n} + A^{(1)}_3 X^{(1)}_3 \]
\[ = A^{(1)}_0 X^{(1)}_0 \cdot 2^{4n} + (A^{(2)}_1 X^{(2)}_1 \cdot 2^{2n} + (A^{(2)}_2 X^{(2)}_2 + A^{(2)}_3 X^{(2)}_3) \cdot 2^n + A^{(2)}_4 X^{(2)}_4) \cdot 2^{3n} \]
\[ + A^{(2)}_5 X^{(2)}_5 \cdot 2^{2n} + A^{(2)}_6 X^{(2)}_6 \cdot 2^{2n} + A^{(1)}_3 X^{(1)}_3 \]

Clearly, the last three terms in bold, \( A^{(2)}_2 X^{(2)}_2 \cdot 2^{2n}, A^{(2)}_3 X^{(2)}_3 \cdot 2^{2n}, \) and \( A^{(1)}_3 X^{(1)}_3, \) 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.

![Graphical Representation of Truncation in a Two-Level Recursive Multiplier](image)

**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>
<thead>
<tr>
<th>4n</th>
<th>Number of Correction Bits</th>
<th>$E_{avg}$</th>
<th>$E_{max}$</th>
<th>$E_{max}$</th>
<th>$\sigma_E^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>-0.266</td>
<td>0.500</td>
<td>-1.313</td>
<td>0.165</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.086</td>
<td>0.938</td>
<td>-0.668</td>
<td>0.137</td>
</tr>
<tr>
<td></td>
<td>Max.</td>
<td>0.072</td>
<td>0.938</td>
<td>-0.625</td>
<td>0.140</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>-0.501</td>
<td>0.500</td>
<td>-2.504</td>
<td>0.208</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-0.128</td>
<td>1.109</td>
<td>-1.125</td>
<td>0.136</td>
</tr>
<tr>
<td></td>
<td>Max.</td>
<td>0.094</td>
<td>1.109</td>
<td>-0.961</td>
<td>0.122</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0.017</td>
<td>0.500</td>
<td>-3.250</td>
<td>0.131</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-0.008</td>
<td>1.220</td>
<td>-1.814</td>
<td>0.128</td>
</tr>
<tr>
<td></td>
<td>Max.</td>
<td>0.007</td>
<td>1.220</td>
<td>-1.115</td>
<td>0.116</td>
</tr>
</tbody>
</table>

For the truncated component, $A^{(i)}_0X^{(i)} = \sum_{i=0}^{2n-1} \sum_{j=0}^{2n-1} a_{x_i} \cdot 2^{i+j}$, 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 $A^{(i)}_0X^{(i)}$. The normalized error function $e(d, 2n)$ for the error due to replacing $A^{(i)}_0X^{(i)}$ by $C(d, 2n)$ can be defined as follows:

$$e(d, 2n) = \frac{1}{2^{4n}}(C(d, 2n) - A^{(i)}_0X^{(i)}) = \sum_{k=1}^{d} a_{2n-k} x_{2n-k} \cdot 2^{-k} - \sum_{i=0}^{2n-1} \sum_{j=0}^{2n-1} a_{x_i} \cdot 2^{i+j-4n},$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
For the other two truncated terms (smaller multipliers), their error correction terms and error functions are given by:

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

and

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

respectively.

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

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:

\[ \sum_{i=1}^{k} 2^{i-1} e(d, 2^{k-i} n) + 0.5 \leq \frac{1}{4}(2^k + 1). \]

Maximum post rounding error, \( \pm 0.5 \), 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:

\[
50\left(1 - \frac{1}{2^k}\right).
\]

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>
<thead>
<tr>
<th>Levels of Recursion (( k ))</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. Pos. Error</td>
<td>0.75</td>
<td>1.25</td>
<td>2.25</td>
<td>4.25</td>
<td>8.25</td>
<td>( \frac{1}{4}(2^k +1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Levels of Recursion (( k ))</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approx. Complexity Savings (%)</td>
<td>25.0</td>
<td>37.5</td>
<td>43.8</td>
<td>46.9</td>
<td>48.4</td>
<td>( 50(1 - \frac{1}{2^k}) )</td>
</tr>
</tbody>
</table>
Figure 7.2: Graphical Representation of Complexity Savings for \(k = 1, 2, \text{ and } 3\)

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


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();

int main()
{
  float B,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,AE0,AE1,AE2,AE3;
  float PCI0,PCI1,PCI3,DI,D2,D3;
  int CI0,CI1,CI2,CI3,CI10,CI11;
  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;
    AE0=0;AE1=0;AE2=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==CI0) CI1=CI1-1;
```
AL1 = AL / (B / 2);
XL1 = XL / (B / 2);

C2 = AH * XH + (AH * XL + AL * XH) / B + (AH * XL) / B + 0.5;
D2 = C2 - C110;
if (D2 == 0.5)
{
if (C110 / 2 * 2 == C110) C12 = C12 - 1;
}

C12 = AH * XH + (AH * XL + AL * XH) / B + (AH * XL) / B + 0.5 + 0.5;

/*

E0 = C10 - C;
if (E0 > MAE0) MAE0 = E0;
if (E0 < MIE0) MIE0 = E0;
E0 = E0 * E0;
E1 = C11 - C;
if (E1 > MAE1) MAE1 = E1;
if (E1 < MIE1) MIE1 = E1;
E1 = E1 * E1;
E2 = C12 - C;
if (E2 > MAE2) MAE2 = E2;
if (E2 < MIE2) MIE2 = E2;
EE2 = E2 * E2;
AE0 = AE0 + E0;
AE0 = AE0 * EE0;
AE1 = AE1 + E1;
AE1 = AE1 * EE1;
AE2 = AE2 + E2;
AE2 = AE2 * EE2;

AE0 = AE0 / B / B / B / B;
AE0 = AE0 / B / B / B / B;
AE1 = AE1 / B / B / B / B;
AE1 = AE1 / B / B / B / B;
AE2 = AE2 / B / B / B / B;
AE2 = AE2 / B / B / B / B;

printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var=%f\n", AE0, MAE0, MIE0, AE0 - AE0 * AE0);
printf("(True Round) AE1=%f, MaxE=%f, MinE=%f, Var=%f\n", AE1, MAE1, MIE1, AE1 - AE1 * AE1);
printf("(Trunc. Schm) AE2=%f, MaxE=%f, MinE=%f, Var=%f\n", AE2, MAE2, MIE2, AE2 - AE2 * AE2);
*/

#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,AEEO,AEE1,AEE2,AEE3;
float FC10,FC11,FC12,DI,D2,D3;
int CI0,CI1,CI2,CI3,CI10,CI11;
int n,i,j,k,ii,jj,XX;

/*
Case 1:
0 <= XH,XL,AH,AL <= 2^n-1
Case 2:
2^(n-1) <= XH,AH <= 2^n-1
0 <= XL,AL <= 2^n-1
*/

for (n=8;n<9;n++)
{
printf("-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\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;
}
AL=AL/(B/2);
XL=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-CI10;
if (D2==0.5)
{
if (CI10/2*2==CI10) 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*EO;
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;
A0=AE0+EO;
AE0=AE0+EO0;
AE1=AE1+E1;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
AEE1 = AEE1 + EE1;
AE2 = AE2 + E2;
AEE2 = AEE2 + EE2;

AE0 = AE0 / B / B / B / B;
AE0 = AE0 / B / B / B / B;
AE1 = AE1 / B / B / B / B;
AE1 = AE1 / B / B / B / B;
AE2 = AE2 / B / B / B / B;
AE2 = AE2 / B / B / B / B;

printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var0=%f\n", AE0, MAEO, MIEO, AEEO - AE0 * AE0);
printf("(True Bound) 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
******************************************************************************

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

int test();
int test1();

main (){

float B, C1, C2, C3, C4, C5, C22, C33, C55;
float EO, E1, E2, E3, AE0, AE1, AE2, AE3;
float MAEO, MAE1, MAE2, MAE3;
float MIEO, MIE1, MIE2, MIE3;
float EEO, EE1, EE2, EE3, AEEO, AEE1, AEE2, AEE3;
float FCIO, FC11, FC13, DL, D2, D3;
int CI0, CI1, CI2, CI3, CI01, CI11;
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;
AE0 = 0; AE1 = 0; AE2 = 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-C10;
if(D1==0.5)
{
    if(C10/2*2==C10) C11=C11-1;
}
AL1=AL/(B/2);
XL1=XL/(B/2);
C2=AH*XH+(AH*XL+AL*XL)/B+AL1*XL1;

CI2=AH*XH+(AH*XL+AL*XL)/B+AL1*XL1+0.5;
D2=C2-C110;
if(D2==0.5)
{
    if(C110/2*2==C110) C12=C12-1;
}
E0=C10-C;
if(E0>MAE0) MAE0=E0;
if(E0<MIE0) MIE0=E0;
EE0=E0*E0;
E1=C11-C;
if(E1>MAE1) MAE1=E1;
if(E1<MIE1) MIE1=E1;
EE1=E1*E1;
E2=C12-C;
if(E2>MAE2) MAE2=E2;
if(E2<MIE2) MIE2=E2;
EE2=E2*E2;

AE0=AE0+E0;
AE0=AE0+E0;
AE1=AE1+E1;
AEQ=AEQ+E1;
AE2=AE2+E2;
AE2=AE2+E2;
}
AE0=AE0/B/B/B/B;
AE0=AE0/B/B/B/B;
AE1=AE1/B/B/B/B;
AE1=AE1/B/B/B/B;
AE2=AE2/B/B/B/B;
AE2=AE2/B/B/B/B;
printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var0=%f\n",AE0,MAE0,MIE0,AE0-MAE0*AE0);
printf("(True Round) AE1=%f, MaxE=%f, MinE=%f, Var1=%f\n",AE1,MAE1,MIE1,AE1-MAE1*AE1);
printf("(Trunc.Schm) AE2=%f, MaxE=%f, MinE=%f, Var2=%f\n",AE2,MAE2,MIE2,AE2-MAE2*AE2);
int XH, XL, AH, AL, XL1, AL1, XL2, AL2, XL3, AL3, XL4, AL4, XL5, AL5, XL6, AL6, XL7, AL7, XL8, AL8;

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 FCIO, FC11, FC13, D1, D2, D3;
int CI0, CI1, CI2, CI3, CI10, CI11;
int n, i, j, k, ii, jj, kk;

/*
Case 1:
0 <= XH, XL, AH, AL <= 2^n-1
Case 2:
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;
    AE0 = 0; AE1 = 0; AE2 = 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 = CI0 - 0.5;
                    if (D1 = 0.5)
                    {
                        if (CI0 / 2 * 2 == CI0) CI1 = CI1 - 1;
                    }
                    AL1 = AL / (B / 2);
                    XL1 = XL / (B / 2);
                }
            }
        }
    }
}
A L 2 = A L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 4 );
X L 2 = X L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 4 );

A L 3 = A L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 8 ) - A L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 8 );
X L 3 = X L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 8 ) - X L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 8 );

A L 4 = A L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 16 ) - A L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 16 ) -
A L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 16 );
X L 4 = X L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 16 ) - X L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 16 ) -
X L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 16 );

A L 5 = A L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 32 ) - A L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 32 ) -
A L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 32 ) - A L 4 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 32 );
X L 5 = X L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 32 ) - X L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 32 ) -
X L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 32 ) - X L 4 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 32 );

A L 6 = A L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) - A L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) -
A L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) - A L 4 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) -
A L 5 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) - A L 6 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 );
X L 6 = X L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) - X L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) -
X L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) - X L 4 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) -
X L 5 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 ) - X L 6 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 64 );

A L 7 = A L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) - A L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) -
A L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) - A L 4 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) -
A L 5 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) - A L 6 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 );
X L 7 = X L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) - X L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) -
X L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) - X L 4 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) -
X L 5 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 ) - X L 6 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 128 );

A L 8 = A L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) - A L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) -
A L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) - A L 4 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) -
A L 5 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) - A L 6 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 );
X L 8 = X L 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) - X L 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) -
X L 3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) - X L 4 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) -
X L 5 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 ) - X L 6 * 2 * 2 * 2 * 2 * 2 * 2 * 2 / ( B / 256 );

C 2 = A H * X H + ( A H * X L + A L * X H ) / B + ( A H * X L + A L * X H ) / 2 / B / B +
( A L 1 * X L 1 / 2.0 ) + ( A L 2 * X L 2 / 2.0 / 2.0 ) + ( A L 3 * X L 3 / 2.0 / 2.0 / 2.0 ) + ( A L 4 * X L 4 / 2.0 / 2.0 / 2.0 / 2.0 ) + ( A L 5 * X L 5 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 ) + ( A L 6 * X L 6 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 ) + ( A L 7 * X L 7 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 ) + ( A L 8 * X L 8 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 / 2.0 ) + 0.5;

D 2 = C 2 - C I I 0 ;
if ( D 2 == 0.5 ){
  if ( C I I 0 / 2.0 == C I I 0 ) C I I 0 = C I I 0 - 1 ;
}

E 0 = C I 0 - C;
if ( E 0 > M A E 0 ) M A E 0 = E 0 ;
if ( E 0 < M I E 0 ) M I E 0 = E 0 ;
E E 0 = E 0 * E 0 ;
E 1 = C I 1 - C;
if ( E 1 > M A E 1 ) M A E 1 = E 1 ;
if ( E 1 < M I E 1 ) M I E 1 = E 1 ;
E E 1 = E 1 * E 1 ;
E 2 = C I 2 - C;
if ( E 2 > M A E 2 ) M A E 2 = E 2 ;
if ( E 2 < M I E 2 ) M I E 2 = E 2 ;
E E 2 = E 2 * E 2 ;
A E 0 = A E 0 + E 0 ;
A E 0 5 = A E 0 5 + E E 0 ;
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);

MaxE=%f, MinE=%f, Varl = % f

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();

int main ()  {
  int XHH,XHL,XLH,XLL,AHH,ALH,ALL;
  int ALH1,XLH1,ALL1,XHL1;
  int ALI,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,AEO,AE1,AE2,AE3;
  float MAE0,MAE1,MAE2,MAE3;
  float MIE0,MIE1,MIE2,MIE3;
  float EEO,E11,EE2,EE3,AEE0,AEE1,AEE2,AEE3;
  float EEO,E11,EE2,EE3,AEE0,AEE1,AEE2,AEE3;
  int CI0,C1I,C12,C13,C1I0,C1II;
  int n,i,j,k,ii,jj,kk;

  /*
  Case1:  
  0 <= XH,XL,AM,AL <= 2^n-1
  Case2:  
  2^{n-1} <= XH,AM <= 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++)
\[2 = 2 \times 2;\]

\[A E 0 = 0; A E 1 = 0; A E 2 = 0;\]
\[A E 0 = 0; A E 1 = 0; A E 2 = 0;\]
\[M A E 0 = 0; M A E 1 = 0; M A E 2 = 0;\]
\[M I E 0 = 0; M I E 1 = 0; M I E 2 = 0;\]

\[
\text{for}(X H H = 0; X H H < B; X H H + + )
\]
\[
\text{for}(X H L = 0; X H L < B; X H L + + )
\]
\[
\text{for}(X L H = 0; X L H < B; X L H + + )
\]
\[
\text{for}(X L L = 0; X L L < B; X L L + + )
\]
\[
\text{for}(A H H = 0; A H H < B; A H H + + )
\]
\[
\text{for}(A H L = 0; A H L < B; A H L + + )
\]
\[
\text{for}(A L H = 0; A L H < B; A L H + + )
\]
\[
\text{for}(A L L = 0; A L L < B; A L L + + )
\]

\[
C = (A H H \times X H H \times B \times B + (A H H \times X H L + A H L \times X H H) \times B + A H L \times X H L) + (A H H \times X L H \times B \times B + (A H H \times X L L + A H L \times X L H) \times B + A H L \times X L L) / B^2 + (A L H \times X H H \times B \times B + (A L H \times X H L + A L L \times X H H) \times B + A L L \times X H L) / B^2 + (A L H \times X L H \times B \times B + (A L H \times X L L + A L L \times X L H) \times B + A L L \times X L L) / B^2 / B^2;
\]

\[C I 0 = C;\]
\[C I 1 = (A H H \times X H H \times B \times B + (A H H \times X H L + A H L \times X H H) \times B + A H L \times X H L) + (A H H \times X L H \times B \times B + (A H H \times X L L + A H L \times X L H) \times B + A H L \times X L L) / B^2 + (A L H \times X H H \times B \times B + (A L H \times X H L + A L L \times X H H) \times B + A L L \times X H L) / B^2 + (A L H \times X L H \times B \times B + (A L H \times X L L + A L L \times X L H) \times B + A L L \times X L L) / B^2 / B^2 + 0.5;\]
\[D 1 = C - C I 0;\]
\[i f(D 1 == 0.5)\]
\[
\text{if}(C I 0 / 2 \times 2 == C I 0) C I 1 = C I 1 - 1;\]
\]

\/* \text{ALH1=ALH/(B/2)}; \text{XHL1=XHL/(B/2)}; \*/

\[
C I I 0 = C I I 0;\]
\[
C I 2 = (A H H \times X H H \times B \times B + (A H H \times X H L + A H L \times X H H) \times B + A H L \times X H L) + (A H H \times X L H \times B \times B + (A H H \times X L L + A H L \times X L H) \times B + A H L \times X L L) / B^2 + (A L H \times X H H \times B \times B + (A L H \times X H L + A L L \times X H H) \times B + A L L \times X H L) / B^2 + (A L H \times X L H \times B \times B + (A L H \times X L L + A L L \times X L H) \times B + A L L \times X L L) / B^2 / B^2 + 0.5;\]
\[D 2 = C 2 - C I I 0;\]
\[i f(D 2 == 0.5)\]
\[
\text{if}(C I I 0 / 2 \times 2 == C I I 0) C I 2 = C I 2 - 1;\]
\]

\[E 0 = C I 0 - C;\]
\[i f(E 0 > M A E 0) M A E 0 = E 0;\]
\[i f(E 0 < M I E 0) M I E 0 = E 0;\]
\[E E 0 = E 0 ; E 0 ;\]
\[E 1 = C I 1 - C;\]
\[i f(E 1 > M A E 1) M A E 1 = E 1;\]
\[i f(E 1 < M I E 1) M I E 1 = E 1;\]
\[E E 1 = E 1 ; E 1 ;\]
\[E 2 = C I 2 - C;\]
\[i f(E 2 > M A E 2) M A E 2 = E 2;\]
\[i f(E 2 < M I E 2) M I E 2 = E 2;\]
\begin{verbatim}
EE2=E2*E2;
AE0=AO+EO;
AEO=AEO+EO;
AE1=AE1+EO;
AEEO=AEEO+EEO;
AE1=AE1+E1;
AE2=AE2+E2;
AEE2=AEE2+E2;
AE0=AE0/B2/B2/B2/B2;
AEE0=AEO/B2/B2/B2/B2;
printf("(Truncation) AE0=%f, MaxE=%f, MinE=%f, Var0=%f\n", AEO, MIE0, MIE0, AEO-AEO*AE0);
printf("(True Round) AE1=%f, MaxE=%f, MinE=%f, Var1=%f\n", AE1, MIE1, MIE1, AE1-AE1*AE1);
printf("(Trunc.Schm) AE2=%f, MaxE=%f, MinE=%f, Var2=%f\n", AE2, MIE2, MIE2, AE2-AE2*AE2);
\end{verbatim}
/* VERILOG HARDWARE DESCRIPTION LANGUAGE */

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,
                AL11, XL11, AL12, XL12, AL13, XL13, AL14, XL14,
                AL15, XL15, AL16, XL16;

    wire [31:0] SUBMULT1;
    wire [31:0] SUBMULT2, SUBMULT3, SUBMULT4;
    reg [15:0] SHIFTSUBMULT1;
    reg [31:0] SHIFTSUBMULT2;
    reg [47:0] 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, XL3, AL4, XL4, AL5, XL5, AL6, XL6, AL7, XL7, AL8, XL8);

    //ARRAYMULTIPLIER16 base multiplier
    (SUBMULT1, AL, XL); // remove ALX
    //SUBMULT[15:0] = 16'x0;
    ARRAYMULTIPLIER16 (SUBMULT2, AH, XL);
    ARRAYMULTIPLIER16 (SUBMULT3, AL, XH);
    ARRAYMULTIPLIER16 (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];

        SUBMULT1 <= A*B;
        //OUT2 <= A*B;
    end

endmodule

//produced with permission of the copyright owner. Further reproduction prohibited without permission.
//ANDGATE and temp (temp, SHIFTSUBMULT1[2], SHIFTSUBMULT3[4]);

endmodule //RECURSIVEMULTIPLIER

/* module HIGHLOWBITS (CK, A, B, AL, HL, A0, A1, A2, A3, A4, A5, A6, A7, A8, AL0, AL1, AL2, AL3, AL4, AL5, AL6, AL7, AL8, HL0, HL1, HL2, HL3, HL4, HL5, HL6, HL7, HL8);

input CK;
input [15:0] A, B;
output [7:0] AH, AL, XH, XL;
output [0:0] AL1, AL2, AL3, AL4, AL5, AL6, AL7, AL8, HL1, HL2, HL3, HL4, HL5, HL6, HL7, HL8;

reg [7:0] AH, AL, XH, XL;
reg [0:0] AL1, AL2, AL3, AL4, AL5, AL6, AL7, AL8, HL1, HL2, HL3, HL4, HL5, HL6, HL7, HL8;

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, WCOUT04, WCOUT05, WCOUT06, WCOUT07,
WCOUT10, WCOUT11, WCOUT12, WCOUT13, WCOUT14, WCOUT15, WCOUT16, WCOUT17,
WCOUT20, WCOUT21, WCOUT22, WCOUT23, WCOUT24, WCOUT25, WCOUT26, WCOUT27,
WCOUT30, WCOUT31, WCOUT32, WCOUT33, WCOUT34, WCOUT35, WCOUT36, WCOUT37,
WCOUT40, WCOUT41, WCOUT42, WCOUT43, WCOUT44, WCOUT45, WCOUT46, WCOUT47,
WCOUT50, WCOUT51, WCOUT52, WCOUT53, WCOUT54, WCOUT55, WCOUT56, WCOUT57,
WCOUT60, WCOUT61, WCOUT62, WCOUT63, WCOUT64, WCOUT65, WCOUT66, WCOUT67,
WCOUT70, WCOUT71, WCOUT72, WCOUT73, WCOUT74, WCOUT75, WCOUT76, WCOUT77;

wire WCOUTF01, WCOUTF02, WCOUTF03, WCOUTF04, WCOUTF05, WCOUTF06;
wire //assign OUT2 = A*B;

//ANDGATE and A0 (X, A[0], B[0]);

//ARRAY MULTIPLIER ARCHITECTURE

//ROW 0
ARRAYCELL arraycell00 (A[0], B[0], l'bo, l'bo, WCOUT00, OUT[0]);
ARRAYCELL arraycell101 (A[1], B[0], l'bo, l'bo, WCOUT01, WSOUT01);
ARRAYCELL arraycell102 (A[2], B[0], l'bo, l'bo, WCOUT02, WSOUT02);
ARRAYCELL arraycell103 (A[3], B[0], l'bo, l'bo, WCOUT03, WSOUT03);
ARRAYCELL arraycell104 (A[4], B[0], l'bo, l'bo, WCOUT04, WSOUT04);
ARRAYCELL arraycell105 (A[5], B[0], l'bo, l'bo, WCOUT05, WSOUT05);
ARRAYCELL arraycell106 (A[6], B[0], l'bo, l'bo, WCOUT06, WSOUT06);
ARRAYCELL arraycell107 (A[7], B[0], l'bo, l'bo, WCOUT07, WSOUT07);

//ROW 1
ARRAYCELL arraycell110 (A[0], B[1], WCOUT00, WSOUT01, WCOUT10, OUT[1]);
ARRAYCELL arraycell111 (A[1], B[1], WCOUT01, WCOUT02, WCOUT11, WSOUT11);
module ARRAYMULTIPLIER16 (OUT, A, B);

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

wire WCOUT0000,
WCOUT0001,
WCOUT0002,
WCOUT0003,
WCOUT0004,
WCOUT0005,
WCOUT0006,
WCOUT0007,
WCOUT0008,
WCOUT0009,
WCOUT0010,
WCOUT0011,
WCOUT0012,
WCOUT0013,
WCOUT0014,
WCOUT0015,
WCOUT0016,
WCOUT0017,
WCOUT0018,
WCOUT0019,
WCOUT0020,
WCOUT0021,
WCOUT0022,
WCOUT0023,
WCOUT0024,
WCOUT0025,
WCOUT0026,
WCOUT0027,
WCOUT0028,
WCOUT0029,
WCOUT0030,
WCOUT0031,
WCOUT0032,
WCOUT0033,
WCOUT0034,
WCOUT0035,
WCOUT0036,
WCOUT0037,
WCOUT0038,
WCOUT0039,
WCOUT0040,
WCOUT0041,
WCOUT0042,
WCOUT0043,
WCOUT0044,
WCOUT0045,
WCOUT0046,
WCOUT0047,
WCOUT0048,
WCOUT0049,
WCOUT0050,
WCOUT0051,
WCOUT0052,
WCOUT0053,
WCOUT0054,
WCOUT0055,
WCOUT0056,
WCOUT0057,
WCOUT0058,
WCOUT0059,
WCOUT0060,
WCOUT0061,
WCOUT0062,
WCOUT0063,
WCOUT0064,
WCOUT0065,
WCOUT0066,
WCOUT0067,
WCOUT0068,
WCOUT0069,
WCOUT0070,
WCOUT0071,
WCOUT0072,
WCOUT0073,
WCOUT0074,
WCOUT0075,
WCOUT0076,
WCOUT0077,
WCOUT0078,
WCOUT0079,
WCOUT0080,
WCOUT0081,
WCOUT0082,
WCOUT0083,
WCOUT0084,
WCOUT0085,
WCOUT0086,
WCOUT0087,
WCOUT0088,
WCOUT0089,
WCOUT0090,
WCOUT0091,
WCOUT0092,
WCOUT0093,
WCOUT0094,
WCOUT0095,
WCOUT0096,
WCOUT0097,
WCOUT0098,
WCOUT0099,
WCOUT0100,
WCOUT0101,
WCOUT0102,
WCOUT0103,
WCOUT0104,
WCOUT0105,
WCOUT0106,
WCOUT0107,
WCOUT0108,
WCOUT0109,
WCOUT0110,
WCOUT0111,
WCOUT0112,
WCOUT0113,
WCOUT0114,
WCOUT0115,
WCOUT0116,
WCOUT0117,
WCOUT0118,
WCOUT0119,
WCOUT0120,
WCOUT0121,
WCOUT0122,
WCOUT0123,
WCOUT0124,
WCOUT0125,
WCOUT0126,
WCOUT0127,
WCOUT0128,
WCOUT0129,
WCOUT0130,
WCOUT0131,
WCOUT0132,
WCOUT0133,
WCOUT0134,
WCOUT0135,
WCOUT0136,
WCOUT0137,
WCOUT0138,
WCOUT0139,
WCOUT0140,
WCOUT0141,
WCOUT0142,
WCOUT0143,
WCOUT0144,
WCOUT0145,
WCOUT0146,
WCOUT0147,
WCOUT0148,
WCOUT0149,
WCOUT0150,
WCOUT0151,
WCOUT0152,
WCOUT0153,
WCOUT0154,
WCOUT0155,
WCOUT0156,
WCOUT0157,
WCOUT0158,
WCOUT0159,
WCOUT0160,
WCOUT0161,
WCOUT0162,
WCOUT0163,
WCOUT0164,
WCOUT0165,
WCOUT0166,
WCOUT0167,
WCOUT0168,
WCOUT0169,
WCOUT0170,
WCOUT0171,
WCOUT0172,
WCOUT0173,
WCOUT0174,
WCOUT0175,
WCOUT0176,
WCOUT0177,
WCOUT0178,
WCOUT0179,
WCOUT0180,
WCOUT0181,
WCOUT0182,
WCOUT0183,
WCOUT0184,
WCOUT0185,
WCOUT0186,
WCOUT0187,
WCOUT0188,
WCOUT0189,
WCOUT0190,
WCOUT0191,
WCOUT0192,
WCOUT0193,
WCOUT0194,
WCOUT0195,
WCOUT0196,
WCOUT0197,
WCOUT0198,
WCOUT0199,
WCOUT0200,
WCOUT0201,
WCOUT0202,
WCOUT0203,
WCOUT0204,
WCOUT0205,
WCOUT0206,
WCOUT0207,
WCOUT0208,
WCOUT0209,
WCOUT0210,
WCOUT0211,
WCOUT0212,
WCOUT0213,
WCOUT0214,
WCOUT0215,
WCOUT0216,
WCOUT0217,
WCOUT0218,
WCOUT0219,
WCOUT0220,
WCOUT0221,
WCOUT0222,
WCOUT0223,
WCOUT0224,
WCOUT0225,
WCOUT0226,
WCOUT0227,
WCOUT0228,
WCOUT0229,
WCOUT0230,
WCOUT0231,
WCOUT0232,
WCOUT0233,
WCOUT0234,
WCOUT0235,
WCOUT0236,
WCOUT0237,
WCOUT0238,
WCOUT0239,
WCOUT0240,
WCOUT0241,
WCOUT0242,
WCOUT0243,
WCOUT0244,
WCOUT0245,
WCOUT0246,
WCOUT0247,
WCOUT0248,
WCOUT0249,
WCOUT0250,
WCOUT0251,
WCOUT0252,
WCOUT0253,
WCOUT0254,
WCOUT0255,
WCOUT0256,
WCOUT0257,
WCOUT0258,
WCOUT0259,
WCOUT0260,
WCOUT0261,
WCOUT0262,
WCOUT0263,
WCOUT0264,
WCOUT0265,
WCOUT0266,
WCOUT0267,
WCOUT0268,
WCOUT0269,
WCOUT0270,
WCOUT0271,
WCOUT0272,
WCOUT0273,
WCOUT0274,
WCOUT0275,
WCOUT0276,
WCOUT0277,
WCOUT0278,
WCOUT0279,
WCOUT0280,
WCOUT0281,
WCOUT0282,
WCOUT0283,
WCOUT0284,
WCOUT0285,
WCOUT0286,
WCOUT0287,
WCOUT0288,
WCOUT0289,
WCOUT0290,
WCOUT0291,
WCOUT0292,
WCOUT0293,
WCOUT0294,
WCOUT0295,
WCOUT0296,
WCOUT0297,
WCOUT0298,
WCOUT0299,
WCOUT0300,
WCOUT0301,
WCOUT0302,
WCOUT0303,
WCOUT0304,
WCOUT0305,
WCOUT0306,
WCOUT0307,
WCOUT0308,
WCOUT0309,
WCOUT0310,
WCOUT0311,
WCOUT0312,
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WCOUT1202, WCOUT1203, WCOUT1204, WCOUT1205, WCOUT1206, WCOUT1207, WCOUT1208, WCOUT1209, WCOUT1210, WCOUT1211, WCOUT1212, WCOUT1213, WCOUT1214, WCOUT1215, WCOUT1200, WCOUT1201, WCOUT1202, WCOUT1203, WCOUT1204, WCOUT1205, WCOUT1206, WCOUT1207, WCOUT1208, WCOUT1209, WCOUT1210, WCOUT1211, WCOUT1212, WCOUT1213, WCOUT1214, WCOUT1215,</td>
<td>wire</td>
<td>WSOUT0001,</td>
<td>WSOUT0002,</td>
<td>WSOUT0003,</td>
<td>WSOUT0004,</td>
<td>WSOUT0005,</td>
<td>WSOUT0006,</td>
<td>WSOUT0007,</td>
<td>WSOUT0008,</td>
<td>WSOUT0009,</td>
<td>WSOUT0010,</td>
<td>WSOUT0011,</td>
<td>WSOUT0012,</td>
<td>WSOUT0013,</td>
<td>WSOUT0014,</td>
<td>WSOUT0015,</td>
<td>WSOUT0100,</td>
<td>WSOUT0101,</td>
<td>WSOUT0102,</td>
<td>WSOUT0103,</td>
<td>WSOUT0104,</td>
<td>WSOUT0105,</td>
<td>WSOUT0106,</td>
<td>WSOUT0107,</td>
<td>WSOUT0108,</td>
<td>WSOUT0109,</td>
<td>WSOUT0110,</td>
<td>WSOUT0111,</td>
<td>WSOUT0112,</td>
<td>WSOUT0113,</td>
<td>WSOUT0114,</td>
<td>WSOUT0115,</td>
<td>WSOUT0200,</td>
<td>WSOUT0201,</td>
<td>WSOUT0202,</td>
<td>WSOUT0203,</td>
<td>WSOUT0204,</td>
<td>WSOUT0205,</td>
<td>WSOUT0206,</td>
<td>WSOUT0207,</td>
<td>WSOUT0208,</td>
<td>WSOUT0209,</td>
<td>WSOUT0210,</td>
<td>WSOUT0211,</td>
<td>WSOUT0212,</td>
<td>WSOUT0213,</td>
<td>WSOUT0214,</td>
<td>WSOUT0215,</td>
<td>WSOUT0300,</td>
<td>WSOUT0301,</td>
<td>WSOUT0302,</td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[1], B[0], l'b0, l'b0, WCOUT0001, WSOUT0001)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[2], B[0], l'b0, l'b0, WCOUT0002, WSOUT0002)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[3], B[0], l'b0, l'b0, WCOUT0003, WSOUT0003)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[4], B[0], l'b0, l'b0, WCOUT0004, WSOUT0004)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[5], B[0], l'b0, l'b0, WCOUT0005, WSOUT0005)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[6], B[0], l'b0, l'b0, WCOUT0006, WSOUT0006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[7], B[0], l'b0, l'b0, WCOUT0007, WSOUT0007)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[8], B[0], l'b0, l'b0, WCOUT0008, WSOUT0008)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[9], B[0], l'b0, l'b0, WCOUT0009, WSOUT0009)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[10], B[0], l'b0, l'b0, WCOUT0010, WSOUT0010)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[11], B[0], l'b0, l'b0, WCOUT0011, WSOUT0011)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10012</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[12], B[0], l'b0, l'b0, WCOUT0012, WSOUT0012)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10013</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[13], B[0], l'b0, l'b0, WCOUT0013, WSOUT0013)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[14], B[0], l'b0, l'b0, WCOUT0014, WSOUT0014)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell10015</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A[15], B[0], l'b0, l'b0, WCOUT0015, WSOUT0015)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

//ROW 0
ARRAYCELL arraycell10000
(A[0], B[0], l'b0, l'b0, WCOUT0000, WSOUT0000, OUT[0]);

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

//ROW 1
ARRAYCELL arraycell10000
(A[0], B[1], WCOUT0000, WSOUT0001, WCOUT0010, OUT[1]);
ARRAYCELL arraycell10010
(A[1], B[1], WCOUT0001, WSOUT0002, WCOUT0011, WSOUT0010);
ARRAYCELL arraycell10012
(A[2], B[1], WCOUT0002, WSOUT0003, WCOUT0012, WSOUT0012);
ARRAYCELL arraycell10013
(A[3], B[1], WCOUT0003, WSOUT0004, WCOUT0013, WSOUT0013);
ARRAYCELL arraycell10014
(A[4], B[1], WCOUT0004, WSOUT0005, WCOUT0014, WSOUT0014);
ARRAYCELL arraycell10015
(A[5], B[1], WCOUT0005, WSOUT0006, WCOUT0015, WSOUT0015);
ARRAYCELL arraycell10016
(A[6], B[1], WCOUT0006, WSOUT0007, WCOUT0016, WSOUT0016);
| 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, WSOUT0100, WCOUT0109, WSOUT0109); |
| ARRAYCELL | arraycell0110 (A[10], B[1], WCOUT0100, WSOUT0101, WCOUT0110, WSOUT0110); |
| ARRAYCELL | arraycell0111 (A[11], B[1], WCOUT0101, WSOUT0102, WCOUT0111, WSOUT0111); |
| ARRAYCELL | arraycell0112 (A[12], B[1], WCOUT0102, WSOUT0103, WCOUT0112, WSOUT0112); |
| ARRAYCELL | arraycell0113 (A[13], B[1], WCOUT0103, WSOUT0104, WCOUT0113, WSOUT0113); |
| ARRAYCELL | arraycell0114 (A[14], B[1], WCOUT0104, WSOUT0105, WCOUT0114, WSOUT0114); |
| ARRAYCELL | arraycell0115 (A[15], B[1], WCOUT0105, 1'b0, WSOUT0105, WCOUT0105); |
| //ROW 2 | ARRAYCELL arraycell0200 (A[0], B[2], WCOUT0200, WSOUT0201, WCOUT0200, WSOUT0200); |
| ARRAYCELL | arraycell0201 (A[1], B[2], WCOUT0201, WSOUT0202, WCOUT0201, WSOUT0201); |
| ARRAYCELL | arraycell0202 (A[2], B[2], WCOUT0202, WSOUT0203, WCOUT0202, WSOUT0202); |
| ARRAYCELL | arraycell0203 (A[3], B[2], WCOUT0203, WSOUT0204, WCOUT0203, WSOUT0203); |
| ARRAYCELL | arraycell0204 (A[4], B[2], WCOUT0204, WSOUT0205, WCOUT0204, WSOUT0204); |
| ARRAYCELL | arraycell0205 (A[5], B[2], WCOUT0205, WSOUT0206, WCOUT0205, WSOUT0205); |
| ARRAYCELL | arraycell0206 (A[6], B[2], WCOUT0206, WSOUT0207, WCOUT0206, WSOUT0206); |
| ARRAYCELL | arraycell0207 (A[7], B[2], WCOUT0207, WSOUT0208, WCOUT0207, WSOUT0207); |
| ARRAYCELL | arraycell0208 (A[8], B[2], WCOUT0208, WSOUT0209, WCOUT0208, WSOUT0208); |
| ARRAYCELL | arraycell0209 (A[9], B[2], WCOUT0209, WSOUT0210, WCOUT0209, WSOUT0209); |
| ARRAYCELL | arraycell0210 (A[10], B[2], WCOUT0210, WSOUT0211, WCOUT0210, WSOUT0210); |
| ARRAYCELL | arraycell0211 (A[11], B[2], WCOUT0211, WSOUT0212, WCOUT0211, WSOUT0211); |
| ARRAYCELL | arraycell0212 (A[12], B[2], WCOUT0212, WSOUT0213, WCOUT0212, WSOUT0212); |
| ARRAYCELL | arraycell0213 (A[13], B[2], WCOUT0213, WSOUT0214, WCOUT0213, WSOUT0213); |
| //ROW 3 | ARRAYCELL arraycell0300 (A[0], B[3], WCOUT0300, WSOUT0301, WCOUT0300, WSOUT0300); |
| ARRAYCELL | arraycell0301 (A[1], B[3], WCOUT0301, WSOUT0302, WCOUT0301, WSOUT0301); |
| ARRAYCELL | arraycell0302 (A[2], B[3], WCOUT0302, WSOUT0303, WCOUT0302, WSOUT0302); |
| ARRAYCELL | arraycell0303 (A[3], B[3], WCOUT0303, WSOUT0304, WCOUT0303, WSOUT0303); |
| ARRAYCELL | arraycell0304 (A[4], B[3], WCOUT0304, WSOUT0305, WCOUT0304, WSOUT0304); |
| ARRAYCELL | arraycell0305 (A[5], B[3], WCOUT0305, WSOUT0306, WCOUT0305, WSOUT0305); |
| ARRAYCELL | arraycell0306 (A[6], B[3], WCOUT0306, WSOUT0307, WCOUT0306, WSOUT0306); |
| ARRAYCELL | arraycell0307 (A[7], B[3], WCOUT0307, WSOUT0308, WCOUT0307, WSOUT0307); |
| ARRAYCELL | arraycell0308 (A[8], B[3], WCOUT0308, WSOUT0309, WCOUT0308, WSOUT0308); |
| ARRAYCELL | arraycell0309 (A[9], B[3], WCOUT0309, WSOUT0310, WCOUT0309, WSOUT0309); |
| ARRAYCELL | arraycell0310 (A[10], B[3], WCOUT0310, WSOUT0311, WCOUT0310, WSOUT0310); |
| ARRAYCELL | arraycell0311 (A[11], B[3], WCOUT0311, WSOUT0312, WCOUT0311, WSOUT0311); |
| ARRAYCELL | arraycell0312 (A[12], B[3], WCOUT0312, WSOUT0313, WCOUT0312, WSOUT0312); |
| ARRAYCELL | arraycell0313 (A[13], B[3], WCOUT0313, WSOUT0314, WCOUT0313, WSOUT0314); |
| ARRAYCELL | arraycell0314 (A[14], B[3], WCOUT0314, WSOUT0315, WCOUT0314, WSOUT0314); |
| ARRAYCELL | arraycell0315 (A[15], B[3], WCOUT0315, 1'b0, WSOUT0315, WCOUT0315); |
| //ROW 4 | ARRAYCELL arraycell0400 (A[0], B[4], WCOUT0400, WSOUT0401, WCOUT0400, WSOUT0400); |
| ARRAYCELL | arraycell0400 (A[1], B[4], WCOUT0401, WSOUT0402, WCOUT0401, WSOUT0401); |
| ARRAYCELL | arraycell0401 (A[2], B[4], WCOUT0402, WSOUT0403, WCOUT0402, WSOUT0402); |

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>ARRAYCELL</th>
<th>arraycell0914</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A[14], B[9], WCOUT0814, WSOUT0815, WCOUT0914, WSOUT0914)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell0915</td>
</tr>
<tr>
<td>(A[15], B[9], WCOUT0815, 1'b0, WCOUT0915, WSOUT0915)</td>
<td></td>
</tr>
</tbody>
</table>

// Row 10
<table>
<thead>
<tr>
<th>ARRAYCELL</th>
<th>arraycell1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A[0], B[10], WCOUT0900, WSOUT0901, WCOUT1000, WSOUT1000)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1001</td>
</tr>
<tr>
<td>(A[1], B[10], WCOUT0901, WSOUT0902, WCOUT1001, WSOUT1001)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1002</td>
</tr>
<tr>
<td>(A[2], B[10], WCOUT0902, WSOUT0903, WCOUT1002, WSOUT1002)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1003</td>
</tr>
<tr>
<td>(A[3], B[10], WCOUT0903, WSOUT0904, WCOUT1003, WSOUT1003)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1004</td>
</tr>
<tr>
<td>(A[4], B[10], WCOUT0904, WSOUT0905, WCOUT1004, WSOUT1004)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1005</td>
</tr>
<tr>
<td>(A[5], B[10], WCOUT0905, WSOUT0906, WCOUT1005, WSOUT1005)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1006</td>
</tr>
<tr>
<td>(A[6], B[10], WCOUT0906, WSOUT0907, WCOUT1006, WSOUT1006)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1007</td>
</tr>
<tr>
<td>(A[7], B[10], WCOUT0907, WSOUT0908, WCOUT1007, WSOUT1007)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1008</td>
</tr>
<tr>
<td>(A[8], B[10], WCOUT0908, WSOUT0909, WCOUT1008, WSOUT1008)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1009</td>
</tr>
<tr>
<td>(A[9], B[10], WCOUT0909, WSOUT0910, WCOUT1009, WSOUT1009)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1010</td>
</tr>
<tr>
<td>(A[10], B[10], WCOUT0910, WSOUT0911, WCOUT1010, WSOUT1010)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1011</td>
</tr>
<tr>
<td>(A[11], B[10], WCOUT0911, WSOUT0912, WCOUT1011, WSOUT1011)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1012</td>
</tr>
<tr>
<td>(A[12], B[10], WCOUT0912, WSOUT0913, WCOUT1012, WSOUT1012)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1013</td>
</tr>
<tr>
<td>(A[13], B[10], WCOUT0913, WSOUT0914, WCOUT1013, WSOUT1013)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1014</td>
</tr>
<tr>
<td>(A[14], B[10], WCOUT0914, WSOUT0915, WCOUT1014, WSOUT1014)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1015</td>
</tr>
<tr>
<td>(A[15], B[10], WCOUT0915, 1'b0, WCOUT1015, WSOUT1015)</td>
<td></td>
</tr>
</tbody>
</table>

// Row 11
<table>
<thead>
<tr>
<th>ARRAYCELL</th>
<th>arraycell1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A[0], B[12], WCOUT1000, WSOUT1001, WCOUT1100, WSOUT1100)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1101</td>
</tr>
<tr>
<td>(A[1], B[11], WCOUT1001, WSOUT1002, WCOUT1101, WSOUT1101)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1102</td>
</tr>
<tr>
<td>(A[2], B[11], WCOUT1002, WSOUT1003, WCOUT1102, WSOUT1102)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1103</td>
</tr>
<tr>
<td>(A[3], B[11], WCOUT1003, WSOUT1004, WCOUT1103, WSOUT1103)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARRAYCELL</th>
<th>arraycell1104</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A[4], B[11], WCOUT1004, WSOUT1005, WCOUT1104, WSOUT1104)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1105</td>
</tr>
<tr>
<td>(A[5], B[11], WCOUT1005, WSOUT1006, WCOUT1105, WSOUT1105)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1106</td>
</tr>
<tr>
<td>(A[6], B[11], WCOUT1006, WSOUT1007, WCOUT1106, WSOUT1106)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1107</td>
</tr>
<tr>
<td>(A[7], B[11], WCOUT1007, WSOUT1008, WCOUT1107, WSOUT1107)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1108</td>
</tr>
<tr>
<td>(A[8], B[11], WCOUT1008, WSOUT1009, WCOUT1108, WSOUT1108)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1109</td>
</tr>
<tr>
<td>(A[9], B[11], WCOUT1009, WSOUT1010, WCOUT1109, WSOUT1109)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1110</td>
</tr>
<tr>
<td>(A[10], B[11], WCOUT1010, WSOUT1011, WCOUT1110, WSOUT1110)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1111</td>
</tr>
<tr>
<td>(A[11], B[11], WCOUT1011, WSOUT1012, WCOUT1111, WSOUT1111)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1112</td>
</tr>
<tr>
<td>(A[12], B[11], WCOUT1012, WSOUT1013, WCOUT1112, WSOUT1112)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1113</td>
</tr>
<tr>
<td>(A[13], B[11], WCOUT1013, WSOUT1014, WCOUT1113, WSOUT1113)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1114</td>
</tr>
<tr>
<td>(A[14], B[11], WCOUT1014, WSOUT1015, WCOUT1114, WSOUT1114)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1115</td>
</tr>
<tr>
<td>(A[15], B[11], WCOUT1015, 1'b0, WCOUT1115, WSOUT1115)</td>
<td></td>
</tr>
</tbody>
</table>

// Row 12
<table>
<thead>
<tr>
<th>ARRAYCELL</th>
<th>arraycell1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A[0], B[12], WCOUT1100, WSOUT1101, WCOUT1200, WSOUT1200)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1201</td>
</tr>
<tr>
<td>(A[1], B[12], WCOUT1101, WSOUT1102, WCOUT1201, WSOUT1201)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1202</td>
</tr>
<tr>
<td>(A[2], B[12], WCOUT1102, WSOUT1103, WCOUT1202, WSOUT1202)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1203</td>
</tr>
<tr>
<td>(A[3], B[12], WCOUT1103, WSOUT1104, WCOUT1203, WSOUT1203)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1204</td>
</tr>
<tr>
<td>(A[4], B[12], WCOUT1104, WSOUT1105, WCOUT1204, WSOUT1204)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1205</td>
</tr>
<tr>
<td>(A[5], B[12], WCOUT1105, WSOUT1106, WCOUT1205, WSOUT1205)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1206</td>
</tr>
<tr>
<td>(A[6], B[12], WCOUT1106, WSOUT1107, WCOUT1206, WSOUT1206)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1207</td>
</tr>
<tr>
<td>(A[7], B[12], WCOUT1107, WSOUT1108, WCOUT1207, WSOUT1207)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1208</td>
</tr>
<tr>
<td>(A[8], B[12], WCOUT1108, WSOUT1109, WCOUT1208, WSOUT1208)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1209</td>
</tr>
<tr>
<td>(A[9], B[12], WCOUT1109, WSOUT1110, WCOUT1209, WSOUT1209)</td>
<td></td>
</tr>
<tr>
<td>ARRAYCELL</td>
<td>arraycell1210</td>
</tr>
<tr>
<td>(A[10], B[12], WCOUT1110, WSOUT1111, WCOUT1210, WSOUT1210)</td>
<td></td>
</tr>
</tbody>
</table>
module ARRAYCELL (A, B, CIN, SIN, COUT, SOUT);
  input A, B, CIN, SIN;
  output COUT, SOUT;
  wire PP;
  ANDGATE and0 (PP, A, B);
  FULLADDER fulladder0 (PP, SIN, COUT, SOUT);
endmodule

module HALFADDER (A, B, COUT, SUM);
  input A, B;
  output COUT, SUM;
  assign SUM = A&B;
  assign COUT = A&B;
endmodule

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
### Simulation Reports and Logs from Altera Quartus II

**APPENDIX C**

<table>
<thead>
<tr>
<th>File Name with User-Entered Path</th>
<th>Used in Netlist</th>
<th>File Type</th>
<th>File Name with Absolute Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECURSIVEMULTIPLIER.v</td>
<td>yes</td>
<td>User</td>
<td>Verilog HDL File</td>
</tr>
<tr>
<td>C:/altera/quartus51/bin/Thesis/RECURSIVEMULTIPLIER.v</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RECURSIVEMULTIPLIER Analysis & Synthesis Source Files Read**

- Total logic elements: 2233
  - Combination with no register: 2057
  - Combination with a register: 160

**RECURSIVEMULTIPLIER Analysis & Synthesis Resource Usage Summary**

- Logic elements 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
- 1/0 pins: 129
- Maximum fan-out node: CK
- Maximum fan-out: 176
- Total fan-out: 7610
- Average fan-out: 3.22

---

Copyright (C) 1991-2005 Altera Corporation

Your use of Altera Corporation's design tools, logic functions and other software and tools, and its AMPP partner logic functions, and any output files any of the foregoing (including device programming or simulation files), and any associated documentation or information are expressly subject to the terms and conditions of the Altera Program License Subscription Agreement, Altera MegaCore Function License Agreement, or other applicable license agreement, including,
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

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

RECURSIVEMULTIPLIER Analysis & Synthesis Settings

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

<table>
<thead>
<tr>
<th>Option Setting Default Value</th>
<th>Device AUTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal Probe signals routed during normal compilation</td>
<td>Off Off</td>
</tr>
<tr>
<td>Use smart compilation</td>
<td>Off Off</td>
</tr>
<tr>
<td>Router Timing Optimization Level</td>
<td>Normal Normal</td>
</tr>
<tr>
<td>Placement Effort Multiplier</td>
<td>1.0 1.0</td>
</tr>
<tr>
<td>Router Effort Multiplier</td>
<td>1.0 1.0</td>
</tr>
<tr>
<td>Optimize Hold Timing</td>
<td>IO Paths and Minimum TPD Paths I0 Paths and Minimum TPD Paths</td>
</tr>
<tr>
<td>Optimize Fast-Corner Timing</td>
<td>Off Off</td>
</tr>
<tr>
<td>Optimize Timing Normal compilation</td>
<td>Normal compilation</td>
</tr>
<tr>
<td>Optimize IOC Register Placement for Timing</td>
<td>On On</td>
</tr>
<tr>
<td>Limit to One Fitting Attempt</td>
<td>Off Off</td>
</tr>
<tr>
<td>Final Placement Optimizations Automatically</td>
<td>Automatically Automatically</td>
</tr>
<tr>
<td>Fitter Aggressive Routability Optimizations</td>
<td>Automatically Automatically</td>
</tr>
<tr>
<td>Fitter Initial Placement Seed</td>
<td>1 1</td>
</tr>
<tr>
<td>Slow Slew Rate</td>
<td>Off</td>
</tr>
<tr>
<td>PCI I/O</td>
<td>Off</td>
</tr>
<tr>
<td>Weak Pull-Up Resistor</td>
<td>Off</td>
</tr>
<tr>
<td>Enable Bus-Hold Circuitry</td>
<td>Off Off</td>
</tr>
<tr>
<td>Auto Global Memory Control Signals</td>
<td>Off Off</td>
</tr>
<tr>
<td>Auto Packed Registers -- Stratix/Stratix GX</td>
<td>Auto Auto</td>
</tr>
<tr>
<td>Auto Delay Chains</td>
<td>On On</td>
</tr>
<tr>
<td>Auto Merge PLLs</td>
<td>On On</td>
</tr>
<tr>
<td>Perform Physical Synthesis for Combinational Logic</td>
<td>Off Off</td>
</tr>
<tr>
<td>Perform Register Duplication</td>
<td>Off Off</td>
</tr>
<tr>
<td>Perform Register Retiming</td>
<td>Off Off</td>
</tr>
<tr>
<td>Perform Asynchronous Signal Pipelining</td>
<td>Off Off</td>
</tr>
<tr>
<td>Fitter Effort Auto Fit</td>
<td>Auto Fit</td>
</tr>
<tr>
<td>Physical Synthesis Effort Level</td>
<td>Normal Normal</td>
</tr>
<tr>
<td>Logic Cell Insertion - Logic Duplication</td>
<td>Auto Auto</td>
</tr>
<tr>
<td>Auto Register Duplication</td>
<td>Off Off</td>
</tr>
<tr>
<td>Auto Global Clock</td>
<td>On On</td>
</tr>
<tr>
<td>Auto Global Register Control Signals</td>
<td>On On</td>
</tr>
</tbody>
</table>

### RECURSIVEMULTIPLIER Fitter Settings

<table>
<thead>
<tr>
<th>Option Setting Default Value</th>
<th>Device AUTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal Probe signals routed during normal compilation</td>
<td>Off Off</td>
</tr>
<tr>
<td>Use smart compilation</td>
<td>Off Off</td>
</tr>
<tr>
<td>Router Timing Optimization Level</td>
<td>Normal Normal</td>
</tr>
<tr>
<td>Placement Effort Multiplier</td>
<td>1.0 1.0</td>
</tr>
<tr>
<td>Router Effort Multiplier</td>
<td>1.0 1.0</td>
</tr>
<tr>
<td>Optimize Hold Timing</td>
<td>IO Paths and Minimum TPD Paths I0 Paths and Minimum TPD Paths</td>
</tr>
<tr>
<td>Optimize Fast-Corner Timing</td>
<td>Off Off</td>
</tr>
<tr>
<td>Optimize Timing Normal compilation</td>
<td>Normal compilation</td>
</tr>
<tr>
<td>Optimize IOC Register Placement for Timing</td>
<td>On On</td>
</tr>
<tr>
<td>Limit to One Fitting Attempt</td>
<td>Off Off</td>
</tr>
<tr>
<td>Final Placement Optimizations Automatically</td>
<td>Automatically Automatically</td>
</tr>
<tr>
<td>Fitter Aggressive Routability Optimizations</td>
<td>Automatically Automatically</td>
</tr>
<tr>
<td>Fitter Initial Placement Seed</td>
<td>1 1</td>
</tr>
<tr>
<td>Path Number</td>
<td>Type</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td>1</td>
<td>Worst-case tsu</td>
</tr>
<tr>
<td>2</td>
<td>Worst-case tco</td>
</tr>
<tr>
<td>3</td>
<td>Worst-case th</td>
</tr>
<tr>
<td>4</td>
<td>Clock Setup: 'CK'</td>
</tr>
</tbody>
</table>
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 whether this will -- migration. When -- tables. If it is a -- in a future design -- If it is an unused -- signal on the board -- for a different -- GND+: Unused input pin. It can also be used to report unused dual-purpose pins.
  -- 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.
<table>
<thead>
<tr>
<th>Pin Name/Usage</th>
<th>Location</th>
<th>Dir.</th>
<th>I/O Standard</th>
<th>Voltage</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCCINT</td>
<td>A1</td>
<td></td>
<td></td>
<td>1.5V</td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCIO4</td>
<td>A3</td>
<td></td>
<td></td>
<td>3.3V</td>
<td>4</td>
</tr>
<tr>
<td>GND*</td>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT[8]</td>
<td>A7</td>
<td></td>
<td>output</td>
<td>LVTTL</td>
<td>4</td>
</tr>
<tr>
<td>GND</td>
<td>A9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCIO4</td>
<td>A10</td>
<td></td>
<td>power</td>
<td>3.3V</td>
<td>4</td>
</tr>
<tr>
<td>AL[0]</td>
<td>A11</td>
<td></td>
<td>input</td>
<td>LVTTL</td>
<td>4</td>
</tr>
<tr>
<td>VCCIO3</td>
<td>A13</td>
<td></td>
<td>power</td>
<td>3.3V</td>
<td>3</td>
</tr>
<tr>
<td>GND</td>
<td>A14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AL[15]</td>
<td>A15</td>
<td></td>
<td>input</td>
<td>LVTTL</td>
<td>3</td>
</tr>
<tr>
<td>OUT[28]</td>
<td>A16</td>
<td></td>
<td>output</td>
<td>LVTTL</td>
<td>3</td>
</tr>
<tr>
<td>OUT[40]</td>
<td>A17</td>
<td></td>
<td>output</td>
<td>LVTTL</td>
<td>3</td>
</tr>
<tr>
<td>OUT[1]</td>
<td>A18</td>
<td></td>
<td>output</td>
<td>LVTTL</td>
<td>3</td>
</tr>
<tr>
<td>GND*</td>
<td>A19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCIO3</td>
<td>A20</td>
<td></td>
<td>power</td>
<td>3.3V</td>
<td>3</td>
</tr>
<tr>
<td>GND</td>
<td>A21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCINT</td>
<td>A22</td>
<td></td>
<td>power</td>
<td>1.5V</td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>AA1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA2</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>GND*</td>
<td>AA3</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>GND*</td>
<td>AA4</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>GND*</td>
<td>AA5</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>GND*</td>
<td>AA6</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>GND*</td>
<td>AA7</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>AL[15]</td>
<td>AA8</td>
<td></td>
<td>input</td>
<td>LVTTL</td>
<td>7</td>
</tr>
<tr>
<td>NC</td>
<td>AA9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NC</td>
<td>AA10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA11</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td>Voltage</td>
<td>Pin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT[61]</td>
<td>AA12</td>
<td>output</td>
<td>LVTTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT[50]</td>
<td>AA13</td>
<td>output</td>
<td>LVTTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND+</td>
<td>AA14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AA21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>AA22</td>
<td>gnd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCINT</td>
<td>AB1</td>
<td>power</td>
<td>1.5V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>AB2</td>
<td>gnd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCIO7</td>
<td>AB3</td>
<td>power</td>
<td>3.3V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>AB9</td>
<td>gnd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCIO7</td>
<td>AB10</td>
<td>power</td>
<td>3.3V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCIO8</td>
<td>AB13</td>
<td>power</td>
<td>3.3V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>AB14</td>
<td>gnd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>AB19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCIO8</td>
<td>AB20</td>
<td>power</td>
<td>3.3V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>AB21</td>
<td>gnd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCCINT</td>
<td>AB22</td>
<td>power</td>
<td>1.5V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>B1</td>
<td>gnd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND*</td>
<td>B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GND*: B3
GND*: B4
GND*: B5
GND*: B6
OUT[4]: B7: output: LV TTL
AH[9]: B8: input: LV TTL
NC: B9
NC: B10
AH[2]: B11: input: LV TTL
AL[6]: B12: input: LV TTL
AL[1]: B13: input: LV TTL
AL[13]: B14: input: LV TTL
OUT[18]: B15: output: LV TTL
OUT[25]: B16: output: LV TTL
OUT[32]: B17: output: LV TTL
OUT[44]: B18: output: LV TTL
OUT[16]: B19: output: LV TTL
GND*: B20
GND*: B21
GND: B22: gnd
VCCI05: C1: power: 3.3V
GND*: C2
GND*: C3
GND*: C4
GND*: C5
GND*: C6
XL[3]: N
XL[8]: N
XH[7]: N
NC: C10
NC: C11
XH[3]: C12: input: LV TTL
AH[1]: C13: input: LV TTL
OUT[19]: C14: output: LV TTL
XH[8]: C15: input: LV TTL

94

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL[12]</td>
<td>L15 : input : LVTTL</td>
</tr>
<tr>
<td>GNDG_PLL1</td>
<td>L18 : gnd</td>
</tr>
<tr>
<td>GMDA_PLL1</td>
<td>L19 : gnd</td>
</tr>
<tr>
<td>GND+</td>
<td>L20 :</td>
</tr>
<tr>
<td>GND+</td>
<td>L21 :</td>
</tr>
<tr>
<td>GND+</td>
<td>L22 :</td>
</tr>
<tr>
<td>GND+</td>
<td>M1 :</td>
</tr>
<tr>
<td>GND+</td>
<td>M2 :</td>
</tr>
<tr>
<td>GND+</td>
<td>M3 :</td>
</tr>
<tr>
<td>GND+</td>
<td>M4 : gnd</td>
</tr>
<tr>
<td>VCCA_PLL3</td>
<td>M5 : power : 1.5V</td>
</tr>
<tr>
<td>GND*</td>
<td>M6 :</td>
</tr>
<tr>
<td>GND+</td>
<td>M9 : gnd</td>
</tr>
<tr>
<td>VCCINT</td>
<td>M10 : power : 1.5V</td>
</tr>
<tr>
<td>GND</td>
<td>M11 : gnd</td>
</tr>
<tr>
<td>VCCINT</td>
<td>M12 : power : 1.5V</td>
</tr>
<tr>
<td>GND</td>
<td>M13 : gnd</td>
</tr>
<tr>
<td>VCCINT</td>
<td>M14 : power : 1.5V</td>
</tr>
<tr>
<td>GND*</td>
<td>M16 :</td>
</tr>
<tr>
<td>GND*</td>
<td>M17 :</td>
</tr>
<tr>
<td>VCCA_PLL2</td>
<td>M18 : power : 1.5V</td>
</tr>
<tr>
<td>GMDA_PLL2</td>
<td>M19 : gnd</td>
</tr>
<tr>
<td>GND+</td>
<td>M20 :</td>
</tr>
<tr>
<td>GND+</td>
<td>M21 :</td>
</tr>
<tr>
<td>GND+</td>
<td>M22 :</td>
</tr>
<tr>
<td>VCCIO6</td>
<td>N1 : power : 3.3V : 6</td>
</tr>
<tr>
<td>GND*</td>
<td>N2 :</td>
</tr>
<tr>
<td>GND*</td>
<td>N3 :</td>
</tr>
<tr>
<td>GNDG_PLL3</td>
<td>N4 : gnd</td>
</tr>
<tr>
<td>VCCG_PLL3</td>
<td>N5 : power : 1.5V : 1</td>
</tr>
</tbody>
</table>
GND* : W15 : : : 8
GND* : W16 : : : 8
GND* : W17 : : : 8
GND* : W18 : : : 8
GND* : W19 : : : 8
GND* : W20 : : : 8
GND* : W21 : : : 1
GND* : W22 : : : 1
VCCIO06 : 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[56] : Y14 : output : LVTTL : : 8
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.