

# Least-Significant Bit Optimization Techniques for FPGAs

Mark L. Chang and Scott Hauck  
Department of Electrical Engineering  
University of Washington  
Seattle, Washington  
Email: {mchang,hauck}@ee.washington.edu

**Abstract**—In this paper we present a methodology for FPGA datapath precision optimization subject to user-defined area and error constraints. This work builds upon our previous research [1] which presented a methodology for optimizing for dynamic range—the most significant bit position. In this work, we derive area and error models of a general island-style FPGA architecture in order to optimize the least-significant bit position of circuit datapaths. We present some preliminary results describing the effectiveness of our techniques on typical signal and image processing kernels.

## I. INTRODUCTION

With the widespread growth of reconfigurable computing platforms in education, research, and industry, more software developers are being exposed to hardware development. Many are seeking to achieve the enormous gains in performance demonstrated in the research community by implementing their software algorithms in a reconfigurable fabric. For the novice hardware designer, this effort usually begins and ends with futility and frustration as they struggle with unwieldy tools and new programming paradigms.

One of the more difficult paradigm shifts to grasp is the notion of bit-level operations. On a typical FPGA fabric, logical and arithmetic operators can work at the bit level instead of the word level. With careful optimization of the precision of the datapath, the overall size and relative speed of the resulting circuit can be dramatically improved.

In this paper we present a methodology that broadens the work presented in [1]. We begin with a more detailed background of precision analysis and previous research efforts. We describe the problem of least-significant bit optimization and propose several optimization techniques that provide finer control of area-to-error tradeoffs than more traditional methods. Finally, we show examples utilizing our techniques to optimize the datapath of image processing circuits and draw some conclusions.

## II. BACKGROUND

General-purpose processors are designed to perform operations at the word level, typically 8, 16, or 32 bits. Supporting this paradigm, programming languages and compilers abstract these word sizes into storage classes, or data-types, such as `char`, `int`, and `float`. In contrast, most mainstream reconfigurable logic devices, such as FPGAs, operate at the bit level. This allows the developer to tune datapaths to any

word size desired. Unfortunately, choosing the appropriate size for datapaths is not trivial. Choosing a wide datapath, as in a general-purpose processor, usually results in an implementation that is larger than necessary. This consumes valuable resources and potentially reduces the performance of the design. On the other hand, if the hardware implementation uses too little precision, errors can be introduced at runtime through quantization effects, such as roundoff and truncation.

To alleviate the programmer’s burden of doing manual precision analysis, researchers have proposed many different solutions. Techniques range from semi-automatic to fully-automated methods that employ static and dynamic analysis of circuit datapaths.

### A. The Least-Significant Bit Problem

In determining the fixed-point representation of a floating-point datapath, we must consider both the most-significant and least-significant ends. Reducing the relative bit position of the most-significant bit reduces the maximum value that the datapath may represent, sometimes referred to as the dynamic range. On the other end, increasing the relative bit position of the least-significant bit (toward the most-significant end) reduces the maximum precision that the datapath may attain. For example, if the most-significant bit is at the  $2^7$  position, and the least-significant bit is at the  $2^{-3}$  position, the maximum value attainable by an unsigned number will be  $2^8 - 1 = 255$ , while the precision will be quantized to multiples of  $2^{-3} = 0.125$ . Values smaller than 0.125 cannot be represented as the bits necessary to represent, for example, 0.0625, do not exist.

Having a fixed-point datapath means that results or operations may exhibit some quantity of error compared to their infinite-precision counterparts. This quantization error can be introduced in both the most-significant and least-significant sides of the datapath. If the value of an operation is larger than the maximum value that can be represented by the datapath, the quantization error is typically a result of truncation or saturation, depending on the implementation of the operation. Likewise, error is accumulated at the least-significant end of the datapath if the value requires greater precision than the datapath can represent, resulting in truncation or round-off error.

Previous research includes [2], [3], which only performs the analysis on the most-significant bit position of the datapath. While this method achieves good results, it ignores the potential optimization of the least-significant bit position. Other research, including [4], [5] begin to touch on fixed-point integer representations of numbers with fractional portions. Finally, more recent research, [6], [7] begin to incorporate error analysis into the overall analysis of the fractional part of the datapath elements.

Most of the techniques introduced deal with either limited scope of problem, such as linear time-invariant (LTI) systems, and/or perform the analysis completely automatically, with minimal input from the developer. While again, these methods achieve good results, it is our belief that the developer should be kept close at hand during all design phases, as there are some things for which an automatic optimization method simply cannot handle.

Simply put, a “goodness” metric must be devised in order to guide an automatic precision optimization tool. This “goodness” function is then evaluated by the automated tool to guide its precision optimization. In some cases, such as image processing, a simple block signal-to-noise ratio (BSNR) may be appropriate. In many cases, though, this metric is difficult or impossible to evaluate programmatically. A human developer, therefore, has the benefit of having a much greater sense of context in evaluating what is an appropriate tradeoff between error in the output and performance of the implementation. We have used this idea as the guiding principle behind the design of our precision analysis tool Précis [1]. In this paper we provide the metrics and methodology for performing least-significant-bit optimization.

The observation that the relative bit position of the least-significant bit introduces a quantifiable amount of error over an infinite-precision datapath is an important one. After performing the optimization for the most-significant bit position as described in [1], we must perform another area/error analysis phase to optimize the position of the least-significant bit. We begin our discussion by introducing models for area and error of a general island-style FPGA.

### III. ERROR MODELS

Consider an integer value that is  $M'$  bits in length. This value has an implicit binary point at the far right—to the right of the least-significant bit position. By truncating bits from the least-significant side of the word, we reduce the area impact of this word on downstream arithmetic and logic operations. It is common practice to simply truncate the bits from the least-significant side to reduce the number of bits required to store and operate on this word. We propose an alternate method—replace the bits that would normally be truncated with zeros instead. Therefore, for an  $M'$ -bit value, we have the notation  $A_m 0_p$ . This is word that has  $m$  correct bits and  $p$  zeros inserted to signify bits that have been effectively truncated, giving us an  $M' = m + p$ -bit word.

Having performed a reduction in the precision that can be obtained by this datapath with a substitution of zeros, we have

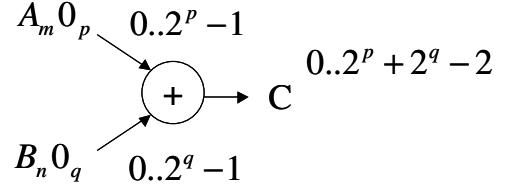


Fig. 1. Error model of an adder.

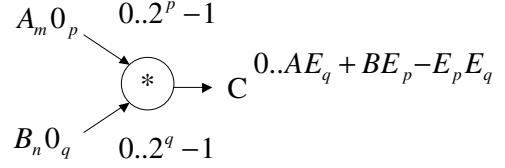


Fig. 2. Error model of a multiplier.

introduced a quantifiable amount of error into the datapath. For an  $A_m 0_p$  value, substituting  $p$  zeros for the lower portion of the word, gives us an error range of  $0..2^p - 1$ . Meaning at best, if the bits replaced by zeros were originally zeros, we have incurred no error. At worst, if the bits replaced were originally ones, we have incurred maximum error,  $2^p - 1$ .

This error model can be used to determine the effective error of combining quantized values in arithmetic operators. To investigate the impact, we will discuss an adder and multiplier in greater detail.

#### A. Adder Error Model

An adder error model is shown in Fig. 1. The addition of two quantized values,  $A_m 0_p + B_n 0_q$ , results in an output,  $C$ , which has a total of  $\max(M', N') + 1$  bits, where  $\min(p, q)$  of them are substituted zeros at the least-significant end. Perhaps more importantly, the range of error for  $C$  is the sum of the error ranges of  $A$  and  $B$ . This gives us an error range of  $0..2^p + 2^q - 2$  at the output of the adder.

#### B. Multiplier Error Model

Just as we can derive an error model for the adder, we do the same for a multiplier. Again we have two quantized input values,  $A_m 0_p * B_n 0_q$ , multiplied together to form the output,  $C$ , which has a total of  $M' + N'$  bits, where  $p + q$  of them are substitute zeros at the least-significant end. This structure is shown in Fig. 2.

The output error is slightly more complex in the multiplier structure than the adder structure. The input error ranges are the same,  $0..2^p - 1$  and  $0..2^q - 1$  for  $A_m 0_p$  and  $B_n 0_q$ , respectively. Unlike the adder, multiplying these two inputs together requires us to multiply the error terms as well, as shown in (1).

$$\begin{aligned}
 C &= A * B \\
 &= (A - (2^p - 1)) * (B - (2^q - 1)) \\
 &= AB - B(2^p - 1) - A(2^q - 1) + (2^p - 1)(2^q - 1)
 \end{aligned} \tag{1}$$

The first line of (1) indicates the desired multiplication operation between the two input signals. Since we are introducing errors into each signal, line two shows the impact of the error range of  $A_m 0_p$  by subtracting  $2^p - 1$  from the error-free input  $A$ . The same is done for input  $B$ .

Performing a substitution of  $E_p = 2^p - 1$  and  $E_q = 2^q - 1$  into (1) yields the simpler (2):

$$\begin{aligned} C &= AB - BE_p - AE_q + E_p E_q \\ &= AB - (AE_q + BE_p - E_p E_q) \end{aligned} \quad (2)$$

From (2) we can see that the range of error resulting on the output  $C$  will be  $0..AE_q + BE_p - E_p E_q$ . That is to say the error that the multiplication will incur is governed by the actual correct values of  $A$  and  $B$ , multiplied by the error attained by each input. In terms of maximum error, this occurs when we consider the maximum attainable value of the inputs multiplied by the maximum possible error of the inputs.

#### IV. HARDWARE MODELS

In the previous section we derived error models for adder and multiplier structures. Error is only one metric with which we will base optimization decisions upon. Another crucial piece of information is hardware cost in terms of area.

By performing substitution rather than immediate truncation, we introduce a critical difference in the way hardware will handle this datapath. Unlike the case of immediate truncation, we do not have to change the implementation of downstream operators to handle different bit-widths on the inputs. Likewise, we do not have to deal with alignment issues, as all inputs to operators will have the same location of the binary point.

For example, in an adder, as we reduce the number of bits on the inputs, the area requirement of the adder decreases. The same relationship holds true when we substitute zeros in place of variable bits on an input. This is true because we can simply use wires to represent static zeros or static ones, so the hardware cost in terms of area is essentially zero.

If the circuit is specified in a behavioral fashion using a hardware description language (HDL), this optimization is likely to fall under the jurisdiction of vendor tools such as the technology mapper and the logic synthesizer. Fortunately, this constant propagation optimization utilizing wires is implemented in most current vendor tools.

In the next sections we outline the area models used to perform area estimation of our datapath. We will assume a simple 2-LUT architecture for our target FPGA and validate this assumption through implementation on target hardware.

##### A. Adder Hardware Model

In a 2-LUT architecture, a half-adder can be implemented with a pair of 2-LUTs. Combining two half-adders together and adding an OR gate to complete a full-adder requires five 2-LUTs. To derive the hardware model for the adder structure as described in previous sections, we utilize the example shown in Fig. 3.

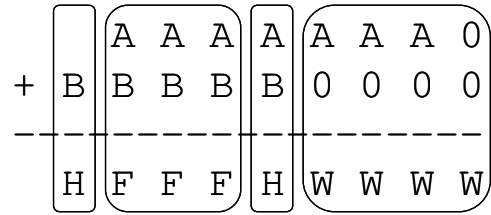


Fig. 3. Adder hardware requirements.

TABLE I  
ADDER AREA

Number	Hardware
$\max( M' - N' , 0)$	half-adder
$\max(M', N') - \max(p, q) -  M' - N'  - 1$	full-adder
1	half-adder
$\max(p, q)$	wire

Starting at the least-significant side, all bit positions that overlap with zeros need only wires. The next most significant bit will only require a half-adder, as there can be no carry-in from any lower bit positions, as they are all wires. For the rest of the overlapping bit positions, we require a regular full-adder structure, complete with carry propagation. Finally, at the most-significant end, if there are any bits that do not overlap, we require half-adders to add together the non-overlapping bits with the possible carry-out from the highest overlapping full-adder bit.

The relationship described in the preceding paragraph is generalized into Table I, using the notation previously outlined. For the example in Fig. 3, we have the following formula to describe the addition.

$$\begin{aligned} &A_m 0_p + B_n 0_q \\ &m = 7, p = 1, n = 5, q = 4 \end{aligned}$$

This operation requires two half-adders, three full-adders, and four wires. In total, 19 2-LUTs.

With the equations in Table I, we can plot the area and error impact of zero substitution. In Fig. 4, the area is plotted as a contour graph against the number of zeros substituted into each of two 32-bit input words. The contour reflects our intuition that as more zeros are substituted into the input words of an adder, the area requirements drop. The plot also highlights the fact that we can manipulate either input to achieve varying degrees of area reduction. Fig. 5 also follows intuition, showing clearly that as more zeros are substituted, the normalized error rate increases.

##### B. Multiplier Hardware Model

We use the same approach to characterize the multiplier. A multiply consists of a multiplicand (top value) multiplied by a multiplier (bottom value). The hardware required for an array multiplier consists of AND gates, half-adders, full-adders, and wires. The AND gates form the partial products, which in turn are inputs to an adder array structure as shown in Fig. 7.

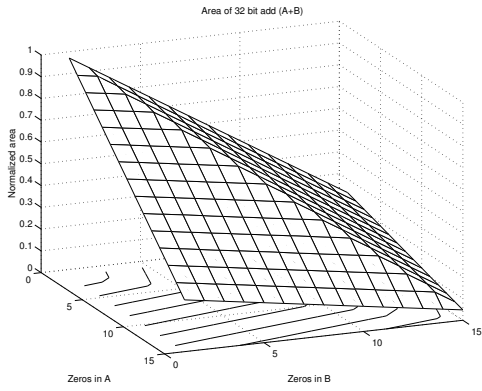


Fig. 4. Adder area vs. number of zeros substituted.

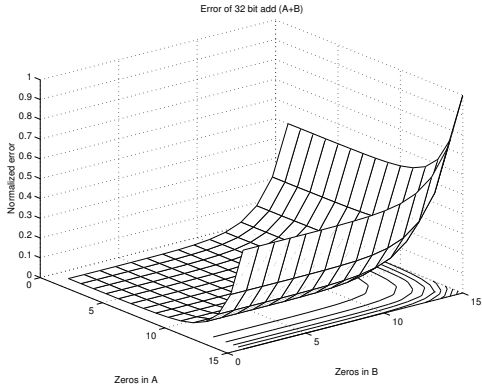


Fig. 5. Adder error vs. number of zeros substituted.

Referring to the example in Fig. 6, each bit of the input that has been substituted with a zero manipulates either a row or column in the partial product sum calculation. For each bit of the multiplicand that is zero, we effectively remove an inner column. For each bit of the multiplier that is zero, we remove an inner row. Thus:

$$A_m 0_p * B_n 0_q$$

$$m = 3, p = 1, n = 2, q = 2$$

is effectively a 3x2 multiply, instead of a 4x4 multiply. This requires two half-adders, one full-adder, and six AND gates,

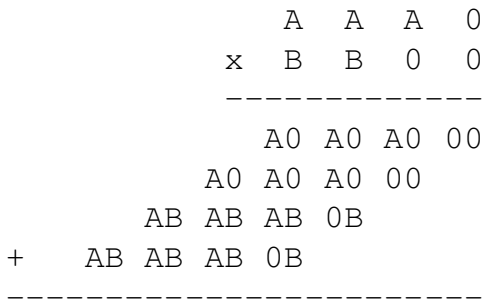


Fig. 6. Multiplication example.

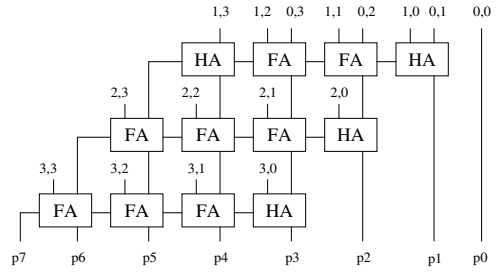


Fig. 7. Multiplication structure.

TABLE II  
MULTIPLIER AREA

Number	Hardware
$\min(m, n)$	half-adder
$mn - m - n$	full-adder
$mn$	AND
$p + q$	wire

for a total of 15 2-LUTs. This behavior has been generalized into formulas shown in Table II.

Again, Table II leads us to extrapolate the area and error impact of zero substitution. These plots are found in Fig. 8 and Fig. 9, respectively. Their interpretation is similar to that of the adder structure.

### C. Model Verification

To verify our hardware models against real-world implementations, we implemented both the adder and multiplier structures in Verilog on the Xilinx Virtex FPGA using vendor-supplied place and route tools. We chose zero-substituted inputs along the spine of Fig. 4 and Fig. 8, meaning equal zero substitution for both inputs.

For the adder structure, we observe in Fig. 10 that our model closely follows the actual implementation area, being at worst, within two percent of the actual Xilinx Virtex hardware implementation.

The multiplier in Fig. 11 has a similar result to the adder, being at worst within 12 percent of the Xilinx Virtex imple-

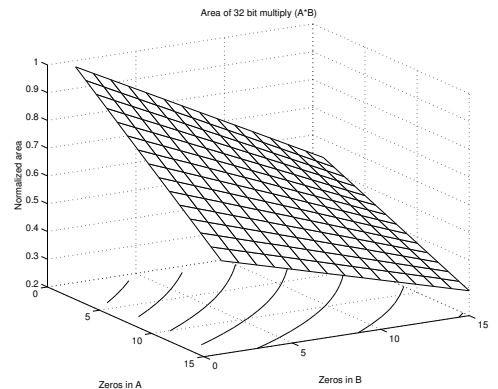


Fig. 8. Multiplier area vs. number of zeros substituted.

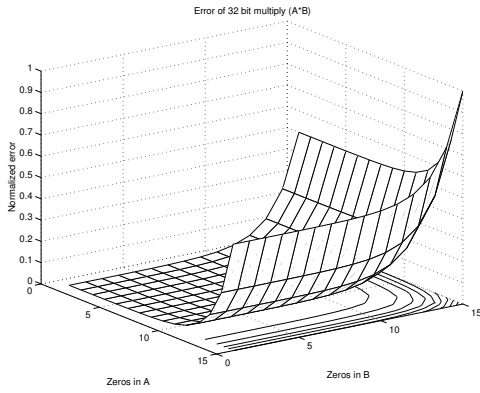


Fig. 9. Multiplier error vs. number of zeros substituted.

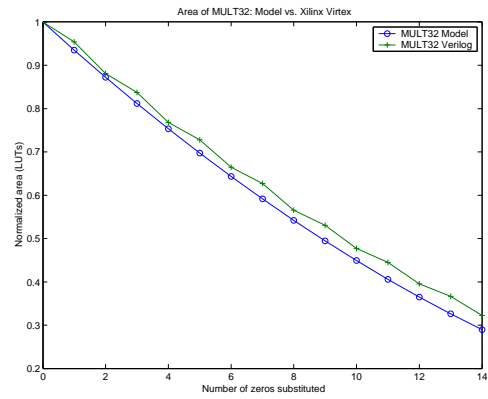


Fig. 11. Multiplier model verification.

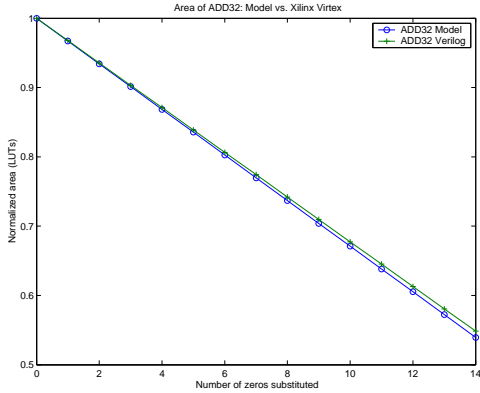


Fig. 10. Adder model verification.

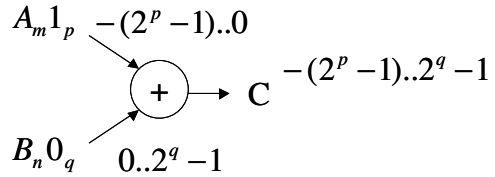


Fig. 12. Normalized error model of an adder.

mentation.

These results support the use of our simple 2-LUT approximation of general island-style FPGAs to within a reasonable degree of accuracy.

## V. OPTIMIZATION METHODS

Using the models described in the previous sections, we can now quantify the tradeoffs between area and error of various optimization methodologies.

### A. The Nature of Error

If we look again at the error models discussed in Fig. 1 and Fig. 2, we see that the error is skewed, or biased, in one direction—positively. As we continue through datapath elements, if we maintain the same zero-substitution policy to reduce the area requirement of our circuits, our lower-bound error will remain zero, while our upper bound will continue to skew toward larger and larger positive values. This also holds true if we were to simply truncate the least-significant bits from our datapath.

This resulting error does not capture our natural understanding of error. More intuitively we consider the “error” of a result to be the *net distance from the correct value*. This implies that the error term can be either positive or negative. Unfortunately, our zero-substitution policy, as defined in previous sections, does not realize this notion of error. But unlike straight

truncation of least-significant bits, we can still utilize our zero-substitution policy to allow us to capture this more intuitive behavior of error. Consider this process “renormalization”. We discuss this as a potential optimization technique next.

### B. Renormalization

It is possible for us to capture the more natural description of error with our method of zero-substitution because the least-significant bits are still present. We can use these bits to manipulate the resultant error range. If instead truncation were performed, no further shaping of the error range would be possible. An example of renormalization in an adder structure is shown in Fig. 12. We describe this method as “in-line renormalization” as the error range is biased during the calculation. It is accomplished by modifying one of the input operands with one-substitution instead of zero-substitution. This effectively flips the error range of that input around zero. The overall effect is to narrow the resultant error range, bringing the net distance closer to zero. Specifically, if the number of substituted zeros and ones are equal, we achieve an error range whose net distance from zero is half that if we were to use zero substitution only.

For example, in Fig. 1, a substitution of  $p, q$  zeros results in an error range of  $0..2^p + 2^q - 2$ . By rethinking the nature of error, with renormalization, this same net distance from the real value can be achieved with more bit substitutions,  $p + 1, q + 1$ , on the input. This will yield a smaller area requirement for the adder. Likewise, the substitution of  $p, q$  zeros with renormalization now incurs half the error on the output,  $-(2^p - 1)..2^q - 1$ , as shown in Fig. 12.

As with the adder structure, renormalization of the multiplier is possible by using different values for least-significant

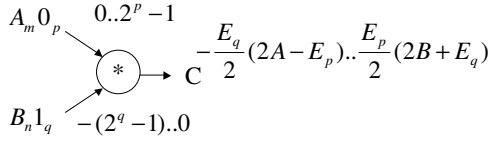


Fig. 13. Normalized error model of a multiplier.

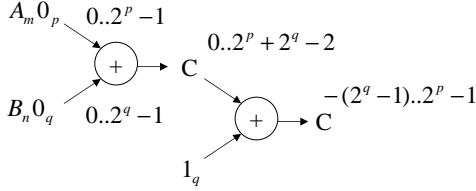


Fig. 14. Inserting a constant add performs an “active renormalization”.

bit substitution, yielding an error range that can be biased. Fig. 13 depicts a normalization centered on zero by substituting ones instead of zeros for input  $B$ .

Another method of renormalization can be accomplished after an operation, or operations, have been completed. By inserting a constant addition, we can accomplish a very similar biasing of error range, this time referred to as “active renormalization”. An example is shown in Fig. 14.

In biasing the error range, either through “in-line” or “active” renormalization, we may incur an area penalty over the base area requirement of the basic (addition or multiplication) operation. For the “in-line” renormalization, any ones substituted that do not line up with substituted zeros in the opposing input will need to be computed upon, consuming adder resources. When substituted ones and zeros on the inputs completely overlap, there is no immediate area penalty, as the static ones can be represented, as before, with wires tied to constants. But even if the substituted ones and zeros completely overlap, consideration must be made for downstream operations, as we now have ones in the least-significant bit positions which must be operated upon in any subsequent calculations. This may adversely impact the overall area of the circuit, and “active” renormalization should be considered an alternative that can be implemented cheaply later on in the datapath to “fix up” the error range using a constant bias.

“Active” renormalization also has an area penalty. As it is simply an addition between an input value and a constant positive bias, the impact is simply the area requirement of the biasing adder.

### C. Alternative Arithmetic Structures

As discussed in previous sections, our zero-substitution method for multipliers gives a reduced area footprint at the cost of increased error in the output over an exact arithmetic multiplication. An alternative to this method of area/error tradeoff is one described in [8]. This work, and the work of others ([9], [10]), focuses on removing a number of least-significant columns of the partial-product array.

As described in [9], by removing the  $n$  least-significant columns from an array-multiplier multiplication, we save (for

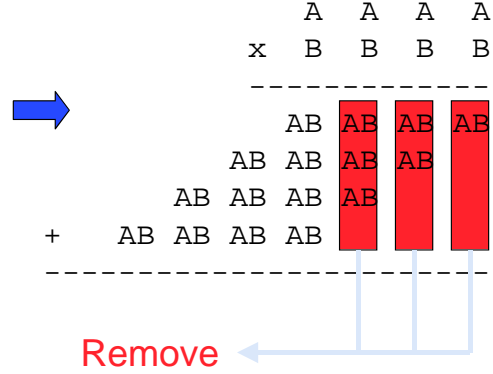


Fig. 15. A truncated multiplier removes least-significant columns from the partial product array.

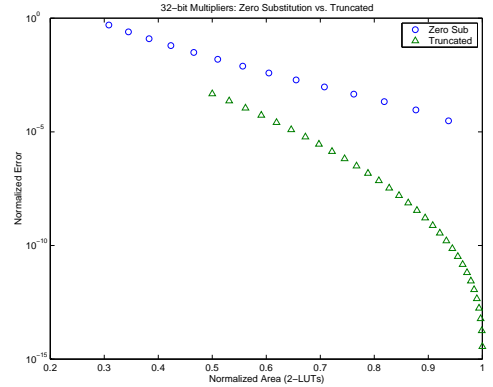


Fig. 16. Error to area profile of zero-substitution 32-bit multiplier and truncated 32-bit multiplier.

$n \geq 2$ )  $\frac{n(n+1)}{n}$  AND gates,  $\frac{(n-1)(n-2)}{2}$  full adders, and  $(n-1)$  half adders. The column removal is depicted in Fig. 15.

This method has a different area-to-error tradeoff profile, and is shown in Fig. 16 for a 32-bit multiplier, and in Fig. 17 for a range of differently-sized multipliers.

While the truncated multipliers have a more favorable area-to-error profile, one drawback in their use is that they require the full precision of both operands to be present at the inputs of the multiplier. This has the effect of requiring higher precision on upstream computations, possibly negating the area gain at a particular instance of a multiplier by requiring larger operations at upstream nodes. This makes it an unlikely candidate for use in multiplications that happen near the end of the datapath.

### D. Precision Steering

Again, utilizing the presence of actual programmable bits at the least-significant end of datapath, we can demonstrate control over error distribution throughout our system. We can compute, at each node, the contribution made to the overall area and error. It follows that given different precisions of input signals, the contribution to area and error from different branches may be unequal. Therefore, we may be able to “steer” the error toward one branch over another.

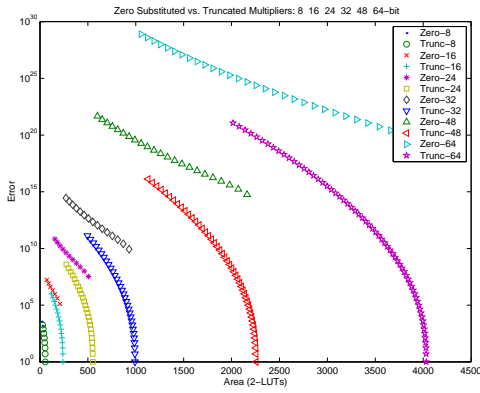


Fig. 17. Error to area profile of zero-substitution multipliers and truncated multipliers.

This notion of trading error in one branch for another proves useful if, for example, the computation performed in one branch is used in a conditional expression. In some cases, it would be wise to allocate more precision to that decision-making branch so it is more likely to choose the correct answer. A similar example would be if three branches of different length, and thus relative area consumption, converged together. An area savings might be realized if the developer steered the error at the output toward the branch with the most operations. Thus, an area savings could possibly be affected over more operator nodes, reducing overall area requirements. Finally, given several branches of computation contributing to a single output, it may be in the interest of the developer to have the error evenly balanced across all incoming branches, regardless of their length and complexity.

## VI. BENCHMARKS

In order to quantify the effectiveness of our proposed methods, we perform the optimizations on some example circuits.

### A. Steering and Renormalization

To illustrate the idea of error “steering”, we present this example. Fig. 18 depicts a synthetic circuit with a slightly biased datapath. At the final adder,  $+_4$ , one branch,  $E$  is short, while the other branch, the output from  $+_3$  is longer. To set a baseline, assume that all inputs are 4-bit values without any inherent error. This results in a total of 77 2-LUTs consumed.

Assume the developer has found that an error range of 0..8 is tolerable at the output. If the goal is to reduce area requirements, then we may choose to steer the error at node  $+_4$ . As previously discussed, we may attempt three (and possibly more) types of optimizations.

1) *Towards shorter branch*: If the longer branch were used to compute a conditional expression, we can steer error away from that branch. For the node  $+_4$ , we can allocate an error range of 0..7 to the lower half, and 0..1 to the upper half. If we again steer the error at  $+_3$  down to  $+_2$ , we can substitute a single zero on either of the inputs to  $+_2$ . This results in a

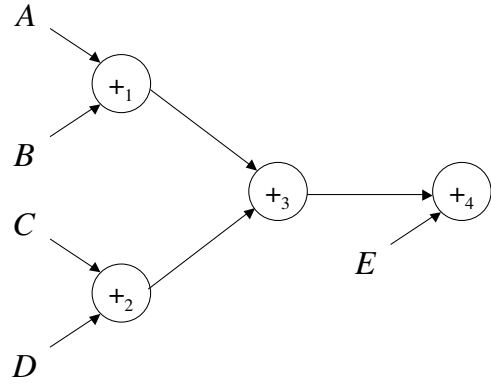


Fig. 18. Precision steering example.

total area requirement of 57 2-LUTs. The increase in error is accompanied by a decrease in area, as expected.

2) *Towards longer branch*: Alternatively, the developer could steer the error toward the longest branch in order to realize an area savings. For node  $+_4$ , all of the 0..8 error can be steered to the upper branch. From  $+_3$ , another steering decision can be made, this time to put 0..6 on the upper branch and 0..2 on the lower branch. This, in effect, gives us the inputs:

$$A_20_2, B_20_2, C_30_1, D_30_1, E_40_0$$

This combination of inputs gives us a total area requirement of 47 2-LUTs, a reduction over the previous steering example while still incurring the same error range at the final node.

3) *Balanced error*: The final example illustrates keeping area constant while reducing the error range. To achieve this, we utilize a more balanced steering of error. Perhaps easier is to start at the inputs. Here we can insert zeros uniformly across all inputs:

$$A_30_1, B_30_1, C_30_1, D_30_1, E_30_1$$

This achieves an error range at the input to  $+_3$  of 0..2 on both branches, and 0..4 on the upper input branch to  $+_4$  and 0..1 on the lower branch. In total, this implementation gives us an error range of 0..5 and an area requirement of 57 2-LUTs.

4) *Renormalization*: We can perform renormalization on this circuit to achieve better results in terms of both area and error. For this example, let us make the baseline inputs 8-bits. We calculate the area and error impact of several different scenarios and summarize the impact in Table III. The different optimization methods include zero-substitution with one and two bits and two different renormalization implementations.

The baseline implementation performs no optimizations and has the least amount of error but the highest area requirement. The two zero-substitution implementations achieve better area results (14.6% and 34.1%, respectively) but require the user to tolerate more error at the output.

The implementation of Renormalize  $A$  performs “in-line” renormalization on  $+_1$  and  $+_2$ . The inputs  $A, B, C, D$  have three bits of zero substitution ( $A_50_3$ , for example), and input

TABLE III  
IMPACT OF DIFFERENT OPTIMIZATION TECHNIQUES I

Optimization	Area	Error
None	157	0
Zero-substitute 1b	137	0..5
Zero-substitute 2b	117	0..15
Renormalize A	103	-16..15
Renormalize B	110	-11..12

$E$  is  $E_60_2$ . The “in-line” renormalization has very little area impact because the bias value of +8 can be accomplished by turning the least-significant bit half-adders in both  $+_1$  and  $+_2$  into full-adders and adding the bias value through the carry-in. This renormalization makes the output error range of both  $+_1$  and  $+_2$   $-8..6$ . As a result, the overall area of 103 2-LUTs is 52.4% less than the baseline implementation. Comparing this to the two zero-substitution methods (which are akin to straight truncation), we achieve 33% less area than one-bit substitution, and 13.6% less area than two-bit substitution. Finally, compared to two-bit substitution, we achieve nearly the same net-error range (absolute distance from zero) while saving area.

The implementation of Renormalize  $B$  achieves an even tighter error range of  $-11..12$  with slightly higher area requirement than Renormalize  $A$ . In this implementation the inputs are  $A_50_3$ ,  $B_60_2$ ,  $C_50_3$ ,  $D_60_2$ , and  $E_61_2$ . Again,  $+_1$  is renormalized by rolling in a +8 bias into the adder structure and achieving an output error range from  $+_1$  of  $-8..2$ . Also,  $+_4$  is renormalized to  $-3..0$ . The resulting implementation requires 110 2-LUTs, 6.4% better than the two-bit substitution, but achieves a 25% narrower maximum error range.

### B. Wavelet Transform

In this benchmark, we perform similar optimizations to the ones in the previous synthetic benchmark, but upon a wavelet transform algorithm. The wavelet transform is a form of image processing, primarily serving as a transformation prior to applying a compression scheme, such as SPIHT[11], [12]. A typical discrete wavelet transform performs a high-pass filter and a low-pass filter over the input image in one dimension. The results are then downsampled by a factor of two and the process is repeated in the other dimension. Each pass results in a new image composed of a high-pass and low-pass sub-band, each half the size of the original input stream. These sub-bands can be used to reconstruct the original image.

Fig. 19 illustrates the low-pass filter section of the wavelet transform that will be the benchmark for our optimization techniques. The results of several optimization techniques are summarized in Table IV.

We first create a baseline by using 8-bit values for all the primary inputs. This baseline has an area impact of 2257 2-LUTs and introduces no error into the datapath. By performing a single bit of zero-substitution we can achieve a 21.9% reduction in area at the cost of error.

We then perform two types of renormalization on the zero-substituted implementation. In Renormalize  $A$ , we perform

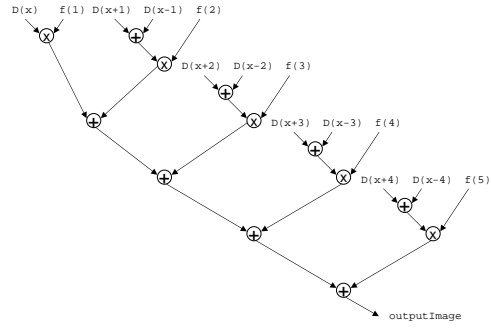


Fig. 19. Structure of the low-pass portion of the wavelet transform.

“active” renormalization by performing the addition of a bias value (2048) to shift the error range to  $-2048..2537$ —a much narrower absolute error range than the one-bit zero-substitution method. This renormalization adds very little area to the implementation because the bias value of 2048 is mostly zeros in binary form, which has zero cost for overlapping zeros in the lower-order bits.

Alternatively, we may renormalize with a more exact bias value to obtain an even narrower absolute error. By biasing by half the maximum output error from the last addition,  $0..4585$ , we require slightly more area than the Renormalize  $A$  method, but obtain a 9.6% narrower error range.

While the two former renormalization techniques achieve good results, they do not maintain a balanced error range at intermediate points in the datapath. For circuits that have intermediate nodes with some degree of fanout, it may be disadvantageous to perform “active” renormalization for each fanout path. Instead, we can perform “in-line” renormalization at each node and obtain the results as shown for the method “Renormalize”. While this method requires 13.4% more area than the one-bit zero-substitution method, it achieves a 50.1% narrower absolute error range. While not as efficient as either Renormalize  $A$  or Renormalize  $B$ , it is still compelling as it maintains a balanced error range throughout the datapath.

Finally, we show the results of using truncated multipliers in place of traditional multipliers throughout the datapath. This method has a greater impact than the previous methods as the truncated multiplier is very efficient and introduces very little error into the output. The Truncated method requires 11.6% less area than the perfect implementation. Versus the single-bit zero-substitution method, the Truncated multiplier method requires 14.7% more area but has 18.7 times less error. Versus Renormalize  $A$ , it has 13.3% more area and 10.4 times less error. And versus Renormalize  $B$ , it has 10.5% more area and 9.36 times less error.

While the results of this optimization method are compelling, it is worth noting that the truncated multiplier requires the inputs to be full precision instead of zero-substituted or truncated in any way to obtain the narrow output error ranges. For the wavelet transform structure, since the multiplier nodes are near the input, we can utilize truncated multipliers.

TABLE IV  
IMPACT OF DIFFERENT OPTIMIZATION TECHNIQUES II

Optimization	Area	Error
None	2257	0
Zero-substitute 1b	1763	0..4585
Renormalize <i>A</i> Zero-substitute 1b	1785	-2048..2537
Renormalize <i>B</i> Zero-substitute 1b	1828	-2292..2293
Renormalize	2000	-2299..2291
Truncated	2022	0..245

## VII. CONCLUSIONS AND FUTURE WORK

We have described and motivated the need to investigate the optimization of the least-significant bit position. In order to do so, we have proposed models of area and error for an alternative area reduction technique to straight truncation—zero-substitution. Using this method and models, we have proposed several optimization techniques aimed at giving the developer more control over the area-to-error tradeoff during datapath precision optimization. These techniques include renormalization, which allows the user to capture a more intuitive notion of error as the net distance from the correct answer during and post calculation; alternative arithmetic structures, specifically truncated multipliers, which can yield better area-to-error profiles than traditional multipliers in certain situations; and precision steering, which allows the user to devote differing amounts of error to different branches of a calculation.

Finally, we have demonstrated, for a small set of benchmarks, the ability of our methods to achieve a wider range of area and error profiles than straight truncation.

In future work, we will integrate these optimization techniques into our precision optimization tool, Précis [1]. This will allow a more automated (and exhaustive) exploration of the various combinations of techniques proposed here.

## REFERENCES

- [1] M. L. Chang and S. Hauck, “Précis: A design-time precision analysis tool,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 229–238.
- [2] M. Stephenson, J. Babb, and S. Amarasinghe, “Bitwidth analysis with application to silicon compilation,” in *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, June 2000.
- [3] M. W. Stephenson, “Bitwise: Optimizing bitwidths using data-range propagation,” Master’s thesis, Massachusetts Institute of Technology, May 2000.
- [4] W. Sung and K.-I. Kum, “Simulation-based word-length optimization method for fixed-point digital signal processing systems,” *IEEE Transactions on Signal Processing*, vol. 43, no. 12, pp. 3087–3090, December 1995.
- [5] S. Kim, K.-I. Kum, and W. Sung, “Fixed-point optimization utility for C and C++ based digital signal processing programs,” in *Workshop on VLSI and Signal Processing*, Osaka, 1995.
- [6] A. Nayak, M. Haldar, *et al.*, “Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs,” in *Design Automation & Test*, March 2001.
- [7] G. A. Constantinides, P. Y. Cheung, and W. Luk, “The multiple wordlength paradigm,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.
- [8] Y. Lim, “Single-precision multiplier with reduced circuit complexity for signal processing applications,” *IEEE transactions on Computers*, vol. 41, no. 10, pp. 1333–1336, October 1992.

- [9] M. J. Schulte and J. Earl E. Swartzlander, “Truncated multiplication with correction constant,” in *VLSI Signal Processing VI, IEEE Workshop on VLSI Signal Processing*, October 1993, pp. 388–396.
- [10] K. E. Wires, M. J. Schulte, and D. McCarley, “FPGA resource reduction through truncated multiplication,” in *Proceedings of the 11th International Conference on Field Programmable Logic and Applications*, August 2001, pp. 574–583.
- [11] T. W. Fry, “Hyperspectral image compression on reconfigurable platforms,” Master’s thesis, University of Washington, Seattle, WA, May 2001.
- [12] T. W. Fry and S. Hauck, “Hyperspectral image compression on reconfigurable platforms,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 251–260.