Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks

by

Shreesha Srinath

A thesis submitted in partial fulfillment of the

requirements for the degree of

Master of Science

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2010

APPROVED

By

Advisor Signature: _____

Advisor Title: _____

Date:_____

Abstract

The introduction of asymmetric embedded multiplier blocks in recent Xilinx FPGAs complicates the design of larger multiplier sizes. The two different input bitwidths of the embedded multipliers lead to two different shifting factors for the partial products that must be summed. This makes even the most straightforward multiplier design less intuitive. In this thesis, I present a methodology and set of equations to automatically generate Verilog hardware description code for arbitrary multiplier sizes composed of arbitrarily-sized asymmetric embedded multiplier cores. The presented technique also uses intelligent rearrangement of the multiplier block outputs into partial product terms to reduce the overall delay of the circuit. Multipliers created with this generator are faster and use fewer DSP blocks than either those created using Xilinx Core Generator or those created by simply using the '*' operator in Verilog. It also uses fewer LUTs than those created using the '*' operator. Finally, the presented generator can create multipliers larger than possible with Core Generator, and is limited only by the number of available embedded multipliers.

Acknowledgement

I would like to thank my advisor Katherine Compton for providing me the opportunity to pursue this research under her supervision. I thank her for the invaluable guidance and support she has extended throughout the duration of my degree.

I would like to thank all the professors at University of Wisconsin-Madison for the courses they taught me and adding value to my education. I would like to thank Kyle Rupnow, Amin Farmahini-Farahani and Tony Gregerson for providing valuable inputs through the duration of this research.

Finally, I would like to thank my family for their constant support and encouragement to pursue higher studies. I would like to thank my friends Chaitanya Baone, Vaishali Karanth and Syed Gilani for making this experience memorable.

Table of Contents

List of Figures

List of Tables

1 Introduction

Multiplication is an important arithmetic function for many applications [1][2]. A key requirement for many of the DSP applications to achieve their needed performance is the availability of processing elements such as adders, multipliers, dedicated hardware for division and square root [3][4]. Modern FPGAs provide a heterogeneous mixture of different hardware blocks, such as dedicated memory blocks, carry-chains for addition and multipliers. These embedded multipliers have been included in the FPGAs for some time to improve multiplication performance [5][6][7][8][9]. Many DSP applications are highly-parallel, and thus demand a large number of these dedicated resources; furthermore, because the size of the embedded multiplier blocks is fixed within a particular FPGA family, larger multipliers are composed of multiple of these embedded blocks [10][11].

Decimal-digit: One decimal digit [0-9]

Digit: A grouping of binary bits, equal in bit-width to one of the inputs of the embedded multiplier block.

Digit-product: The result of multiplying two digits or two decimal-digits together.

Partial-product: Grouping of one or more digit-products into a partial result. For hand-multiplication, partial-products generally group the digit-products created by the same lower-operand digit.

Symmetric multiplier block: An $n \times n$ multiplier, where the inputs have equal bitwidth. The output is $2n$ bits.

Asymmetric multiplier block: An n × m multiplier, where $n \neq m$, and the output is $m + n$ bits.

Input: An input to a single embedded multiplier block, representing a single digit of an operand

Operand: A data word processed by the larger multiplier we construct from smaller embedded multipliers.

Figure 1: Definition of terms used in this thesis

The composition of large multipliers from small ones is relatively straightforward when the multiplier blocks are symmetrical—both inputs have equal bitwidth. However, some of the modern FPGAs now have asymmetric multiplier blocks [7][8], where the multiplier block inputs (and thus the digit-sizes used for each operand) differ in bit-width. This complicates the composable multiplier design; the shifts of *digit-products* that make up each *partial product* differ from the shifts of the partial products relative to one another. These terms are defined in Figure 1. As a result, digit-products from a given digit of an operand overlap in bit-positions, potentially increasing the additions required. In this work, we present a general methodology to apply the "divide and conquer approach" to implement large multipliers using asymmetric multiplier blocks. We divide the process into three steps namely: Operand Decomposition, Partial Product Generation and Partial Product summation. We implemented an automatic generator tool in MATLAB that produces synthesizable hardware description code for any given required multiplication computation using arbitrarily-sized (designer-specified) asymmetric multiplier blocks. [12].

2 Related Work

Previously, FPGAs supported multiplication only through the use of a collection of fine-grained logic structures. Mapping multipliers to these structures is inefficient due to the small granularity and large interconnection delays. Now FPGA vendors include dedicated, specialized multiplier blocks in their devices [5][13][7][8], which reduces the delay overheads of configurability and interconnections. These blocks often also include adders to implement multiply-accumulate operations; in these cases they are generally referred to as “DSP blocks”. Multiplications that are larger than the “native” embedded multiplier size must be composed from these embedded blocks, but this still provides dramatic improvement over constructing those large multipliers out of lookup-table (LUT) based logic.

The “divide and conquer” approach to construct large multipliers out of a set of small, symmetric multiplier blocks is a well-known technique [10][11][14][15]. The main steps involved in this process are (1) split the input operands into smaller multi-bit “digits”, (2) multiply the digits to form digit-products and generate a set of partial products by concatenating the digit-products, and finally (3) sum the generated partial products. In [15], the authors provide an overview of some of the popular techniques to compose a large multiplier using smaller symmetric blocks with focus on Xilinx Virtex-II devices. An improved multiplication approach and its application to the special case of squaring have been presented in [16]. The authors also present design techniques of parameterized fixed-point integer multiplication and fractional division units which use a hybrid of the embedded 18x18 multipliers and LUTs present in the Xilinx Virtex-II devices. The techniques in [15][16] are well suited for smaller sized integer computations.

In [14], the authors present an efficient methodology for the implementation of multiplication and squaring functions for large unsigned integers. They propose a general architecture for the multiplier and squarer which are composed by using smaller symmetric multiplier blocks. They explore timing- and area-oriented organizations of the partial products. In [17], the authors explore the design space of implementing large-size signed and unsigned multipliers using multi-granular embedded multipliers as found in the DSP blocks in Altera’s FPGAs [5]. These multipliers can be configured to operate as 9x9, 18x18 or 36x36-bit multipliers.

The authors in [18] explore three alternative types of large integer multiplier generation for FPGAs: Karatsuba-Ofman algorithm, non-standard tiling (an alternate, less regular form of divide and conquer) and specialized squarers. The Karatsuba-Ofman algorithm trades multiplications for additions by rearranging the creation of partial products and thereby reducing the number of multipliers/DSP blocks required. This methodology uses symmetric multiplier blocks. Their non-standard tiling technique optimizes large multiplier implementations using asymmetric multipliers. They decompose input operands using a mix of the two multiplier block input sizes (24 × 17 multipliers in the target Xilinx Virtex-5 device), each operand is decomposed into a mix of 24-bit digits and 17-bit digits, some of which may overlap. The resulting digit-products are carefully re-combined to form the final product. The aim is to minimize the overall multiplier block requirement. Furthermore, they use LUTs to implement small digit-products. The authors demonstrate this approach specifically for 41 x 41, 58 x 58 and 65 x 65 multipliers, but do not yet generalize the approach to arbitrary sizes. The work also does not discuss strategies for summing the resulting partial-products.

Wallace [19] and Dadda [20], introduced efficient compressor trees in the context for parallel multiplication which are widely used for ASICs and custom designs to speed-up the partial product summation. However, the LUT structures and fast-carry chains propagate chains in FPGAs blocks favor

the carry propagate adders (CPAs) [10][11]. The compressor tree adds multiple operands of a given bit size using carry save adders (CSAs) [10][11], as opposed to an adder tree which adds multiple operands using carry propagate adders. The authors in [21] present techniques to efficiently map compressor trees onto modern FPGA devices.

Finally, other techniques to optimize multiplication focus on fine-grained manipulations that are suitable for ASIC or custom implementations [22][23][24][25]. However, because FPGAs are multi-purpose devices, they must support more general multiplication structures. Thus these approaches are not ideal in most cases for multiplier implementation in FPGAs, particularly for large multiplications of two variables. This work instead focuses on attempting to find the best way to make use of the existing hardware already provided in FPGA devices.

3 Composable Multipliers with Symmetric Multiplier Blocks

Composing a larger multiplier using smaller multipliers is similar to "long multiplication"—the method generally used to perform multiplication by hand (Figure 2). Each decimal digit of one operand is multiplied by each decimal-digit of the other, but only one *digit-product* can be computed at a time. Digit-products for a particular decimal-digit of the lower operand are aggregated as they are computed; the "tens" digit of the digit-product is carried to the next position. It is then summed with the "ones" digit of the next digit-product. The complete result of processing one decimal-digit of the lower operand against all decimal-digits of the upper operand is a single *partial-product*. The definition of these terms as used in this thesis is given in Figure 1. For the Figure 2 example of that has three digits in the upper operand and three digits in the lower, we have three partial-products, P_0, P_1, and P_2, each created from three *digit-products*.

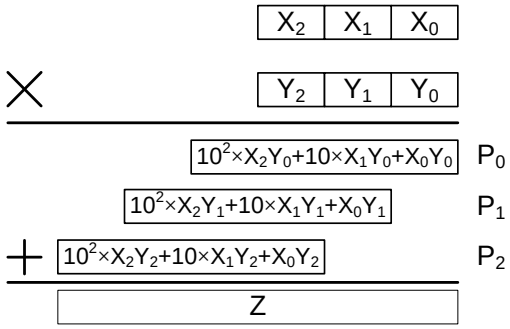


Figure 2: Long (hand) multiplication of two three-digit decimal numbers, where the "carries" of digit-products are immediately incorporated into the partial-products.

Like long-multiplication, we can divide our large binary operands into sets of *digits* that we can multiply using the smaller embedded multiplier blocks. If several multiplier blocks can be used to generate digit-products, waiting for the carry-out of the previous digit-product before combining it with the lower part of the next digit-product to compute the next digit of the partial-product (as in long-multiplication) creates a long critical path. Instead, with enough multiplier blocks, all digits from the top operand can be simultaneously multiplied with all digits from the bottom operand, creating a set of digit-products equal in number to the product of the digit counts of the operands.

The "divide and conquer approach" is explained as follows. Each digit is equal in bit-width to the input size of the multiplier blocks. For example, if the operand size for the needed large multiplier is 48 bits, with 8×8 multiplier blocks the operands would each contain 48/8 = 6 digits, and each digit would be 8 bits wide. If the operand size is not an exact multiple of the digit-size, it is zero-padded to fill the most-significant digit. Next, each digit from one operand is multiplied with each digit from the other operand, creating the set of digit-products. We will use this term throughout the paper to refer to the outputs of the embedded multipliers. These digit-products are shifted to the correct position depending on the position of the source digits in the input operands. The digit-products are then summed, generally using a tree of adders.

Figure 3 illustrates creating a larger multiplier from nine parallel (smaller) symmetric $n \times n$ multiplier blocks [10][11][14][17]. The bitwidth of each operand (X and Y) is three times the input bitwidth of the multiplier blocks. This could happen, for example, if each operand were 24 bits wide and we used 8×8

multipliers, or if each operand were 6 bits wide and we used 2 × 2 multipliers. The number of required multipliers is the product of the digit counts of the two operands, which in this case is 3 × 3 = 9.

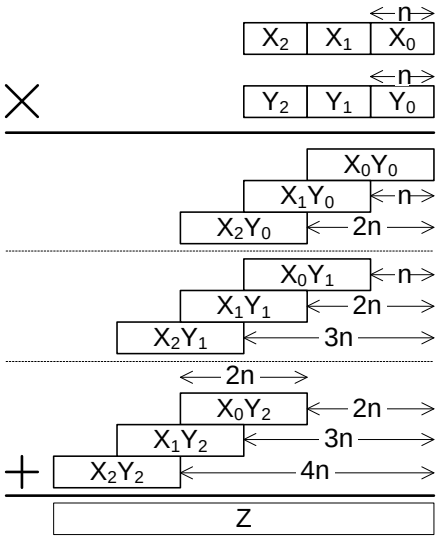


Figure 3: Multiplication of two binary values using parallel symmetric multiplier blocks to generate digit-products. Digit-products are summed with an adder tree.

Like long multiplication, we show the digit-products for digit Y_0 above the digit-products for digit Y_1, which in turn are above those for digit Y_2. Within the set of digit-products for a given lower-operand digit, each successive term is shifted by one digit position (*n*-bits) to the left. Also, each group of digit-products for a given lower-operand digit is shifted by one digit position (*n*-bits) relative to the digit-product group for the previous lower-operand digit. These shifts are based on the relative locations of the digits in the input operands. Note that because the embedded multiplier blocks are symmetric, the shift of digit-products within a group is equal to the relative shift between groups. The digits within a group are shifted by the digit size of X (*n*-bits), and the groups are shifted by the digit size of Y (*n*-bits). Finally, note that within a partial-product grouping, the upper half of one digit-product overlaps exactly with the lower half of the next digit-product.

After we create these digit-products, we can treat each digit-product as a separate partial product. The partial products are combined using an adder tree to produce the final result (labeled Z in the figure). The early stages of the tree may first combine digit-products from the same group and then sum the results for each group. However, the depth of the complete tree is still no less than $\lceil log_2 D \rceil$, where D is the number of digit-products.

The adder tree depth can be reduced by grouping adjacent but non-overlapping digit-products [10][11][14][17]. This grouping requires only concatenation, not addition, and thus no computational latency. Figure 4 shows the digit-products from Figure 3 rearranged in this method, with widest groupings listed towards the top. This creates a total of five partial-products, compared to the nine in Figure 3. Some digit-products cannot be grouped with any others because of overlaps. An adder tree that processes the nine individual digit-products shown in Figure 3 requires four levels; an adder tree that processes the five combined partial-products from Figure 4 requires only three levels. The size of the adders needed for the tree is not overall increased, since the original adder tree of Figure 3 would have required adders just as wide in its later levels. The benefit is that some of the digit-products are

"summed" by concatenation instead of actual addition, avoiding adder levels and carry chains in those cases.

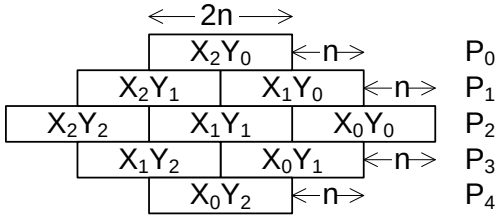


Figure 4: The same digit-products as Figure 3 but rearranged to reduce the depth of the required adder tree by first concatenating non-overlapping digit-products.

Each Xilinx Virtex-4 DSP48E block contains one signed 18x18 bit multiplier unit and a 48-bit addition/subtraction unit [26]. The multiplier unit can perform a 17x17 bit unsigned multiplication with the last bit of each input operand indicating the sign of the operand. The output of the multiplier can be cascaded to the adder of another DSP48E block with a fixed 17-bit shift dedicated routing.

Each Altera Stratix-II DSP block contains four 18x18 multipliers and two levels of adders [5]. It can also be configured to implement a 36x36 unit. Unlike the Xilinx Virtex-4 DSP48E block, it can implement the 18x18 unsigned multiplications. In Stratix-III device family two half-DSP blocks are grouped as a block [13]. A half-DSP block contains four 18x18 multipliers, two 36-bit adders and a 44-bit adder/accumulator unit, which has a cascaded input from the other half-DSP block in its group.

4 Composable Multipliers with Asymmetric Multiplier Blocks

Section 0 applied the divide-and-conquer strategy to compose large multiplications using symmetric multiplier blocks. Some have suggested expanding the technique to asymmetric multipliers blocks where one of the two multiplier block inputs is a multiple of the other by using multiple blocks to form a "square" multiplier, and then building the complete multiplier from a set of (pre-composed) square multipliers, again turning the problem into one of composition using symmetric blocks [11]. The use of asymmetric multiplier blocks has been suggested, but not explored [10]. However, embedded multiplier blocks in FPGAs may not have one input size that is a multiple of the other; in fact, the current FPGAs that have asymmetric multipliers have a 24×17 multiplier size [7][8]. Little work has yet examined methods for using these asymmetric multipliers to best advantage. One proposed technique shows, for a set of specific multiplier sizes, efficient methodologies to decompose operands and generate partial products. That work is potentially complementary to the work presented in this thesis, but does not examine the step of partial-product summation.

Figure 5 illustrates applying the divide-and-conquer strategy using asymmetric embedded multiplier blocks. In this example, the X operand is divided into two *n*-bit digits, and the Y operand into three *m*-bit digits. X and Y may each be 6 bits wide, with X divided into two 3-bit digits, and Y divided into three 2-bit digits, for use with a 2×3 embedded multiplier block. As before, the number of required multiplier blocks for parallel multiplication of the operand digits is the product of the digit counts of the two operands. In Figure 5, 2×3 = 6 embedded multiplier blocks are required to produce the six digit-products. Unlike in the symmetric case, X_1Y_0 and X_0Y_1 do not exactly overlap positions; X_1Y_0 is offset by *n* bits, whereas X_0Y_1 is offset by *m* bits. More specifically, the digit-products within a group of digit-products produced for the same lower-operand digit are offset from one another by *n* bits (the digit size of the upper operand), but the inter-group offset is *m* bits (the digit size of the lower operand).

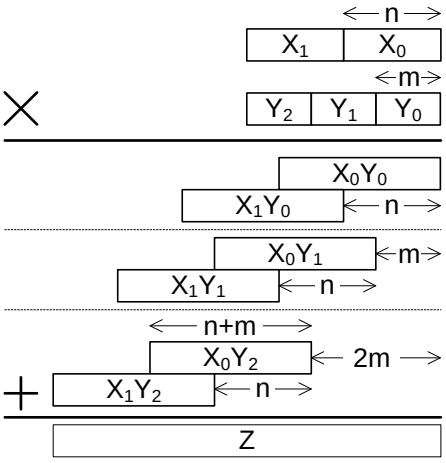


Figure 5: Digit-products and their positions when using asymmetric n × m embedded multiplier blocks. X is decomposed into n-bit digits and Y into m-bit digits.

To reduce the depth of the partial-product adder tree for the asymmetric multiplier block case, we can use a digit-product concatenation technique similar to what was used for symmetric multipliers. Applying this technique to the problem shown in Figure 5 gives the set of partial products shown in Figure 6. Although initially this shape may appear identical to Figure 4, the two different shift factors are an important feature. In the symmetric case there may be multiple ways to choose which digit-products to group into partial products because of the uniform shifting; in the asymmetric case the choices are more constrained, and digit-products within adjacent partial-products in the figure partially overlap. The original set of six partial products given in Figure 5 would require a three-level adder tree; the rearranged set given in Figure 6 requires only a two-level adder tree.

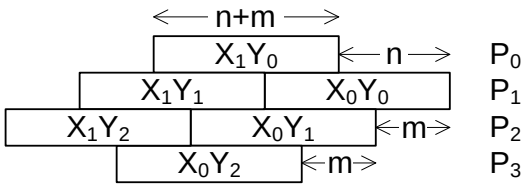


Figure 6: The same digit-products as Figure 5 rearranged similarly to Figure 4. Note the difference in offsets compared to the symmetric case.

The Xilinx Virtex-5 and Virtex-6 DSP48E blocks contain one signed 25x18 bit multiplier unit and a 48-bit addition/subtraction unit [7][8]. The multiplier unit can perform a 24x17 bit unsigned multiplication because the most-significant bit of each operand is reserved as a sign, not data, bit. As in case of the Virtex-4 devices, the output of the multiplier can be cascaded to the adder of another DSP48E block ("chained") with a fixed 17-bit shift dedicated routing. However, this is only useful when the required shift amount is exactly 17-bits. As discussed previously, the shift amount between partial products is not a fixed 17 bits in all cases. The summation of partial products thus requires a more complex summation tree, since it cannot take full advantage of DSP-block "chaining". This work focuses on using these asymmetric DSP blocks more efficiently in composable multipliers and presents a general methodology for their creation in the following sections.

5 Automatic Generation Method

In this section we present our method for composing a large size multiplication using the smaller asymmetric embedded blocks. The embedded multipliers are $n \times m$ in size, where n and m represent the bitwidths of the two multiplier inputs, and $n \neq m$. The multiplier that we create from the embedded cores implements $Z = X \times Y$, where operands X and Y may or may not have equal bitwidths. We separate the process of implementing the X×Y multiplier into three steps: operand decomposition, partial product generation (which includes the concatenation step), and partial product summation. Note that the presented operand decomposition step defines new variables used in later steps to simplify equations.

5.1 Operand Decomposition

The first step is to decompose the input operands into sets of digits that match the input sizes of the $n \times m$ embedded multiplier blocks. Unlike when using symmetric embedded multipliers, we have two possible options to consider for decomposition: we can decompose X by n and Y by m, or X by m and Y by n. The number of digits obtained by decomposing each of the input operands determines the total number digit-products or the total number of the DSP blocks needed for the implementation. The input operands are decomposed into multiples of the input sizes $n \times m$ embedded multiplier blocks. If the input size of an operand is not an exact multiple of the inputs (m, n) of the embedded multiplier, the last digit obtained by the decomposition is zero-padded to match the nearest multiple. Exploiting the leading zeros and approaches similar to the non-standard tiling [18] is a task considered for future.

The goal in this step therefore is to minimize the total number of digit-products and hence, the total number of partial product summations. To minimize the number of used multiplier blocks, we should choose the decomposition that minimizes the number of digit-products. The number of digit-products is the product of the number of digits in each of the the two operands ($C_X \times C_Y$). Thus we compare the two options given in Figure 7, and choose the C_X and C_Y pair that gives the smallest $C_X \times C_Y$ product. If both options result in an equal product, we choose the option with the smallest $C_X + C_Y$ sum, because the number of overall additions needed is $C_X + C_Y - 1$. Next, based on the chosen decomposition, we choose our “upper” operand A and “lower” operand B for generating the partial products, such that B has as many or more digits than A. This simplifies the notation in the generation algorithm. Finally, we set j to be the digit size of A, and k to be the digit size of B (where $\{k = n, j = m\}$ or $\{k = m, j = n\}$, depending on the decomposition chosen). The algorithm for this process is given in Figure 7.

let $C_{XN} = \left\lceil \frac{X_{bitwidth}}{n} \right\rceil$ and $C_{YM} = \left\lceil \frac{Y_{bitwidth}}{m} \right\rceil$

let $C_{XM} = \left\lceil \frac{X_{bitwidth}}{m} \right\rceil$ and $C_{YN} = \left\lceil \frac{Y_{bitwidth}}{n} \right\rceil$

if ($C_{XN} \times C_{YM}$) < ($C_{XM} \times C_{YN}$) then decompose X by *n* and Y by *m*
else if ($C_{XM} \times C_{YN}$) < ($C_{XN} \times C_{YM}$) then decompose X by *m* and Y by *n*
else if ($C_{XN} + C_{YM}$) < ($C_{XM} + C_{XN}$) then decompose X by *n* and Y by *m*
else if ($C_{XM} + C_{YN}$) < ($C_{XN} + C_{YM}$) then decompose X by *m* and Y by *n*
else arbitrarily decompose X by *n* and Y by *m*

if X decomposed by *n* then $C_X = C_{XN}$, else $C_X = C_{XM}$
if Y decomposed by *n* then $C_Y = C_{YN}$, else $C_Y = C_{YM}$
if ($C_X < C_Y$) then A = X, $C_A = C_X$, B = Y, $C_B = C_Y$
else A = Y, $C_A = C_Y$, B = X, $C_B = C_X$
let *j* = *n* if A decomposed by *n*, else *j* = *m*
let *k* = *m* if B decomposed by *m*, else *k* = *n*

Now we have:

$$A = 2^{(C_A-1)j} A_{C_A-1} + 2^{(C_A-2)j} A_{C_A-2} + \cdots + 2^j A_1 + A_0$$
$$B = 2^{(C_A-1)k} B_{C_B-1} + 2^{(C_B-2)k} B_{C_B-2} + \cdots + 2^k B_1 + B_0$$

Figure 7: Decomposition of operands for use with asymmetric embedded multiplier blocks.

5.2 Partial Product Generation

The multiplier output Z is calculated as shown in Equation 1. The digit-products of this equation can also be represented as a grid (Figure 8). Digit-products in the grid are not shown shifted to their exact positions as in Figure 6; instead the grid highlights different ways of grouping digit-products. Figure 5 represents a "horizontal" grouping, and Figure 6 represents a "diagonal" grouping. Digits along the same line (horizontal, vertical, or diagonal) are grouped into the same partial product before the partial products are summed. A horizontal grouping most closely mimics long multiplication, but a diagonal grouping can use concatenation to group digit-products instead of addition (unlike horizontal and vertical groupings). Figure 9 shows the diagonal grouping for the general case.

$$Z = A \times B$$

$$= \left(2^{(C_A-1)j} A_{C_A-1} + 2^{(C_A-2)j} A_{C_A-2} + \cdots + 2^j A_1 + A_0\right) \times \left(2^{(C_B-1)k} B_{C_B-1} + 2^{(C_B-2)k} B_{C_B-2} + \cdots + 2^k B_1 + B_0\right)$$

$$\begin{aligned} = {} & \left(2^{(C_A-1)j+(C_B-1)k} A_{C_A-1} B_{C_B-1}\right) + \left(2^{(C_A-1)j+(C_B-2)k} A_{C_A-1} B_{B-2}\right) + \cdots + \left(2^{(C_A-1)j+k} A_{C_A-1} B_1\right) \\ & + \left(2^{(C_A-1)j} A_{C_A-1} B_0\right) + \left(2^{(C_A-2)j+(C_B-1)k} A_{C_A-2} B_{C_B-1}\right) + \cdots + \left(2^{(C_A-2)j+k} A_{C_A-2} B_1\right) \\ & + \left(2^{(C_A-2)j} A_{C_A-2} B_0\right) + \cdots + \left(2^{j+(C_B-1)k} A_1 B_{C_B-1}\right) + \left(2^{j+(C_B-2)k} A_1 B_{C_B-2}\right) + \cdots + \left(2^{j+k} A_1 B_1\right) \\ & + \left(2^j A_1 B_0\right) + \left(2^{(C_B-1)k} A_0 B_{C_B-1}\right) + \left(2^{(C_B-2)k} A_0 B_{C_B-2}\right) + \cdots + \left(2^k A_0 B_1\right) + \left(A_0 B_0\right) \end{aligned}$$

Equation 1: Computing the final product Z from operand A with CA j-bit digits and operand B with CB k-bit digits

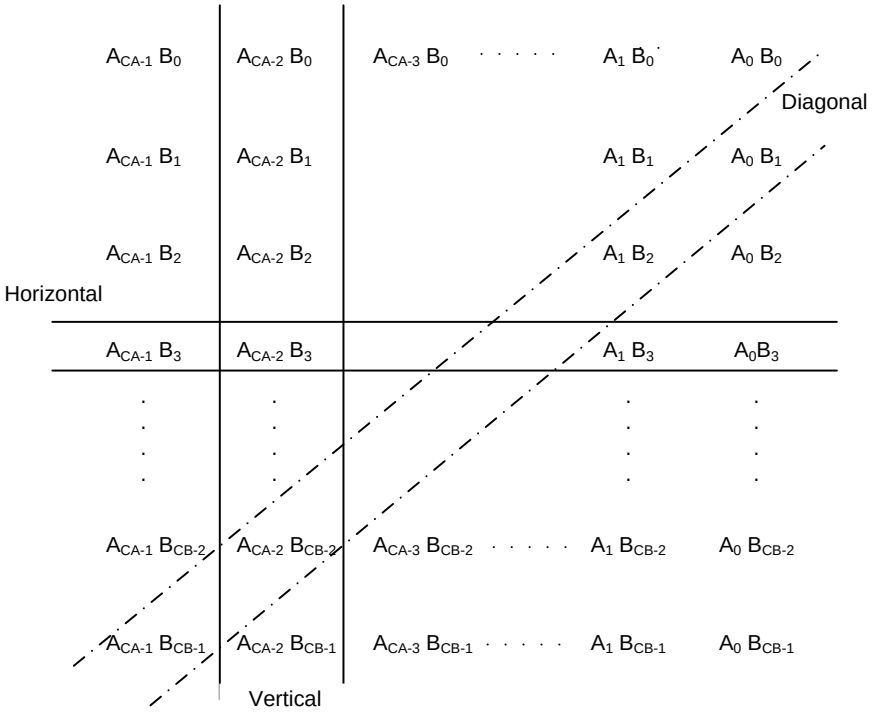


Figure 8: Grid of digit-products produced when multiplying the C_A digits of A (*j* bits each) with the C_B digits of B (*k* bits each). Partial products are formed by grouping digit-products horizontally, vertically, or diagonally as shown. Figure 5 represents a horizontal digit grouping, Figure 6 represents a diagonal grouping. If $C_A = C_B$ the grid is a square; if $C_A \neq C_B$, the grid is a rectangular, taller than it is wide (because decomposition forces $C_A \leq C_B$).

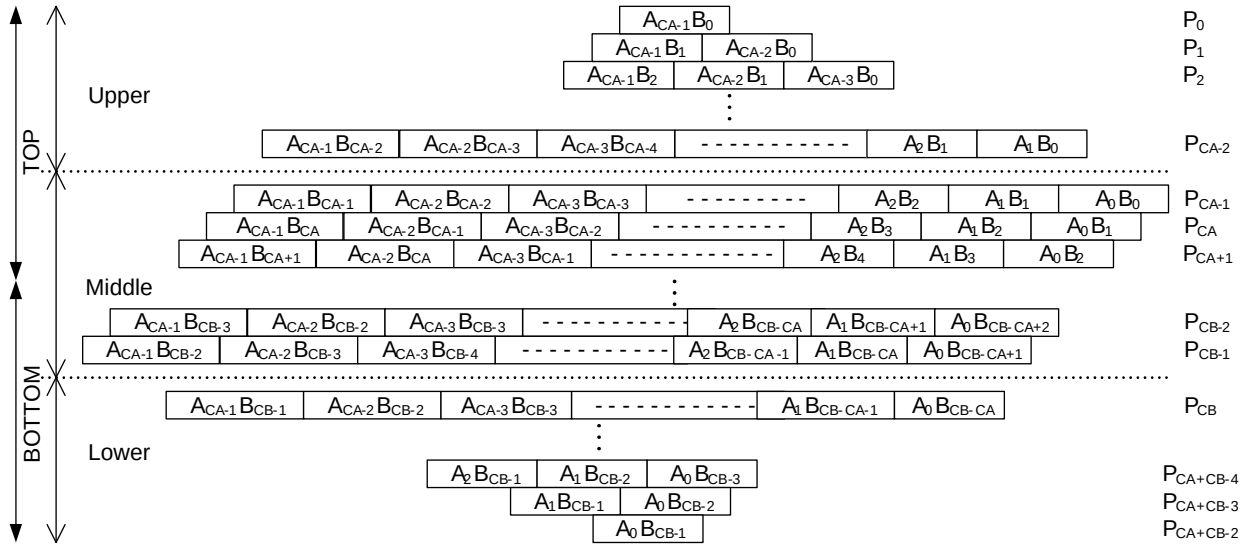


Figure 9: Diagonally-grouped partial products created with asymmetric multiplier blocks, like Figure 6 but for the general case. Partial products come from three regions, the "Upper", "Middle" and "Lower", which we re-partition into "Top" and "Bottom"

The arrangement of digit-products shown in Figure 9 creates three partial product "regions". The Middle region contains all of the "widest" partial-products (those with the maximum number of concatenated digit-products). The Upper contains digit-products on diagonal lines up and to the left of the Middle lines. The Lower contains the digit-products on diagonal lines down and to the right of the Middle lines. The bitwidth of Middle partial-products are identical. The bitwidth of partial-products in the Upper and Lower regions grows (shrinks) linearly. The first Middle partial-product is not shifted; its least-significant bit is position zero. Successive Middle partial-products are each shifted by *k* bits to the left with respect to the previous Middle partial-product. In the Upper region, the widest partial-product begins at position *j*; each partial-product above it is shifted *j* bits further to the left than the one below. In the Lower region, the least-significant bit of the widest partial-product is shifted *k* bits to the left of the least-significant bit of the last partial-product in the Middle region. Each successive Lower partial-product is shifted *k* more bits to the left. To determine how many partial-product terms should lie in the Upper, Middle, and Lower regions, we use the calculations given in Table 1. These equations are based on the calculations performed in Figure 7; for example, they require that $C_A \leq C_B$.

Region	# of Partial Products	Partial-Product Bitwidth
Middle	$\boldsymbol{C_B - C_A + 1}$	$\boldsymbol{C_A} \times (j + k)$
Upper	$\boldsymbol{C_A} - 1$	$s \times (j + k)$ where $\boldsymbol{0 \leq s < C_A - 1}$
Lower	$\boldsymbol{C_A} - 1$	$(\boldsymbol{C_A + C_B} - 1 - \mathrm{s}) \times (j + k)$ where $\boldsymbol{C_B \leq s < C_A + C_B - 1}$

Table 1: Bitwidths of partial-products for each region, where s is the index of partial-product given in Figure 9.

For our adder trees, we re-partition the partial products using two methods. In one method, we create separate adder trees for the portions labeled "Top" and "Bottom" in Figure 9, then sum the result. When the total number of partial products is odd and therefore cannot be evenly split between Top and Bottom, we combine the "middle" term into the Top grouping. In the other method, we create the adder tree from the entire set of partial products, without region subdivision. These adder tree strategies are discussed further in the following section.

5.3 Partial Product Summation

The number of the terms in each region is related to the sizes of the inputs of the asymmetric multiplier and the resulting number of digits in our operands (Table 1). The total number of partial products is C_A+C_B-1, which is controlled by the number of digits in operands A and B. The number of partial products in the Upper and Lower regions is controlled by the number of digits of the operand with the smallest digit count (A). The number in the middle region is then calculated by subtracting the number of upper and lower terms from the total number of partial products. The goals of this stage can be twofold: minimize the overall delay of the implementation or minimize the overall resources used resulting in lesser area. The FPGAs blocks favor the Carry Propagate Adders and we rely on the Xilinx synthesizer tools to optimize for the Verilog '+' operator. The overlaps of the partial product terms vary according to

the region belong. To exploit this region-wise overlap patterns the following four adder strategies are considered.

5.3.1 Delay Table

In [27], the authors present a method of constructing a delay table to aid in the generation of adder trees. The rows and columns of the delay table represent all the partial products which need to be added at a given level. The entries of each column indicate the required size of the adder to add the column partial product to the partial product of each row. Using the delay table, one first chooses the smallest possible adder in the entire table. The partial products belonging to the row and column of the corresponding entry are then marked as used and excluded from further consideration. The process is repeated, finding the next-smallest adder at each step until all partial products have been used. If the number of partial products is odd, the single remaining partial-product at the end of this process is incorporated into the new delay table constructed for the next level of addition. The other entries of this table are the sum results from the first table. The algorithm repeats until the final sum is obtained. We can consider all of the partial products in a single adder-tree construction, or divide the partial products into top and bottom regions and apply the algorithm separately in each region. The step of constructing the delay table in both the cases is illustrated in section 5.4.3

5.3.2 Outside-in

An alternative approach is to process partial-products from the “outside in”, combining the topmost partial-product with the bottom-most partial-product and repeating the addition steps until all the partial products are consumed in the first level. If the number of partial products in the first level is odd, the final partial product (the middle partial product) is considered in the next stage of addition. This strategy is applied at each level of the partial-product summation and is close to what is also used in [14] where the strategy is explored for summing partial-products for multipliers using symmetric (rather than asymmetric) embedded blocks. Similar to the previous strategy, the partial-products could be processed as a single whole block or be divided into top and bottom regions. Section 5.4.3 illustrates the process for both the above cases of Outside-In approach.

5.3.3 Addition types

When summing two partial products, we take advantage of the fact that their least-significant positions do not align. The sum of two partial products is thus partly a concatenation and partly a sum. We call this addition as “Ripple Adder” (RA). The other alternative is to keep track of the carry-bits generated from various levels of the partial product additions and defer the processing of these carry bits until the final stage. In this case we refer only to the carry-out bit at the most-significant position where the two summed partial-products overlap (and thus it is not a true “Carry Save” adder). The strategy allows for more concatenations and minimizes the overall resources required for the addition stage. This type of adder is called “Carry Vector Adder” (CVA). The carry vector holds the generated carry-out bits for every step of the partial-product summations, and allows individual addition steps to only add the overlapping portions of the partial products and concatenate both the lower and upper non-overlapping portions. If more than one carry-bit would be generated for a given bit-position, the tool currently instead uses a Ripple Adder. However, this case does not occur for the experiments presented in this thesis, which use

a 24 × 17 multiplier size, because the two digit-sizes do not share any common factors. The adder tree strategies discussed in the previous section can use either of the two adder types. There is a separate carry vector for each region when the partial-products are separated into Top and Bottom regions. The scenarios for each adder type depending on the overlap patterns are shown in Figure 10 and Figure 11.

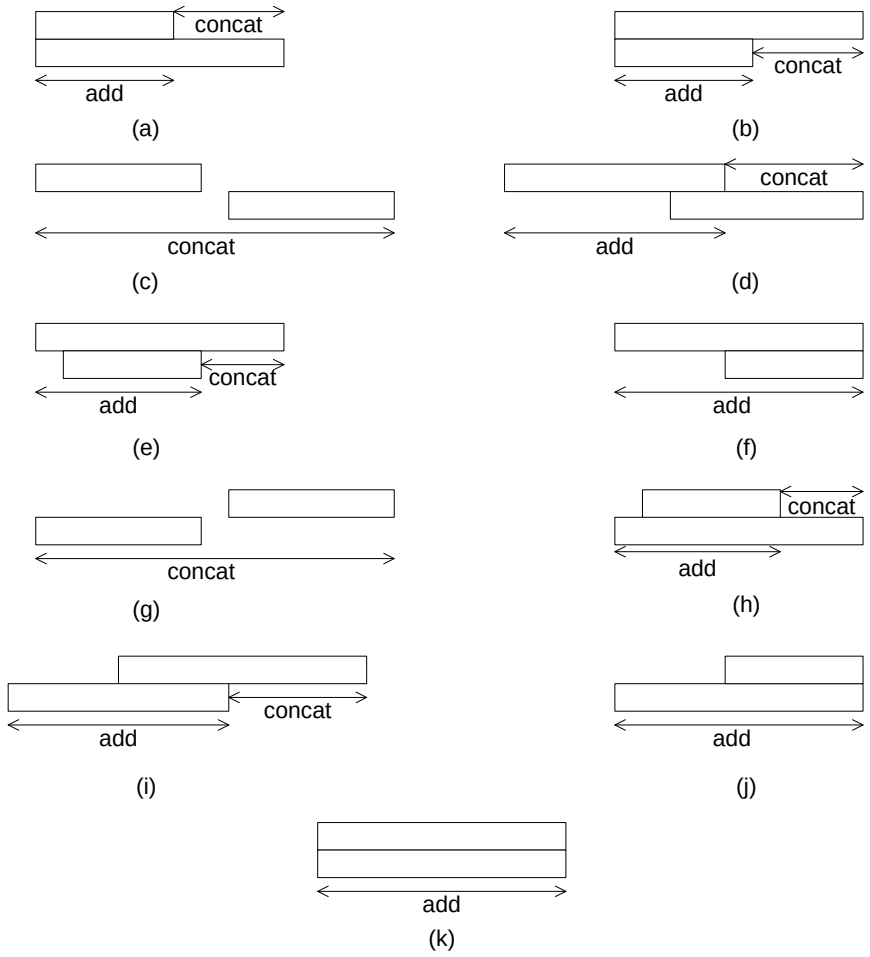


Figure 10: Addition/concatenation scenarios for the Ripple Adder

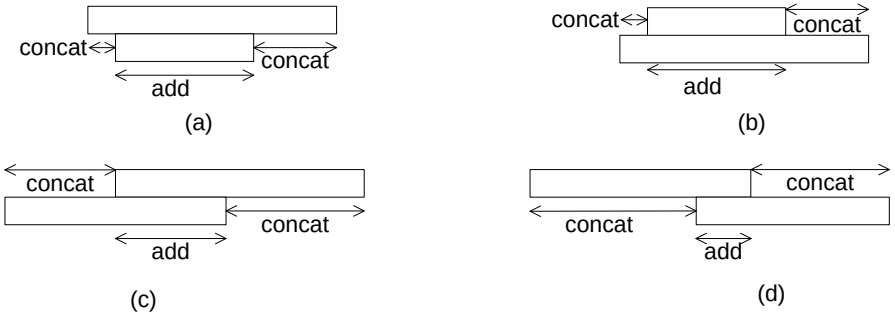


Figure 11: Addition/concatenation scenarios for the Carry Vector Adder where they differ from the Ripple Adder

5.4 Example of Composing 64x64 Multiplier From 24x17 Multiplier Blocks

Given below is the illustration of applying all the methods discussed above to the case of a 64x64 multiplication composed by using 24x17 embedded multipliers. This creates a Z = X × Y multiplier, where X and Y are each equal to 64 bits, *m* = 24 bits, and *n* = 17 bits.

5.4.1 Step 1: Operand Decomposition

Option 1: Decompose X by n and Y by m,

$$C_{XN} = \left\lceil \frac{64}{17} \right\rceil = 4 \text{ , } C_{YM} = \left\lceil \frac{64}{24} \right\rceil = 3$$

Option 2: Decompose X by m and Y by n,

$$C_{XM} = \left\lceil \frac{64}{24} \right\rceil = 3 \text{ , } C_{YN} = \left\lceil \frac{64}{17} \right\rceil = 4$$

Since ($C_{XN} \times C_{YM}$) = ($C_{XM} \times C_{YN}$) and ($C_{XN} + C_{YM}$) = ($C_{XM} + C_{YN}$) , Decompose arbitrarily,

$$C_X = C_{XN} \text{ , } \quad C_Y = C_{YM}$$

We have, ($C_X > C_Y$)

$$A = 72 \text{ , } \quad C_A = C_{YM} = 3 \text{ , } \quad j = 24$$

$$B = 68 \text{ , } \quad C_B = C_{XN} = 4 \text{ , } \quad k = 17$$

The multiplier structure to be implemented is actually 72×68 instead of 64×64 . The unused bits in the last digits of *A* and *B* are zero-padded to make the obtained digits integral multiples of 24 and 17.

Equation 2: Decomposed Example Operands

$$A = 2^{2j} A_2 + 2^j A_1 + A_0$$

$$B = 2^{3k} B_3 + 2^{2k} B_2 + 2^k B_1 + B_0$$

5.4.2 Step 2: Partial Product Generation

Figure 12 shows the digit-products obtained and their alignments with respect to each other. Digit-products belonging to the same digit of the lower operand B shift by the bit-width *j* of the smaller multiplier and the digit-products belonging to different digits of the lower operand B shift by bit-width *k* of the smaller multiplier.

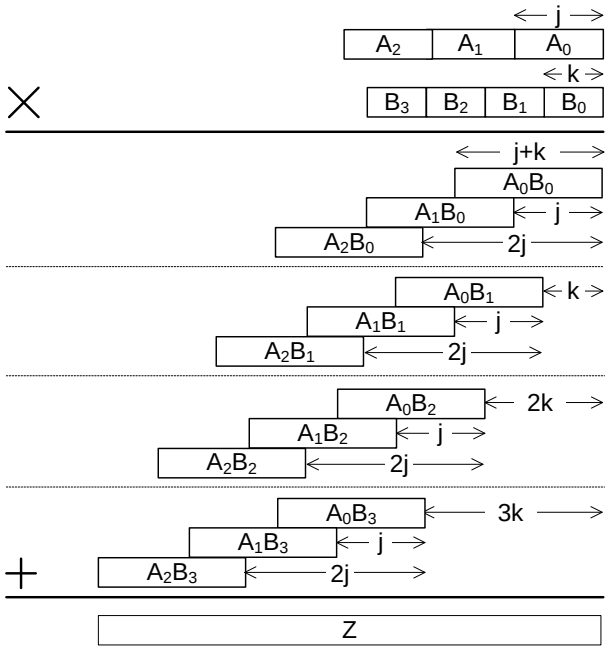
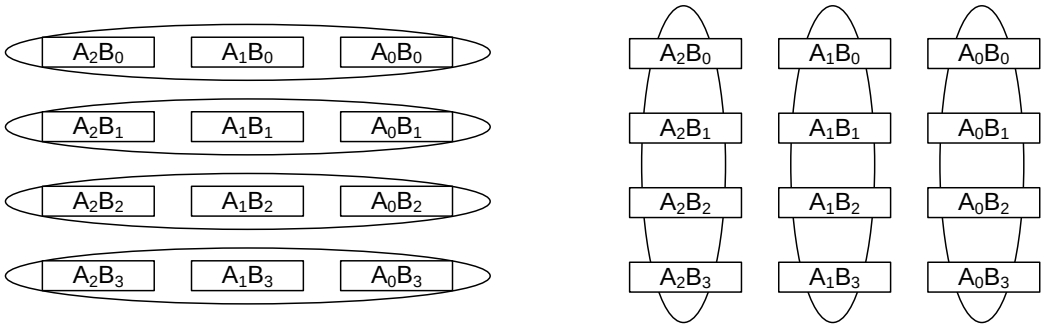
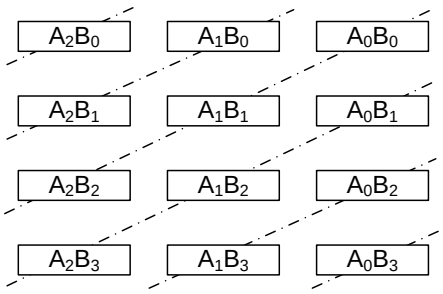


Figure 12: The digit-products of the 72 x 68 multiplier



(a) Horizontal Grouping

(b) Vertical Grouping



(c) Diagonal Grouping

Figure 13: Grid Illustrations for the (a) Horizontal, (b) Vertical and (c) Diagonal Groupings

For the current example, the number of additions required for the horizontal, vertical and diagonal groupings and the number of generated partial products are given in the table below.

Grouping	# of Partial Products	Total number of additions
Horizontal	4	11
Vertical	4	11
Diagonal	6	5

Table 2: Partial Product Groupings

The partial-products generated by the horizontal and vertical groupings require adders to combine the digit-products belonging to the same digit of operand B and A respectively whereas, the partial-products obtained by the diagonal grouping are mere concatenations of the non-overlapping digit-products. Although, the diagonal grouping yields more number of partial-products compared to the horizontal and vertical groupings, it requires the least number of overall additions to obtain the final result. The horizontal grouping and the vertical grouping would require the same number of addition operations but the required size of the adders for the two groupings would differ due to the different shifts between the generated partial products. Table 3 shows the start and the end bit positions of each partial product term along with the bitwidth of each term.

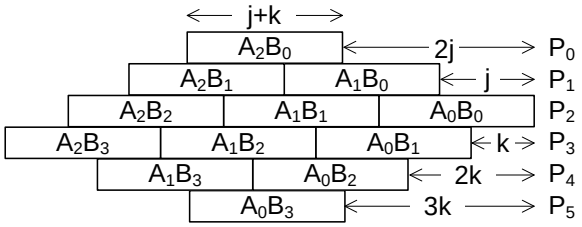


Figure 14: Partial Products obtained by the Diagonal Grouping

Term	Bitwidth	Start Bit	End Bit
P0	41	48	88
P1	82	24	105
P2	123	0	122
P3	123	17	139
P4	82	34	115
P5	41	51	91

Table 3: Partial Product bitwidth, and start and end bit positions

5.4.3 Step 3: Partial Product Summation

Figure 14 shows the six partial products obtained by the diagonal grouping for the given example. Once the partial products are obtained the last step is the partial product summations which yield the final results. The partial product summation is a multilevel addition step and we consider the two mentioned adder types to illustrate the tree generation methods. We have six partial products to sum and hence, require three levels of addition.

5.4.3.1 Ripple Adder (RA)

Delay Table – Whole (DW)

Based on the Delay-Table method tables below (Table 4, Table 5) list the possible adders at each level and the adder size selected for each column is shown in bold. Note that the partial-products are treated as one whole block.

LEVEL 1	P0	P1	P2	P3	P4	P5
P0	–					
P1	59	–				
P2	76	100	–			
P3	93	117	**124**	–		
P4	69	**83**	90	107	–	
P5	**42**	56	73	90	66	–

Table 4: Adders at Level 1 – Delay Table Whole (Ripple Adder)

LEVEL 2	P0_P5	P1_P4	P2_P3
P0_P5	–		
P1_P4	**70**	–	
P2_P3	94	118	–

Table 5: The adders at Level 2 – Delay Table Whole (Ripple Adder). The partial products summed to form the values at this level are indicated in the table heading.

Level 3: P0_P5_P1_P4 + P2_P3 (119-bit adder)

Delay Table – Top and Bottom (DTB)

The strategy of constructing the delay table can also be applied by splitting the partial products in Figure 14 into two regions: Top and Bottom. The tables below (Table 6, Table 7, Table 8) list the possible adders at each level and the adder size selected for each column is shown in bold for each region.

LEVEL 1 (TOP)	P0	P1	P2
P0	–		
P1	**59**	–	
P2	76	100	–

Table 6: The adders at level 1 for Top Region – Delay Table Top & Bottom (Ripple Adder)

LEVEL 1 (BOTTOM)	P3	P4	P5
P3	–		
P4	107	–	
P5	90	**66**	–

Table 7: The adders at level 1 for Bottom Region– Delay Table Top & Bottom (Ripple Adder)

LEVEL 2	
TOP	P0_P1 + P2 (100-bit adder)
BOTTOM	P4_P5 + P3 (107-bit adder)

Table 8: The adders at level 2 – Delay Table Top & Bottom (Ripple Adder). P0_P1 is the sum of partial products P0 and P1 that were summed in level 1, P4_P5 is the sum of P4 and P5 in level 1.

Level 3: P0_P1_P2 + P4_P5_P3) (125-bit adder)

Outside-in Whole (OIW)

For the example considered, the adders at each level for the outside-in method treating the partial products as a whole results in the same adders as in the delay table – whole method.

Outside-in – Top & Bottom (OITB)

The outside-in method can also be applied by splitting the partial products in Figure 14 into two regions: Top and Bottom. The tables below (Table 9, Table 10, Table 11), list the possible adders at each level and the adder size selected for each column is shown in bold for each region.

LEVEL 1 (TOP)	P0	P1	P2
P0	–		
P1	59	–	
P2	**76**	100	–

Table 9: The adders at level 1 for Top Region – Outside-in Top & Bottom (Ripple Adder)

LEVEL 1 (BOTTOM)	P3	P4	P5
P3	–		
P4	107	–	
P5	**90**	66	–

Table 10: The adders at level 1 for Bottom Region – Outside-in Top & Bottom (Ripple Adder)

LEVEL 2	
TOP	P0_P2 + P1 (101-bit adder)
BOTTOM	P3_P5 + P4 (108-bit adder)

Table 11: The adders at level 2 – Outside-in Top & Bottom (Ripple Adder). P0_P2 and P3_P5 are the sums generated in level 1 from the indicated partial products.

Level 3: P0_P2_P1) + P3_P5_P4) (126-bit adder)

5.4.3.2 Carry Vector Adder (CVA)

Delay Table – Whole (DW)

Based on the Delay-Table method tables below (Table 12, Table 13) list the possible adders at each level and the adder size selected for each column is shown in bold. The generated carry bits for each addition

are stored in the Carry Vector which is added after the last stage of addition. Note that the partial-products are treated as one whole block.

LEVEL 1	P0	P1	P2	P3	P4	P5
P0	–					
P1	41	–				
P2	41	82	–			
P3	41	82	**106**	–		
P4	41	**72**	82	82	–	
P5	**38**	41	41	41	41	–

Table 12: The adders at level 1 – Delay Table Whole (Carry Vector Adder)

LEVEL 2	P0_P5	P1_P4	P2_P3
P0_P5	–		
P1_P4	**44**	–	
P2_P3	44	92	–

Table 13: The adders at level 2 – Delay Table Whole (Carry Vector Adder). P0_P5, P1_P4, and P2_P3 were created in level 1 by summing the indicated partial products

Level 3: P0_P5_P1_P4 + P2_P3 (92-bit adder)

Level 4: P0_P5_P1_P4_P2_P3 + CarryVector (52-bit adder)

Delay Table – Top & Bottom (DTB)

The strategy of constructing the delay table can also be applied by splitting the partial products in Figure 14 into two regions: Top and Bottom. The tables below (Table 14, Table 15, Table 16), list the possible adders at each level and the adder size selected for each column is shown in bold for each region. The generated carry bits for each addition are stored in the Carry Vector which is added after the last stage of addition in each region. Note the top and bottom regions each hold a carry-vector.

LEVEL 1 (TOP)	P0	P1	P2
P0	–		
P1	**41**	–	
P2	41	82	–

Table 14: The adders at level 1 for Top Region – Delay Table Top & Bottom (Carry Vector Adder)

LEVEL 1 (BOTTOM)	P3	P4	P5
P3	–		
P4	82	–	
P5	41	**41**	–

Table 15: The adders at level 1 for Bottom Region– Delay Table Top & Bottom (Carry Vector Adder)

LEVEL 2	
TOP	P0_ P1 + P2 – 82 bit-adder
BOTTOM	P4_ P5 + P3 – 82 bit-adder

Table 16: The adders at level 2 – Delay Table Top & Bottom (Carry Vector Adder). P0_P1 and P4_P5 are sums created in level 1 from the indicated partial products.

Level 3 (Top): P0_P1_P2 + CarryVector-Top (18-bit adder)

Level 3 (Bottom): P4_P5_P3 + CarryVector-Bottom (25-bit adder)

Level 4: TopSum + BottomSum (125-bit adder)

Outside-in Whole (OIW)

For the example considered, the adders at each level for the outside-in method treating the partial products as a whole results in the same adders as in the delay table – whole method.

Outside-in – Top & Bottom (OITB)

For the example considered, the adders at each level for the outside-in method treating the partial-products as a whole results in the same adders as in the delay table – top & bottom method.

6 DSP-Only Implementation

The DSP48E blocks in the Xilinx Virtex-5 and Virtex-6 architectures support various functions including multiply, multiply-and-accumulate (MAC), three-input add, barrel shifting, pattern detect, comparator and bit-wise logic functions [7][8]. The DSP48Es are organized as columns and include dedicated routing paths between the blocks which allow them to be efficiently connected together to implement a wider range of DSP functionality.

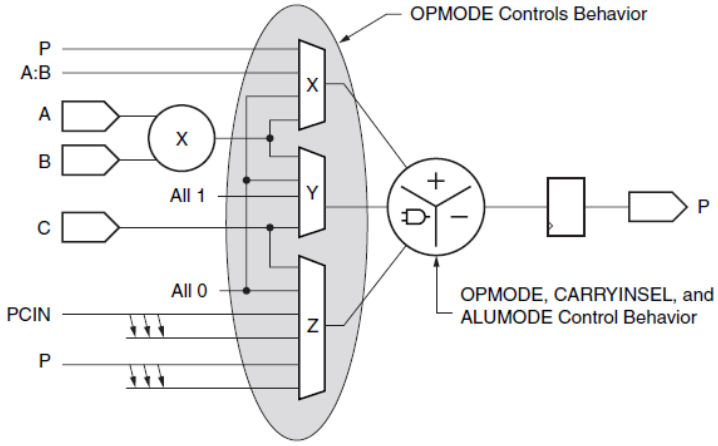


Figure 15: Simplified DSP48E functionality [7]

The DSP48E block has a 25 x 18 bit multiplier where the direct input A can accept a 30-bit input, of which 25 bits are used for the multiplier and the direct input B has 18 bit input (Figure 15). The full 30 bits of A can be concatenated with the 18 bits of B as an input to a logic operation or addition/subtraction in the remaining DSP block logic. The direct input C can accept a 48 bit input which can be added to the result of the 25 x 18 bit multiplier. The blocks also include pipeline registers, but we do not yet handle pipelined multiplier generation. The output P is the result of a multiply or multiply-and-accumulate operation. The PCIN input is the cascaded carry input from the output of a previous block. There is also a dedicated 17-bit cascaded output bus which can feed into the adder/subtractor of the next DSP48E block. This bus is, at least in part, specifically intended to aid in the composition of larger multipliers.

A DSP48E block's 48-bit adder can implement the required addition operations for composable multipliers by cascading the output of one digit-product multiplication to the adder input of the next block, summing the results of multiple stages. These computations can be efficiently pipelined by using the included pipeline registers of the DSP48E block. The throughput of such an implementation would improve greatly compared to our proposed implementations that primarily use LUTs for the addition steps, and do not yet support pipelining. However, the latency of the implementation using this DSP block "chaining" is significantly increased, which can be a problem for latency-sensitive applications [28].

For comparison, we implemented a generator that uses cascaded DSP blocks, and does not use any LUTs for the partial-product summations. We configure and generate a multiplier module using the Xilinx Core-Generator tool which creates a 24 x 17 bit multiply and 48 bit accumulate structure using the A, B,

and C direct inputs of the block. The operand decomposition step is unchanged. Each digit-product, however, now represents a separate partial-product to be summed. We organize these according to increasing order of their least-significant bit positions. Each digit-product is then connected to the C input of the next DSP48E block that computes the next digit-product in the order described above. The final DSP block computes the final product. At each stage, some bits of the final product are finalized, which means that they do not need to be routed to the following DSP block. This prevents the required adder size from exceeding the available adder size in each DSP48E block. Figure 16 shows the 64 x 64 bit multiplier organization using the only the DSP48E blocks. The partial products are ordered according the start bit positions which is shown in Table 17.

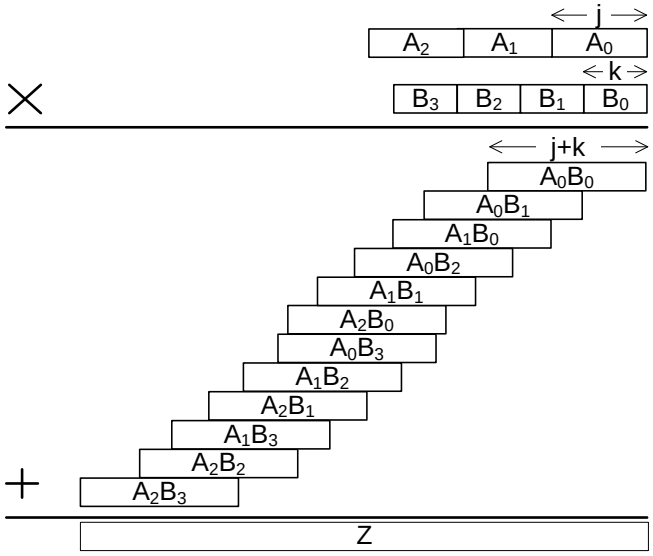


Figure 16: 64 x 64 multiplier implemented using DSP blocks only

Partial-Product	Start bit	End bit
A_0B_0	0	40
A_0B_1	17	57
A_1B_0	24	64
A_0B_2	34	74
A_1B_1	41	81
A_2B_0	48	88
A_0B_3	51	91
A_1B_2	58	98
A_2B_1	65	105
A_1B_3	75	115
A_2B_2	82	122
A_2B_3	99	139

Table 17: Organization of Partial-Products according to the increasing order of Start bit positions of the 64 x 64 bit multiplier shown in Figure 16

7 Results

Our generator program can target any asymmetric multiplier size, but to compare results of different multiplier design styles, we set the multiplier size parameters to match the asymmetric multipliers in the Xilinx Virtex-5 DSP48E blocks described in section 0. We currently only generate combinational structures, although the above techniques could be extended to incorporate pipeline stages. The best location for the pipeline stages would depend on the depth of the adder trees.

Our multiplier designs were synthesized on the Xilinx Virtex-5 XC5VLX155 (speed grade -2) device using the XST tool (version 10.1) with optimization goal set to speed and using normal optimization effort. The device contains 128 of the DSP48E blocks. In all designs, we describe additions using Verilog, and allow XST to choose whether to implement the additions in the DSP48E blocks or in LUT-based logic.

7.1 Operand Decomposition

We first test the case in the *operand decomposition* step where $C_{XN} \times C_{YM} = C_{XM} \times C_{YN}$, and demonstrate that it does in fact matter in that case which decomposition we choose. For this experiment, we implemented a 64 × 128 multiplier (X = 64, Y=128) using *Ripple Adder* and the *Outside-in Whole* (OIW) strategy. For n = 24, m = 17 we get $C_{XN} \times C_{YM} = 3 \times 8 = 24$ and $C_{XM} \times C_{YN} = 4 \times 6 = 24$. Our results in Table 18 confirm that we should pick to decompose X by m and Y by n, which we determine by finding that $C_{XN} + C_{YM} = 11$, and $C_{XM} + C_{YN} = 10$. We then choose the decomposition to be $C_{XM} \times C_{YN}$ as this would result in lesser number of partial product terms and hence, the required number of adders. From the table we observe that by choosing to decompose X by m improves combinational delay by 8.8%, and saves 9.2% of the LUTs.

Decomposition	DSPs	LUTs	Delay (ns)
64 by 24, 128 by 17 $C_{XN} = 3$, $C_{YM} = 8$	24	1161	11.978
64 by 17, 128 by 24 $C_{XM} = 4$, $C_{YN} = 6$	24	1054	10.919

Table 18: Comparing different possible decompositions when $C_{XN} \times C_{YM} = C_{XM} \times C_{YN}$ for a 64 × 128 multiplier.

7.2 Partial Product Generation

Next, we examined the different partial-product generation methods. We compared horizontal, vertical, and our chosen diagonal method for 64 × 64 multiplication. The results in Table 19 show that the diagonal regrouping of the terms helps to reduce the overall combinational delay 24.1% and helps in area savings by 33.23% on average compared to the horizontal and vertical groupings.

`Partial Product Generation	DSPs	LUTs	Delay (ns)
Horizontal	12	683	11.556
Vertical	12	597	12.319
Diagonal	12	456	9.350

Table 19: Comparison of different partial product generation methods for a 64 × 64 multiplier

7.3 Adder Tree Generation and Adder Types

We compared the different adder types and adder tree generation strategies (Delay Table-Whole (DW), Outside-in Whole (OIW), Delay Table-Top & Bottom (DTB) and Outside-in Top & Bottom (OITB)) using 640 different test cases (see tables in appendix). We varied the sizes and aspect ratios of the multipliers (by targeting different numbers of terms in the Upper, Lower and Middle regions. We also use two different adder types (Ripple Adder and Carry Vector Adder). In one set of cases, we the number of Upper and Lower region terms and varied the number of terms in the Middle region, and in another we fixed the number of terms in the Middle region and varied the number of terms in the Upper and Lower regions. We present the results for 96 x W bits multiplier where operand X is 96 bits and we vary W ranging from 68 to 221 bits. We compare each strategy on the basis of DSP48E count (Table 20), LUT count and combinational delay (Figure 17, Figure 18, Figure 19 and Figure 20). Delay Table-Whole (DW) results in the best area-oriented organization for the composed multipliers for both the Ripple and the Carry Vector Adders. The Delay Table strategy applies a greedy approach to repeatedly select the smallest adder possible for the remaining additions within each stage of the adder tree. On average, Outside-in Whole (OIW) results in the least combinational delay for both the Ripple and Carry Vector Adders. Because there are exception cases where one of the other strategies is either smaller or faster, further investigation is required to understand the way the synthesizer tools implement the additions specified by the generated Verilog code, which uses the '+' operator.

Multiplier (X x Y)	DSP48Es
96x68	16
96x85	20
96x102	24
96x119	28
96x136	32
96x153	36
96x170	40
96x187	44
96x204	48
96x221	52

Table 20: 96 x W multiplier DSP48E usage

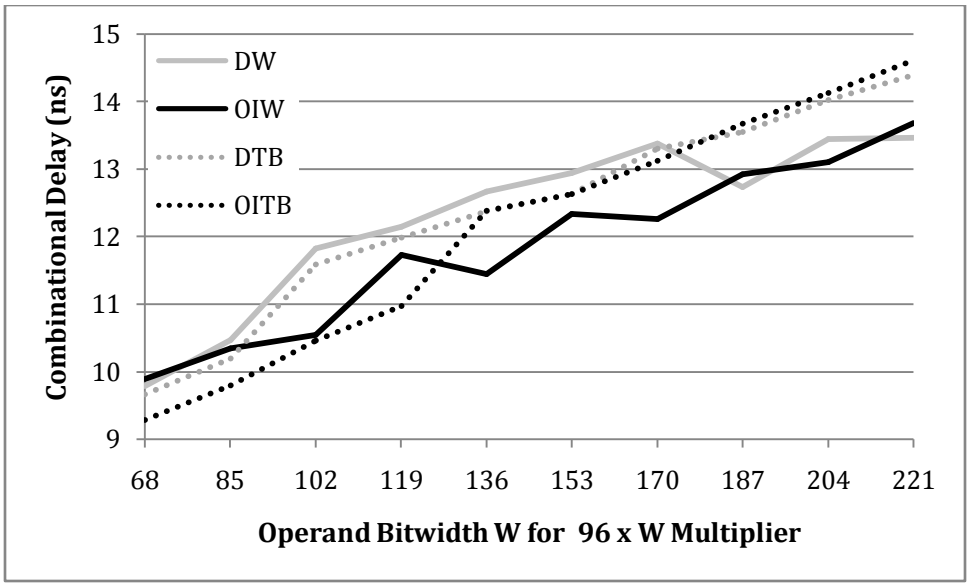


Figure 17: 96 x W multiplier Combinational Delay for Ripple Adder

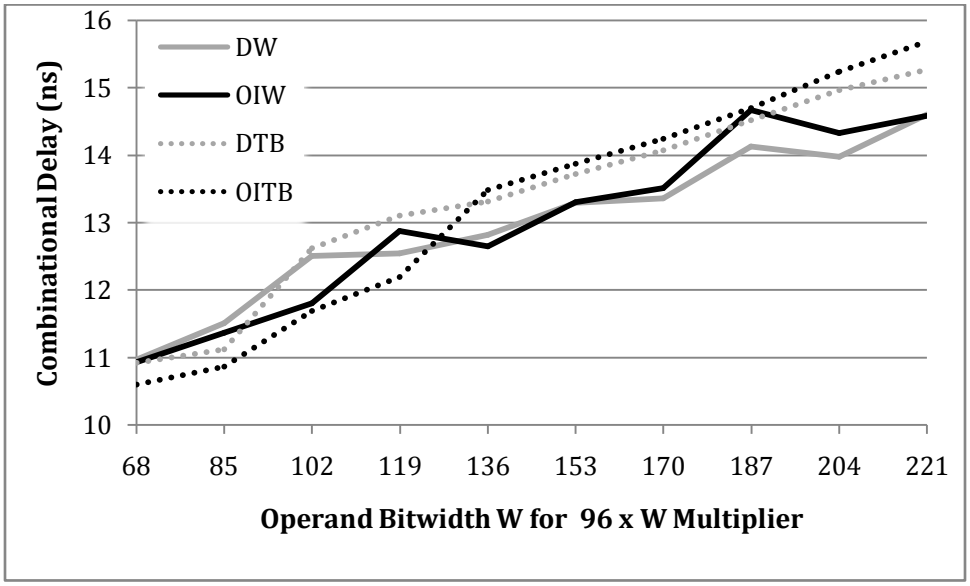


Figure 18: 96 x W multiplier Combinational Delay for Carry Vector Adder

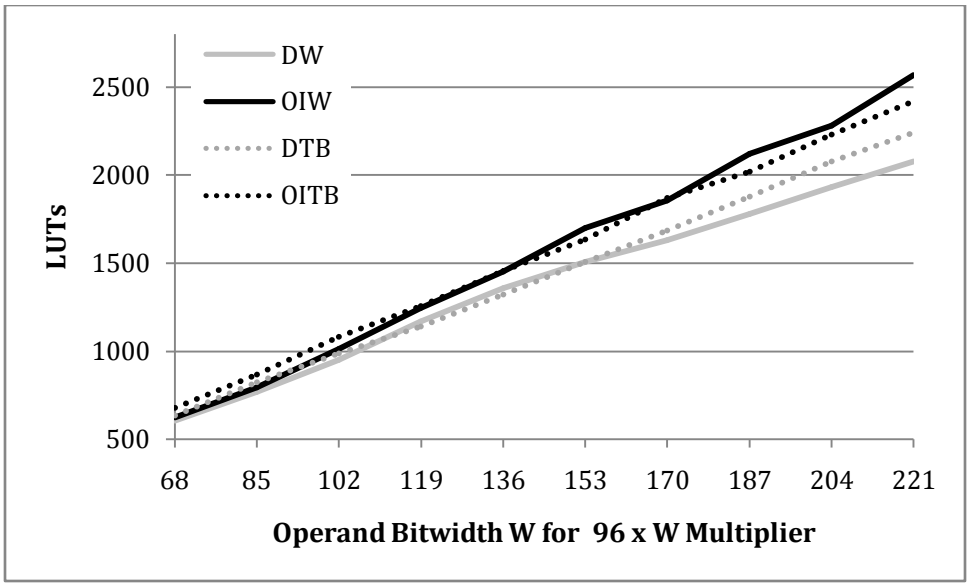


Figure 19: 96 x W multiplier LUT usage for Ripple Adder

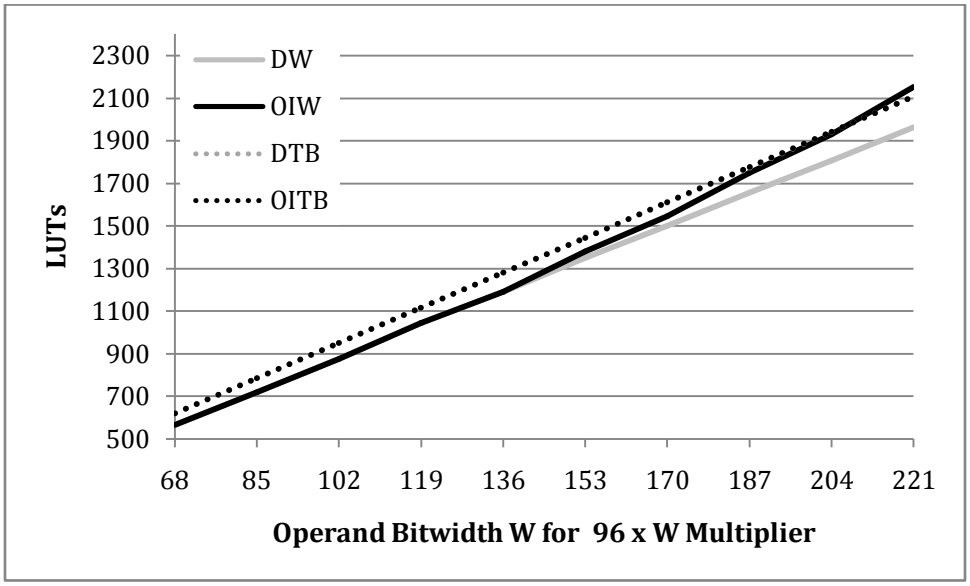


Figure 20: 96 x W multiplier LUT usage for Carry Vector Adder

Figure 21 and Figure 22 show the effects of using a Ripple Adder versus a Carry Vector Adder. Generally, the Ripple Carry Adder type results in a smaller delay. This is likely due to the increased routing congestion and additional adder level caused by the Carry Vector Adder, and the fact that the dedicated carry chain hardware in the FPGA makes the upper portion of the addition avoided by the CVA fairly fast in the RA. There is, however, some noticeable variation across bitwidths. However, the LUT count required for the RA vs. CVA (they have equal DSP48E usage) grows smoothly with increasing bitwidth, and the Carry Vector Adder structures create smaller multipliers. This is due to the fact that the CVAs concatenate non-overlapping upper sections of the addition, whereas the RAs must still propogate the carry through those positions (Figure 10 and Figure 11).

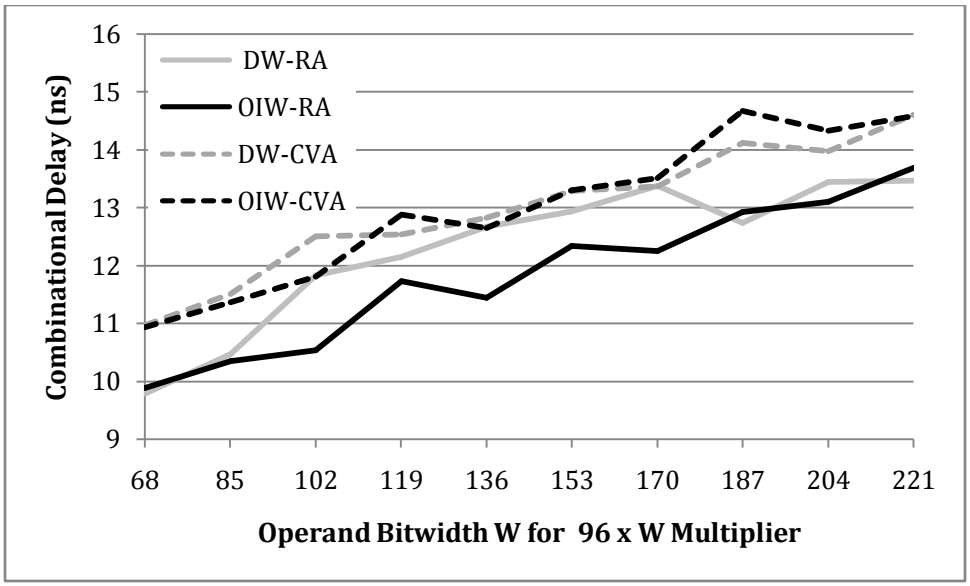


Figure 21: 96 x W multiplier Combinational Delay for DW and OIW (RA versus CVA)

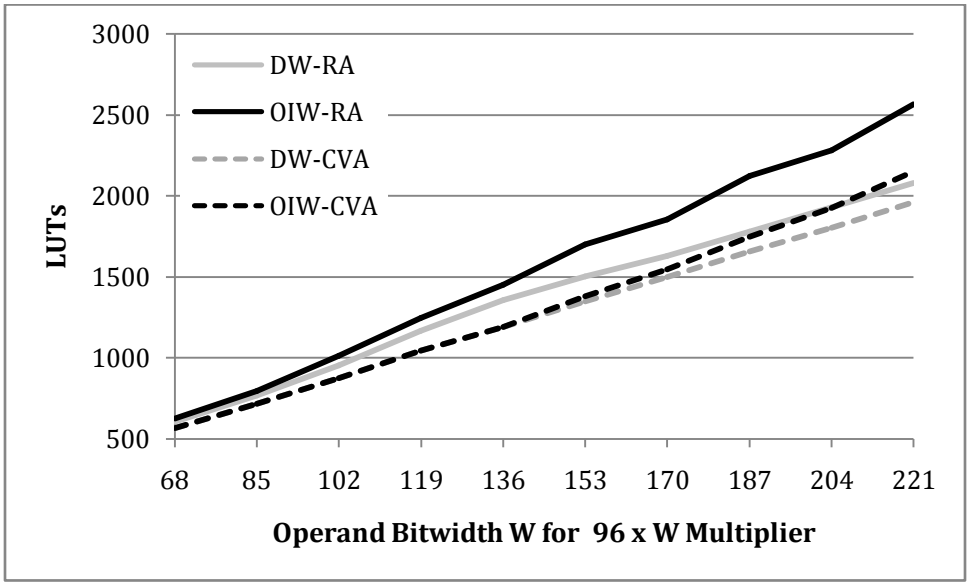


Figure 22: 96 x W multiplier Combinational Delay for DW and OIW (RA versus CVA)

7.4 Comparison to Other Composable Multipliers

We next compared our generated asymmetric multipliers to several different baselines. We aim to improve multiplier latency, which is important for many real-time applications [28]. Our approach with the best timing is the OIW strategy using Ripple Adders. For the following comparisons, we label this methodology as "Asym". First, we compare to the "Naïve" approach of just expressing the multiplication in Verilog as Z = X × Y. Second, we use Xilinx Core Generator (CoreGen) to create multipliers. Because Core Generator restricts generated multiplier sizes to 64 × 64 or smaller, these data points are only provided up to that point. Finally, we also implemented a symmetric multiplier generation method

(Sym) [14] that treats each 24 × 17 multiplier as a 17 × 17 multiplier, and uses the diagonal partial-product generation method described in section 5.2.The adder trees for the Sym multipliers were implemented using a strategy similar to the Outside-In Whole. Although this is clearly inefficient, the purpose of this particular baseline is to highlight the importance of considering asymmetry in multiplier generation.

We compare each of these multiplier methods on the basis of DSP48E count, LUT count, and combinational delay. All multipliers are generated as combinational-only designs. Future work that adds automated adder tree pipelining would also compare pipelined versions of these multipliers. We test a set of large multiplier sizes where both operands have equal width, ranging from 17 to 128 through (Figure 23, Figure 25 and Figure 27). We also test a set of large multiplier sizes where one operand is fixed at 64 bits, and the other varies from 17 to 128 through (Figure 24, Figure 26 and Figure 28).

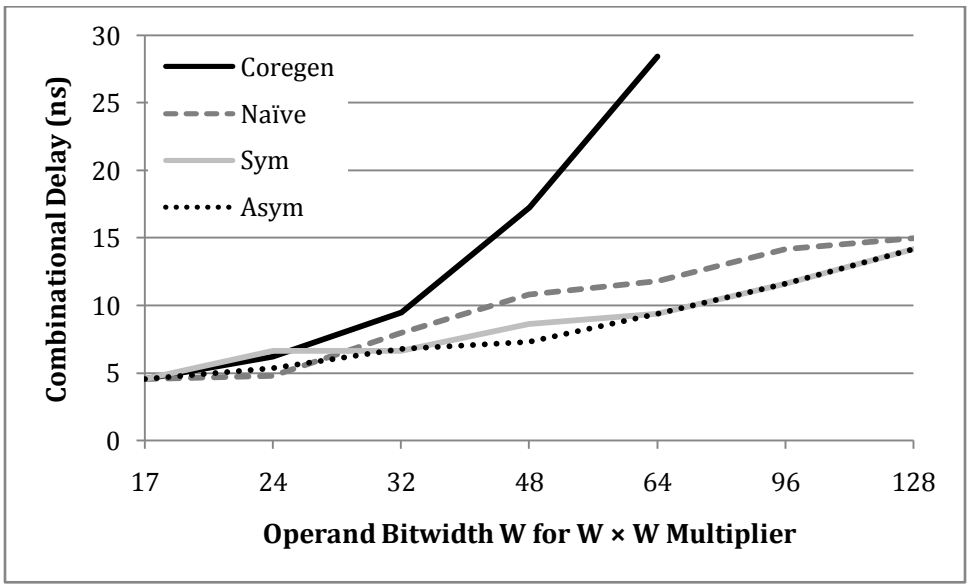


Figure 23: W × W multiplier combinational delay

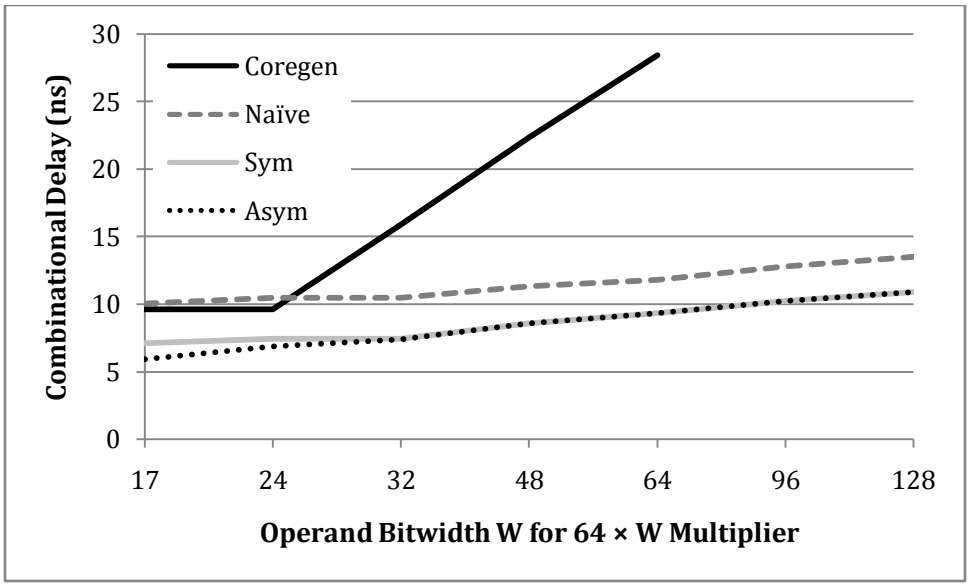


Figure 24: 64 × W multiplier combinational delay

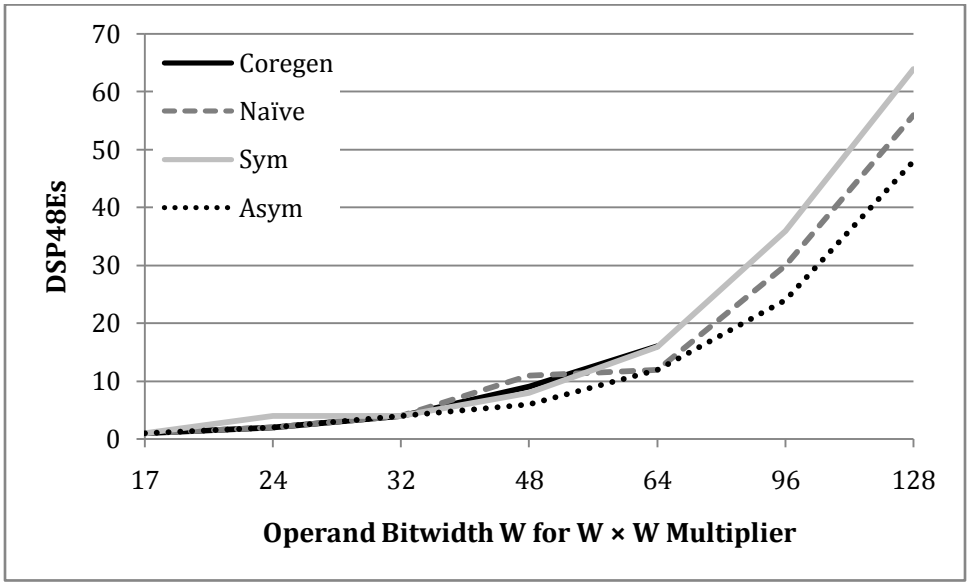


Figure 25: W × W multiplier DSP48E usage

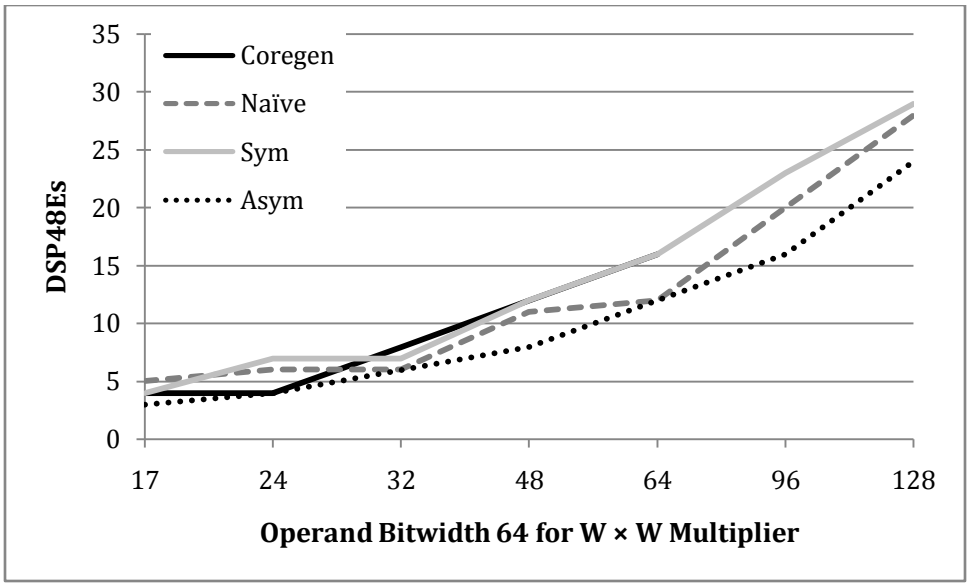


Figure 26: 64 × W multiplier DSP48E usage

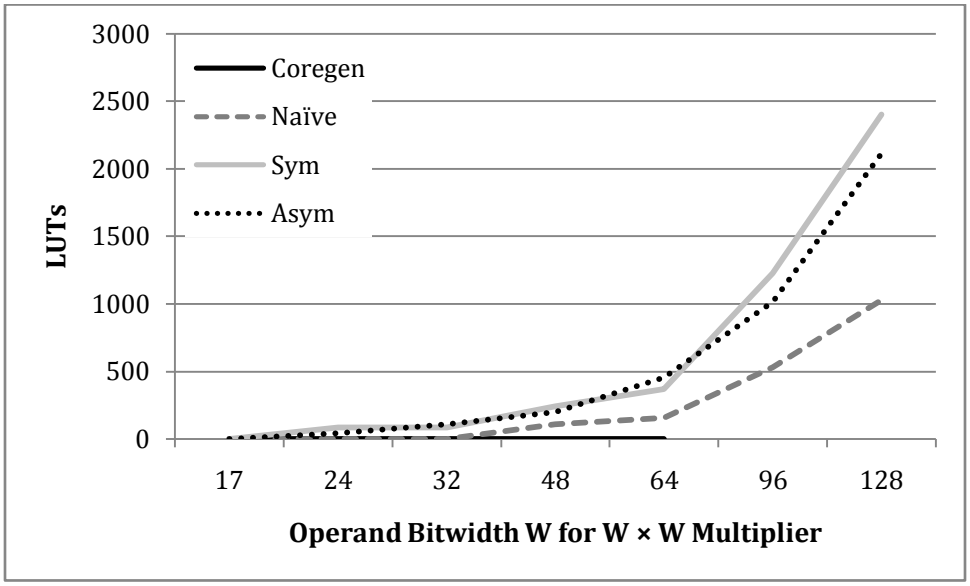


Figure 27: W × W multiplier LUT usage

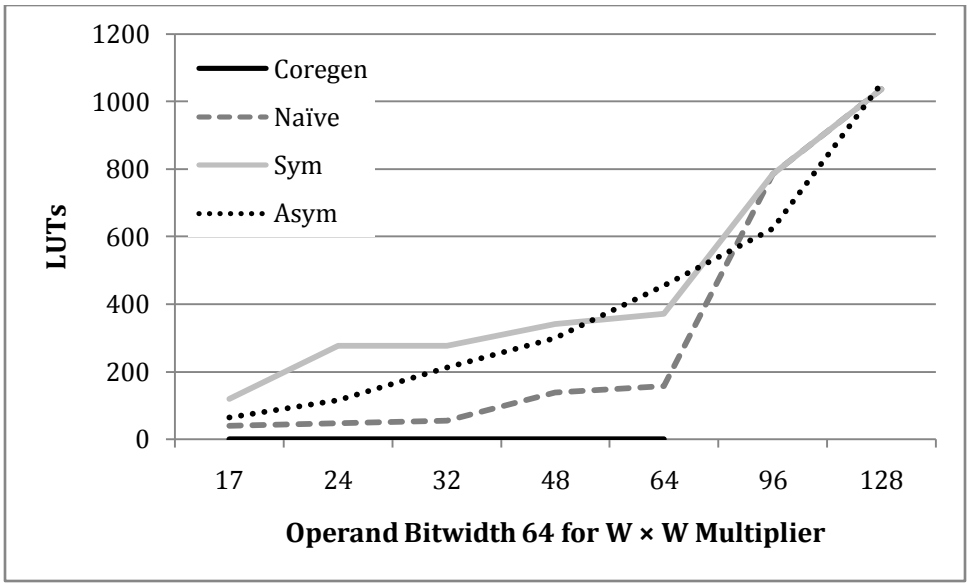


Figure 28: 64 × W multiplier LUT usage

The results show that for all tested multiplier sizes, our generation method (Asym) uses fewer DSP48E blocks than any of the compared methods. Among all multiplier designs, the Asymmetric multipliers use the minimum possible DSP block count. This is because our generator uses the full 24×17 multiplier size, and because we choose our operand decomposition specifically to minimize the DSP block count. The number of DSP blocks required by CoreGen is very similar to what is needed for the Symmetric (Sym) multipliers, indicating that CoreGen does not fully exploit the asymmetric multipliers. The Naïve results are also close to that of the CoreGen results in terms of DSP block use; it appears to use a similar methodology.

7.5 Cascaded (Non-Tree-Based) Partial Product Summation

Finally, we compare all multiplier designs that compute both digit-products and additions entirely in DSP blocks: Coregen and Asym-DSP (section 6). Again, we test a set of large multiplier sizes where both operands have equal width, ranging from 17 to 128 through (Figure 29 and Figure 31). We also test a set of large multiplier sizes where one operand is fixed at 64 bits, and the other varies from 17 to 128 through (Figure 30 and Figure 32). All the designs are tested on the basis of the DSP blocks count and the combinational delay. For comparison purposes we also include the original Asym results from the previous section.

The results show that the Asym-DSP method uses the minimum possible number of DSP48E blocks (when forcing all digit-products to be computed and summed using DSP blocks) for any given multiplier size as it utilizes the full 24x17 multiplier. CoreGen breaks down the multiplier using symmetric 17×17 multiplications for most of the partial products. This is because the DSP blocks provide dedicated routing between them that provides a fixed 17-bit shift between chained blocks, followed by a summation of the DSP block product with the shifted value which results in larger DSP48E counts comparatively. The only exception to the use of 17×17 multipliers is at the most significant digit-products that do not

require shifting—these can use the full multiplier capabilities, as given in the multiplier example from the DSP48E guide [7].

The combinational delay of Asym-DSP closely follows that of CoreGen. However, for the version of Coregen we used, multipliers with the operand size beyond 64 bits cannot be realized. Asym-DSP can realize multipliers implemented by using only DSP blocks for any arbitrary specified multiplier as desired by the user. This is very efficient in terms of LUT usage (no LUTs are required for the adders).

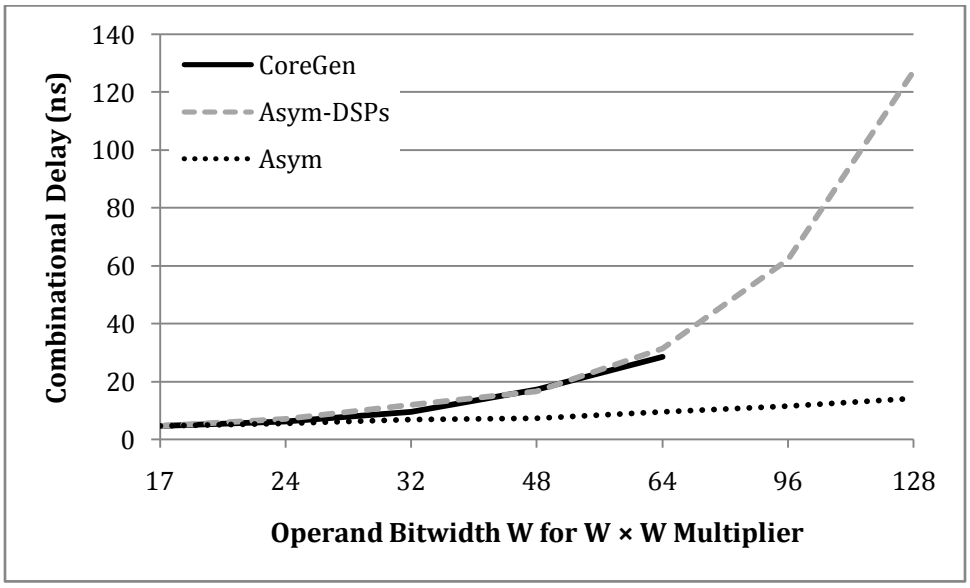


Figure 29: W × W multiplier combinational delay

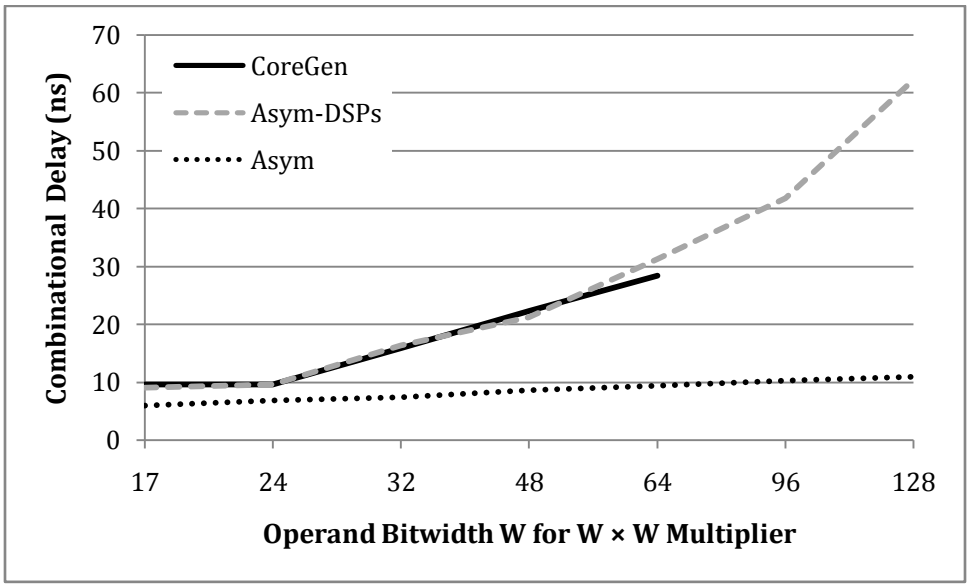


Figure 30: 64 × W multiplier combinational delay

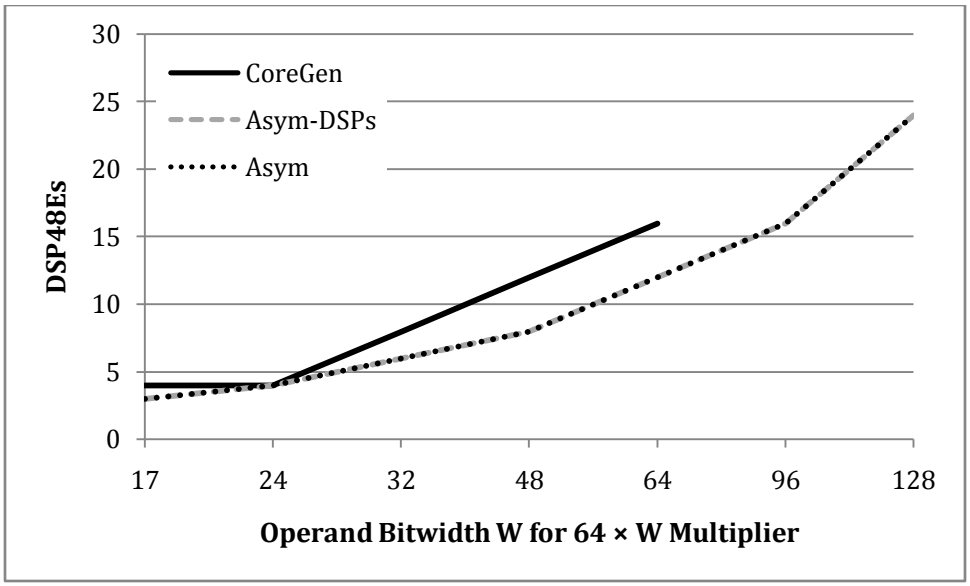


Figure 31: W × W multiplier DSP48E usage (Asym and Asym-DSPs have the same DSP usage)

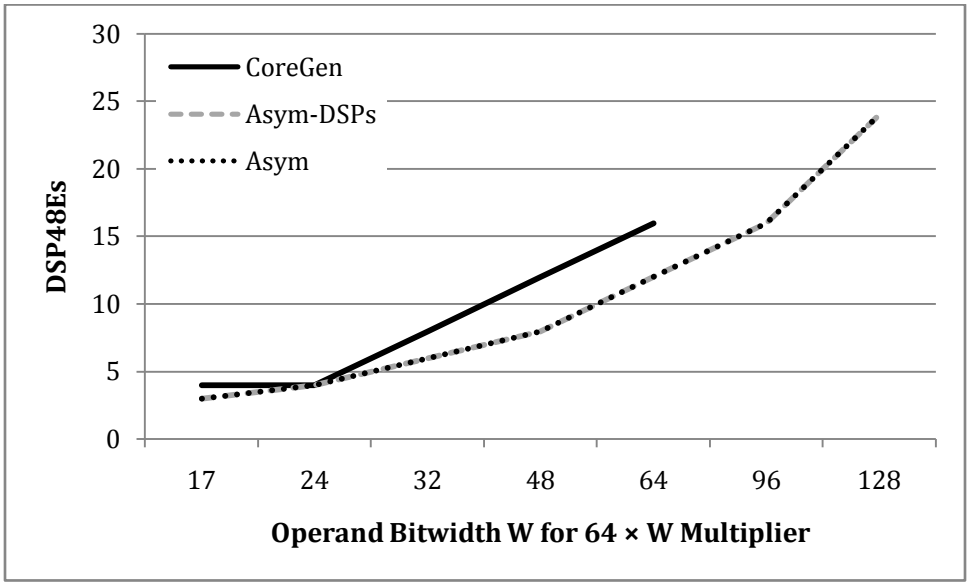


Figure 32: 64 × W multiplier DSP48E usage (Asym and Asym-DSPs have the same DSP usage)

As explained in section 6, the DSP48E block architecture includes only a fixed shift of 17 bits is supported in the dedicated routing paths between DSP blocks. This requires using routing external to the DSP blocks to accomplish the 24-bit shift required in some cases for the Asym-DSP method that uses the full 24 x 17 bit multiplier capability. This approach, although it uses fewer DSP48E blocks than CoreGen, significantly increases the latency. The external routing is much slower than the built-in chained routing of the DSP blocks (which is limited to only the 17-bit shift). This problem could be remedied if the DSP blocks were modified to provide a configurable 24-bit or 17-bit shift, but this would also increase the size and complexity of the DSP blocks. Furthermore, regardless of external vs. dedicated routing, the resulting chained addition would be much "deeper" than the tree-style addition

used in our Asymmetric multipliers. Thus, this solution, although area-efficient, may not be suitable when latency is an issue. It is likely to be a better solution to make use of a small amount of LUTs in our Asymmetric designs in exchange for greatly reduced latency and the faster (but less flexible) DSP blocks.

8 Future Work

The generated multipliers could use fewer multiplier blocks by implementing the digit-products for the uppermost digits in LUTs if those uppermost digits are incomplete (i.e., use far fewer bits than the multiplier block digit size). This could also be augmented to use non-standard tiling [18] and applying the partial-product rearrangement (concatenation) and summation tree techniques to the resulting partial products.

The partial-product summation stage can be further improved by considering the compressor tree mappings for the summation of the operands [21]. Currently, we choose the approach of just specifying the additions by using the Verilog "+" operator and let the vendor tools optimize the implementation. Improving the addition stages would further reduce the latency of the large multipliers.

The Karatsuba-Ofman algorithm [18] is a good choice in cases where the smaller multiplier is symmetric and reduces the overall number of DSP blocks required by trading the multiplications for additions. We believe a candidate solution for cases when the smaller multiplier is asymmetric would be to employ the Vedic multiplication algorithm [29][30]. The Vedic algorithm trades the multiplications for additions but reduces the overall multiplications compares to the Karatsuba-Ofman algorithm. The algorithm can also take into account of the binary ones and zeros present in the input operands and accordingly determine the number of steps for partitioning. Mapping a hybrid solution consisting of the existing divide-by-conquer approach and the Vedic algorithm for modern FPGAs is a work for future.

9 Conclusions

This thesis presented a new automated multiplier generator technique that creates large multipliers out of asymmetric embedded multiplier blocks, as are present in some of the newer commercial FPGAs. Designing a larger multiplier out of smaller multiplier building blocks is more complex for asymmetric than for symmetric multiplier blocks because there are two different shift factors involved (and various combinations of them), and partial products do not line up as exactly. We demonstrated that the decomposition of the two operands must be carefully approached, and that concatenating some of the partial products before they enter the adder tree for partial-product summation provides significant benefit.

Although our technique could be applied to any sized asymmetric blocks, we demonstrated its benefit by applying our method to the Xilinx Virtex-5 FPGA, which contains asymmetric hard multiplier cores. We explored a variety of adder generation strategies and addition types. We compared our generated multipliers with Naïve multipliers (using a single Verilog "*" operator to multiply the complete operands), multipliers created using Xilinx Core Generator, and multipliers created using a previous method designed for symmetric embedded multiplier blocks. In general, our generated multipliers using asymmetric blocks had a lower combinational delay. LUT count was equal to or lower than nearly all compared designs apart from the Core Generator version, which does not use any LUTs and solutions using carry-vector optimizations resulted in fewer LUTs. All the asymmetric multiplier designs used the same or fewer (usually fewer) DSP blocks than all other compared designs. However, as a percent of overall FPGA resources, LUT use of our multipliers is low compared to DSP block use. We demonstrated that a Ripple Adder is more effective for minimizing delay, but a Carry Vector Adder results in lower LUT usage. We also demonstrated that asymmetric multiplier designs can be implemented using only the DSP48E blocks (i.e., by using the adders included within them) is an area efficient solution but has a greatly increased latency. Our proposed approach of using DSP blocks to multiply asymmetric digits and LUT-based partial-product summation results in multipliers that are overall both smaller and lower-latency than those created using these other common techniques.

References

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput.Surv.*, vol. 34, pp. 171-210, 2002.

[2] R. Tessier, "Reconfigurable Computing for Digital Signal Processing: A Survey," *J. VLSI Signal Process.*, vol. 28, pp. 7-27, 2000.

[3] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays.* Springer Verlag, 2007,

[4] I. Kuon, R. Tessier and J. Rose, "Fpga architecture: Survey and challenges," *Foundations and Trends® in Electronic Design Automation,* vol. 2, pp. 135-253, 2008.

[5] Altera Corporation, "Stratix Device Family Data Sheet: Stratix Architecture, July," 2005.

[6] Xilinx Inc., "Virtex-II Platform FPGAs:Introduction and Overview, November," 2007.

[7] Xilinx Inc., "Virtex-5 FPGA XtremeDSP Design Considerations: User Guide, January," 2009.

[8] Xilinx Inc., "Virtex-6 FPGA DSP48E1 Slice: User Guide, September," 2009.

[9] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays,* 2006, pp. 21-30.

[10] B. Parhami, *Computer Arithmetic.* Oxford University Press, 2000,

[11] K. Israel, "Computer arithmetic algorithms," *AK Peters, Ltd,* 2002.

[12] S. Srinath and K. Compton, "Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* 2010, pp. 51-58.

[13] Altera Corporation, "Stratix-III Device Handbook," 2006.

[14] S. Gao, N. Chabini, D. Al-Khalili and P. Langlois, "Optimised realisations of large integer multipliers and squarers using embedded block," *Computers & Digital Techniques, IET,* vol. 1, pp. 9-16, 2007.

[15] J. L. Beauchat and A. Tisserand, "Small multiplier-based multiplication and divison operators," in *12th Conference on Field Programmable Logic and Applications,* 2002, pp. 513-522.

[16] B. Lee and N. Burgess, "Improved small multiplier based multiplication, squaring and division," in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on,* 2003, pp. 91-97.

[17] S. Gao, D. Al-Khalili and N. Chabini, "Efficient scheme for implementing large size signed multipliers using multigranular embedded DSP blocks in FPGAs," *International Journal of Reconfigurable Computing,* vol. 2009, pp. 1, 2009.

[18] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on,* 2009, pp. 250-255.

[19] C. S. Wallace, "A Suggestion for a Fast Multiplier," *Electronic Computers, IEEE Transactions on,* vol. EC-13, pp. 14-17, 1964.

[20] L. DADDAC, "SOME SCHEMES FOR PARALLEL MULTIPLIERS (*>," in *Computer Arithmetic,* 1980, pp. 118.

[21] H. Parandeh-Afshar, P. Brisk and P. Ienne, "Efficient synthesis of compressor trees on FPGAs," in *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific,* 2008, pp. 138-143.

[22] V. Oklobdzija, D. Villeger and S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Trans. Comput.,* vol. 45, pp. 294-306, 1996.

[23] W. C. Yeh and C. W. Jen, "High-speed booth encoded parallel multiplier design," *IEEE Trans. Comput.,* vol. 49, pp. 692-701, 2000.

[24] J. Kang and J. Gaudiot, "A Simple High-Speed Multiplier Design," *IEEE Trans. Comput.,* vol. 55, pp. 1253-1258, 2006.

[25] Z. Huang and M. D. Ercegovac, "High-performance left-to-right array multiplier design," in *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03),* 2003, pp. 4.

[26] Xilinx Inc., "XtremeDSP for Virtex-4 FPGAs User Guide (v2.7)," 2008.

[27] S. Gao, D. Al-Khalili and N. Chabini, "Optimized realization of large-size two's complement multipliers on FPGAs," *Organization,* vol. 10, pp. 0, 2007.

[28] A. Gregerson, A. Farmahini-Farahani, B. Buchli, S. Naumov, M. Bachtis, K. Compton, M. Schulte, W. H. Smith and S. Dasu, "FPGA design analysis of the clustering algorithm for the CERN large hadron collider," in *Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines,* 2009, pp. 19-26.

[29] H. Thapliyal and M. Srinivas, "High speed efficient N X N bit parallel hierarchical overlay multiplier architecture based on ancient indian vedic mathematics," in *ICSP 2004: International Conference on Signal Processing,* 2004,

[30] H. Thapliyal and M. Srinivas, "VLSI implementation of RSA encryption system using ancient Indian Vedic mathematics," *Arxiv Preprint cs/0609028,* 2006.

Appendix

Table 21, Table 22, Table 23, Table 24 and Table 25 show the tests cases used to compare the four adder generation strategies for both the addition types for our generated multipliers using the asymmetric 24 × 17 multipliers in the Virtex-5.

Table 21: DSP48E usage for different multiplier sizes

Multiplier (X x Y)	DSPs
48x34	4
48x51	6
48x68	8
48x85	10
48x102	12
48x119	14
48x136	16
48x153	18
48x170	20
48x187	22

Multiplier (X x Y)	DSPs
72x51	9
72x68	12
72x85	15
72x102	18
72x119	21
72x136	24
72x153	27
72x170	30
72x187	33
72x204	36

Multiplier (X x Y)	DSPs
96x68	16
96x85	20
96x102	24
96x119	28
96x136	32
96x153	36
96x170	40
96x187	44
96x204	48
96x221	52

Multiplier (X x Y)	DSPs
120x85	25
120x102	30
120x119	35
120x136	40
120x153	45
120x170	50
120x187	55
120x204	60
120x221	65
120x238	70

Multiplier (X x Y)	DSPs
48x34	4
72x51	9
96x68	16
120x85	25
144x102	36
168x119	49
192x136	64
216x153	81

Multiplier (X x Y)	DSPs
48x51	6
72x68	12
96x85	20
120x102	30
144x119	42
168x136	56
192x153	72
216x170	90

Multiplier (X x Y)	DSPs
48x68	8
72x85	15
96x102	24
120x119	35
144x136	48
168x153	63
192x170	80

Multiplier (X x Y)	DSPs
48x85	10
72x102	18
96x119	28
120x136	40
144x153	54
168x170	70
192x187	88

Table 22: Combinational Delay (ns) for Ripple Adder Designs

Method	48x34	48x51	48x68	48x85	48x 102	48x 119	48x 136	48x 153	48x 170	48x 187
DW	6.622	7.286	8.975	9.160	9.275	9.265	9.831	10.671	11.164	11.344
OIW	6.622	7.286	8.138	8.738	8.728	9.319	9.536	10.710	10.814	11.058
DTB	6.810	7.100	8.664	8.848	9.494	9.941	11.138	11.535	12.102	12.474
OITB	6.810	7.100	8.749	9.005	9.640	10.109	10.636	10.969	11.827	12.227

Method	72x51	72x68	72x85	72x 102	72x 119	72x 136	72x 153	72x 170	72x 187	72x 204
DW	8.791	8.979	9.826	10.235	11.632	10.836	11.266	11.368	12.267	12.176
OIW	8.286	9.425	9.351	9.728	10.089	11.350	11.409	11.760	11.774	12.207
DTB	8.392	8.716	9.704	10.146	11.511	11.819	12.362	12.900	13.116	13.658
OITB	8.365	8.802	9.513	10.006	10.892	11.347	12.391	12.709	12.749	13.184

Method	96x68	96x85	96x 102	96x 119	96x 136	96x 153	96x 170	96x 187	96x 204	96x 221
DW	9.782	10.466	11.830	12.151	12.673	12.940	13.379	12.735	13.446	13.469
OIW	9.887	10.349	10.541	11.728	11.444	12.336	12.257	12.930	13.106	13.686
DTB	9.663	10.198	11.590	11.990	12.371	12.629	13.308	13.555	14.027	14.393
OITB	9.286	9.800	10.461	10.971	12.381	12.634	13.129	13.674	14.135	14.615

Method	120x 85	120x 102	120x 119	120x 136	120x 153	120x 170	120x 187	120x 204	120x221	120x238
DW	11.754	11.996	12.383	13.018	13.299	13.926	14.348	14.766	15.213	15.889
OIW	10.662	12.458	12.033	12.677	12.687	13.429	13.352	13.901	14.424	15.558
DTB	11.183	12.117	12.082	12.399	13.449	14.074	14.206	14.992	15.637	15.862
OITB	10.317	10.893	12.080	12.452	12.786	13.208	14.183	14.583	14.993	15.380

Method	48x34	72x51	96x68	120x85	144x102	168x119	192x136	216x153
DW	6.622	8.791	9.782	11.754	12.286	13.627	15.322	16.814
OIW	6.622	8.286	9.887	10.662	12.248	13.593	14.727	15.632
DTB	6.810	8.392	9.663	11.183	12.489	13.219	14.719	16.437
OITB	6.810	8.365	9.286	10.317	11.756	12.764	13.983	15.392

Method	48x51	72x68	96x85	120x102	144x119	168x136	192x153	216x170
DW	7.286	8.979	10.466	11.996	12.696	14.065	15.219	17.342
OIW	7.286	9.425	10.349	12.458	13.217	14.398	15.437	17.434
DTB	7.100	8.716	10.189	12.117	12.542	13.857	15.117	17.441
OITB	7.100	8.802	9.800	10.893	12.218	13.427	14.934	16.065

Method	48x68	72x85	96x102	120x119	144x136	168x153	192x170
DW	8.975	8.946	11.830	12.383	13.814	14.698	16.710
OIW	8.138	9.351	10.541	12.033	13.286	14.410	15.716
DTB	8.644	9.704	11.590	12.082	13.518	14.669	16.429
OITB	8.749	9.513	10.461	12.080	12.884	14.170	15.577

Method	48x85	72x102	96x119	120x136	144x153	168x170	192x187
DW	9.160	10.235	12.151	13.018	14.041	15.084	17.514
OIW	8.738	9.728	11.728	12.677	13.765	14.823	16.972
DTB	8.848	10.146	11.990	12.399	13.940	15.085	16.992
OITB	9.005	10.006	10.971	12.452	13.385	14.837	16.024

Table 23: Combinational Delay (ns) for Carry Vector Adder Designs

Method	48x34	48x51	48x68	48x85	48x 102	48x 119	48x 136	48x 153	48x 170	48x 187
DW	7.728	8.413	9.647	9.343	9.829	10.421	11.005	11.499	11.446	12.218
OIW	7.728	8.413	9.215	10.057	9.891	10.397	10.606	11.763	11.832	12.006
DTB	7.802	8.189	9.638	10.100	10.694	11.087	11.723	12.274	13.047	13.576
OITB	7.802	8.189	9.629	10.026	10.694	11.087	11.723	12.274	13.047	13.576

Method	72x51	72x68	72x85	72x 102	72x 119	72x 136	72x 153	72x 170	72x 187	72x 204
DW	9.935	10.053	10.511	10.846	11.528	12.311	12.285	12.820	12.876	13.287
OIW	9.488	10.517	10.394	10.905	11.046	12.687	12.401	12.858	12.795	13.467
DTB	9.488	9.719	10.856	11.143	12.312	12.734	13.380	13.882	13.850	14.172
OITB	9.361	9.761	10.667	11.072	12.170	12.477	13.380	13.882	13.850	14.172

Method	96x68	96x85	96x 102	96x 119	96x 136	96x 153	96x 170	96x 187	96x 204	96x 221
DW	10.967	11.513	12.506	12.542	12.824	13.295	13.366	14.126	13.974	14.607
OIW	10.934	11.372	11.811	12.883	12.654	13.304	13.510	14.672	14.332	14.583
DTB	10.924	11.122	12.621	13.107	13.310	13.721	14.077	14.517	14.966	15.273
OITB	10.599	10.871	11.693	12.196	13.481	13.874	14.249	14.703	15.241	15.686

Method	120x 85	120x 102	120x 119	120x 136	120x 153	120x 170	120x 187	120x 204	120x221	120x238
DW	12.974	13.055	13.072	13.451	13.587	14.348	14.300	15.057	15.474	16.542
OIW	11.984	13.424	13.081	13.940	13.849	14.299	14.443	15.451	15.344	16.642
DTB	12.403	12.713	13.255	13.776	14.240	14.893	15.483	16.017	16.350	16.918
OITB	11.781	12.270	13.182	13.850	14.053	14.607	15.429	15.930	16.271	16.662

Method	48x34	72x51	96x68	120x85	144x102	168x119	192x136	216x153
DW	7.728	9.935	10.967	12.974	13.485	14.788	15.777	17.790
OIW	7.728	9.488	10.934	11.984	13.280	14.683	15.803	17.050
DTB	7.802	9.488	10.924	12.403	13.100	14.547	15.743	17.541
OITB	7.802	9.361	10.599	11.781	12.820	14.544	15.259	16.562

Method	48x51	72x68	96x85	120x102	144x119	168x136	192x153	216x170
DW	8.413	10.053	11.513	13.055	13.893	14.960	16.294	18.625
OIW	8.413	10.517	11.372	13.424	14.339	15.391	16.323	18.802
DTB	8.189	9.719	11.122	12.713	13.782	14.685	16.388	18.219
OITB	8.189	9.761	10.871	12.270	13.660	14.376	15.514	17.200

Method	48x68	72x85	96x102	120x119	144x136	168x153	192x170
DW	9.647	10.511	12.506	13.072	14.364	15.391	17.403
OIW	9.215	10.394	11.811	13.081	14.293	15.340	16.717
DTB	9.638	10.856	12.621	13.255	14.685	15.709	17.631
OITB	9.629	10.667	11.693	13.182	13.962	15.184	16.593

Method	48x85	72x102	96x119	120x136	144x153	168x170	192x187
DW	9.343	10.846	12.542	13.451	14.425	15.749	18.088
OIW	10.057	10.905	12.883	13.940	14.853	16.063	18.617
DTB	10.100	11.143	13.107	13.776	15.072	16.475	18.365
OITB	10.026	11.072	12.196	13.850	14.382	15.661	17.302

Table 24: LUT usage for Ripple Adder designs

Method	48x34	48x51	48x68	48x85	48x 102	48x 119	48x 136	48x 153	48x 170	48x 187
DW	107	199	283	350	425	486	545	624	690	798
OIW	107	199	299	375	516	569	737	804	881	930
DTB	123	206	289	371	471	581	682	799	849	940
OITB	123	206	307	389	489	599	735	852	884	976

Method	72x51	72x68	72x85	72x 102	72x 119	72x 136	72x 153	72x 170	72x 187	72x 204
DW	305	434	592	624	859	973	1085	1204	1288	1407
OIW	320	456	613	804	923	1115	1252	1460	1598	1823
DTB	330	454	596	799	860	991	1133	1264	1405	1587
OITB	346	494	634	852	934	1059	1218	1371	1563	1738

Method	96x68	96x85	96x 102	96x 119	96x 136	96x 153	96x 170	96x 187	96x 204	96x 221
DW	606	770	952	1168	1357	1505	1629	1781	1932	2079
OIW	624	794	1013	1246	1451	1701	1855	2122	2282	2567
DTB	635	824	988	1138	1321	1505	1686	1878	2079	2242
OITB	677	865	1083	1255	1454	1635	1869	2018	2230	2418

Method	120x 85	120x 102	120x 119	120x 136	120x 153	120x 170	120x187	120x204	120x221	120x238
DW	985	1171	1425	1696	1960	2224	2428	2589	2752	2933
OIW	1048	1259	1502	1795	2100	2368	2683	2926	3251	3469
DTB	1032	1238	1462	1684	1906	2109	2352	2572	2794	3036
OITB	1120	1401	1641	1830	2106	2328	2584	2798	3126	3333

Method	48x34	72x51	96x68	120x85	144x102	168x119	192x136	216x153
DW	107	305	606	985	1465	1995	2657	3311
OIW	107	320	624	1048	1547	2128	2822	3670
DTB	123	330	635	1032	1500	2078	2752	3505
OITB	123	346	677	1120	1681	2335	3098	3987

Method	48x51	72x68	96x85	120x102	144x119	168x136	192x153	216x170
DW	199	434	770	1171	1677	2275	2985	3740
OIW	199	456	794	1259	1784	2441	3159	4047
DTB	206	454	824	1238	1773	2389	3130	3875
OITB	206	494	865	1401	1976	2719	3498	4527

Method	48x68	72x85	96x102	120x119	144x136	168x153	192x170
DW	283	592	952	1425	1952	2593	3284
OIW	299	613	1013	1502	2114	2788	3598
DTB	289	596	988	1462	2036	2714	3458
OITB	307	634	1083	1641	2292	3057	3950

Method	48x85	72x102	96x119	120x136	144x153	168x170	192x187
DW	350	624	1168	1696	2293	2979	3694
OIW	375	804	1246	1795	2466	3189	4040
DTB	371	799	1138	1684	2282	3018	3758
OITB	389	852	1255	1830	2569	3350	4341

Table 25: LUT usage for Carry Vector Adder designs

Method	48x34	48x51	48x68	48x85	48x 102	48x 119	48x 136	48x 153	48x 170	48x 187
DW	107	184	265	335	402	471	539	607	673	757
OIW	107	184	265	335	433	520	634	739	787	873
DTB	124	208	291	374	457	550	631	730	797	881
OITB	124	208	291	374	457	550	631	730	797	881

Method	72x51	72x68	72x85	72x 102	72x 119	72x 136	72x 153	72x 170	72x 187	72x 204
DW	298	405	532	651	775	890	1013	1127	1251	1369
OIW	298	405	532	651	775	909	1052	1204	1363	1530
DTB	331	456	580	704	827	951	1074	1199	1324	1466
OITB	331	456	580	704	827	951	1074	1199	1324	1466

Method	96x68	96x85	96x 102	96x 119	96x 136	96x 153	96x 170	96x 187	96x 204	96x 221
DW	566	718	876	1044	1191	1350	1500	1656	1805	1961
OIW	566	718	876	1044	1191	1379	1545	1748	1928	2151
DTB	620	786	950	1115	1281	1445	1608	1774	1941	2106
OITB	620	786	950	1115	1280	1444	1610	1776	1941	2106

Method	120x 85	120x 102	120x 119	120x 136	120x 153	120x 170	120x187	120x204	120x221	120x238
DW	924	1110	1308	1507	1718	1920	2128	2324	2536	2736
OIW	924	1110	1308	1507	1718	1920	2128	2360	2606	2841
DTB	990	1195	1401	1606	1813	2020	2225	2430	2637	2844
OITB	990	1195	1401	1606	1813	2020	2225	2430	2636	2843

Method	48x34	72x51	96x68	120x85	144x102	168x119	192x136	216x153
DW	107	298	566	924	1357	1854	2451	3146
OIW	107	298	566	924	1357	1854	2451	3146
DTB	124	331	620	990	1442	1977	2594	3292
OITB	124	331	620	990	1442	1977	2594	3292

Method	48x51	72x68	96x85	120x102	144x119	168x136	192x153	216x170
DW	199	434	770	1171	1590	2139	2786	3502
OIW	199	456	794	1259	1593	2143	2786	3502
DTB	206	454	824	1238	1690	2266	2924	3661
OITB	206	494	865	1401	1690	2266	2924	3661

Method	48x68	72x85	96x102	120x119	144x136	168x153	192x170
DW	265	532	876	1308	1811	2416	3094
OIW	265	532	876	1308	1811	2416	3094
DTB	291	580	950	1401	1937	2554	3252
OITB	291	580	950	1401	1937	2554	3252

Method	48x85	72x102	96x119	120x136	144x153	168x170	192x187
DW	335	651	1044	1507	2062	2692	3412
OIW	335	651	1044	1507	2062	2692	3412
DTB	374	704	1115	1606	2184	2842	3581
OITB	374	704	1115	1606	2184	2842	3581