Variable Precision Analysis for FPGA Synthesis

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 accurate estimation of the precision requirements of an algorithm subject to user-defined area and error constraints. We derive area-to-error models of a general island-style FPGA architecture and present methods for incorporating these models into the decision-making process of precision optimization. Finally, we present some preliminary results describing the effectiveness of our techniques in guiding users to find a suitable area-to-error tradeoff.**

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 and our methodology for finding the precision of a circuit's datapath. Finally, we present our results and conclusions in the framework of our designer-centric precision analysis tool, Précis[1].

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, including 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, typically via roundoff or 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 range that the datapath may represent. 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 $2^{-3} = 0.125$.

Having a fixed-point datapath means that results or operations will exhibit some quantity of error compared to their infinite-precision counterparts. This quantization error can be introduced on 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 that an automatic optimization method simply cannot account for.

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. In this paper we provide a methodology for performing the least-significant-bit analysis in a user-guided fashion that utilizes both area and error to provide the user with enough information to make informed optimization decisions.

B. Models and Methodology

The observation that the relative bit position of the least-significant bit introduces a quantifiable amount of error over a true floating-point or infinite-precision datapath is crucial to our methodology. As described in [6], what can be considered an error-analysis phase must be performed in order to determine the cumulative error of the datapath. Whereas [6] utilizes a completely automated approach—inferring the resolution required of intermediate operations in compiler passes and through user-specification of tolerable error at the output nodes—we propose a more user-guided approach that, in addition to accounting for error accumulated in the datapath, provides the user with a more powerful methodology to analyze the area-to-error tradeoff at each intermediate node.

We begin our discussion with the simpler case of an integer-only datapath. We will describe our number format and notation, and the corresponding hardware error and area models. We then extend this idea to include binary fixed-point notation in order to cope with a real-valued datapath.

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. We could simply truncate the bits from the least-significant side to reduce the number of bits required to store and operate on this word, but a simpler solution would be to replace the bits with zeros instead. Thus, for an M'-bit value, we have the notation $A_m 0_p$. This is an integer-valued 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.

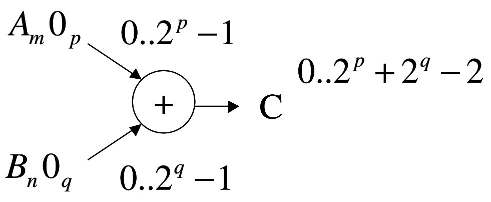

Fig. 1. Error model of an adder

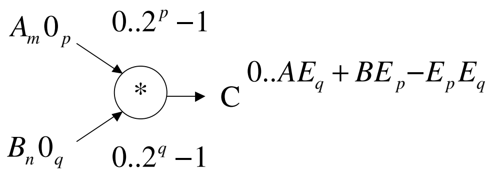

Fig. 2. Error model of a multiplier

Having performed a reduction in the precision that can be obtained by this datapath with a substitution of zeros, we have introduced an 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 $0..2^p - 1$. 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 the maximum error.

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, possibly 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 possibly 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

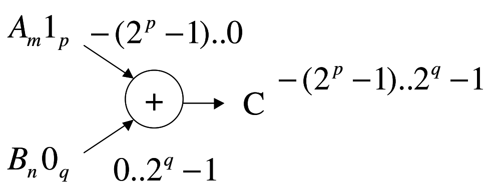

Fig. 3. Normalized error model of an adder

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_m0_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):

$$\begin{aligned} C &= AB - BE_p - AE_q + E_pE_q \\ &= AB - (AE_q + BE_p - E_pE_q) \end{aligned} \tag{2}$$

From (2) we can see that the range of error resulting on the output C will be $0..AE_q + BE_p - E_pE_q$. That is to say the error that the multiplication will incur is governed by the actual correct value 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.

C. Renormalization and Rethinking Error

Looking more closely at the error introduced in both of the models 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 for bit-width reduction, our lower-bound error will remain zero, while our upper bound will continue to skew to larger positive numbers.

In some cases, a more useful error bias would be to attempt to "center" the range of error on a good average case. For example, if the developer knows that an input signal has a non-uniform distribution, they can bias the error such that the most likely case will achieve zero relative error. Consider this procedure "renormalization". An example with an adder structure is shown in Fig. 3.

For instance, if an algorithm is known to have a predetermined steady-state value, then the developer can bias the error term by inserting a pattern other than zeros to have this steady-state value fall in the center of the error range.

As with the adder structure, renormalization of the multiplier is possible by using different values for least-significant bit substitution, yielding an error range that can biased. Fig. 4 depicts a normalization centered on zero by substituting ones instead of zeros for input B.

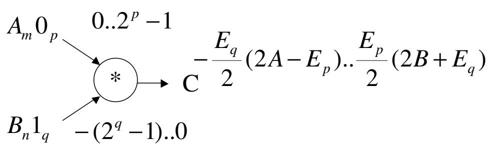

Fig. 4. Normalized error model of a multiplier

Renormalization is perhaps most effective when we change the way error in our system is quantified. Renormalization allows us to move the center of the error output range to any value. If we consider the error as the *net distance from the correct value*, for a given tolerance of error on the output, we can perform more constant substitutions on the inputs than if the error was skewed only positively. In essence, if error is the net distance from correct, we have effectively doubled our error range for the same number of input constant substitutions.

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. 3.

IV. Hardware Models

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

As we reduce the number of bits on the input to, for instance, an adder, 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 the vendor place and route tools. Fortunately, this simple 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 in the following sections.

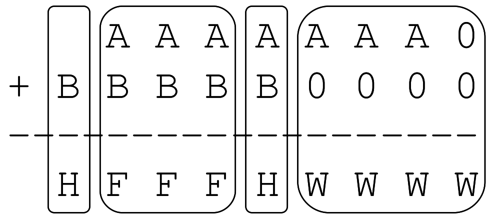

Fig. 5. Adder hardware requirements

TABLE I
ADDER AREA

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

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. 5.

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. 5, we have the following formula to describe the addition.

$$A_m 0_p + B_n 0_q$$
$$m = 7, p = 1, n = 5, q = 4$$

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. 6, 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. 8 also follows intuition, showing clearly that as more zeros are substituted, the normalized error rate increases.

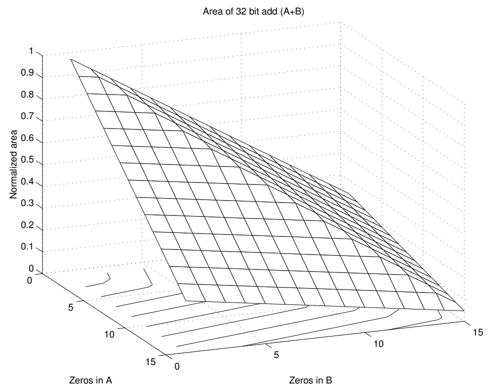

Fig. 6. Adder area vs. number of zeros substituted

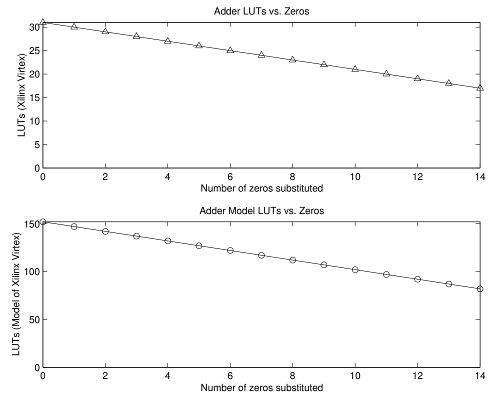

Fig. 7. Adder model verification

To verify our hardware models against real-world implementations, we implemented the adder structure in Verilog on the Xilinx Virtex FPGA using the vendor-supplied place and route tools. Choosing zero-substituted inputs along the spine of Fig. 6, meaning equal zero substitution for both inputs, we see in Fig. 7 that our model closely follows the actual implementation area.

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. 10.

Referring to the example in Fig. 9, 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 rows. 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 three half-adders, one full-adder, and six AND gates,

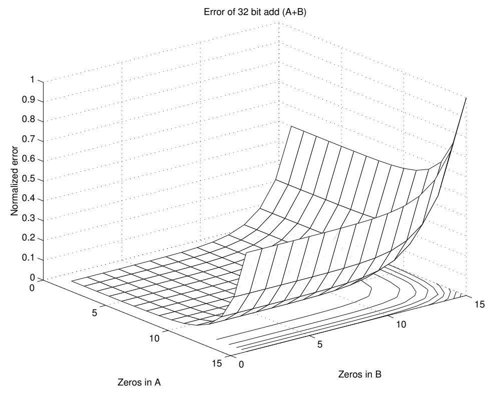

Fig. 8. Adder error vs. number of zeros substituted

```
              A  A  A  0
           x  B  B  0  0
           -------------
             A0 A0 A0 00
          A0 A0 A0 00
       AB AB AB 0B
+   AB AB AB 0B
------------------------
```

Fig. 9. Multiplication example

for a total of 17 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. 11 and Fig. 12, respectively. Their interpretation is similar to that of the adder structure, and their behavior has likewise been verified with hardware implementation on Xilinx Virtex architectures, as shown in Fig. 13.

V. Precision Steering

Armed with models describing the area and error impact of reduced-precision datapaths, we can utilize this more detailed information to aid in manual precision analysis. In particular, we can explore a range of area-to-error tradeoffs.

We can now compute, at each operation node, the contribution made to the overall area and error. It follows that given

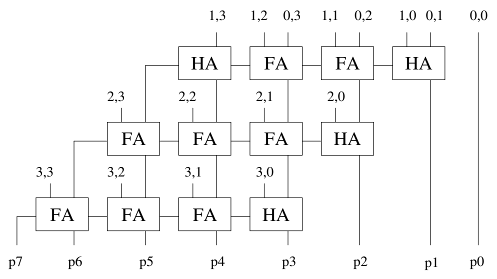

Fig. 10. Multiplication hardware requirements

TABLE II
Multiplier Area

Number	Hardware
$\min(m,n)+1$	half-adder
$mn-m-n$	full-adder
mn	AND
$p+q$	wire

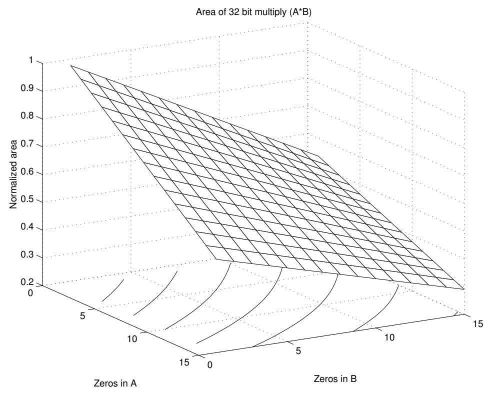

Fig. 11. Multiplier area vs. number of zeros substituted

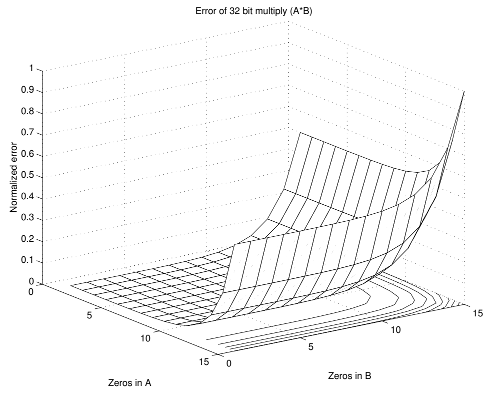

Fig. 12. Multiplier error vs. number of zeros substituted

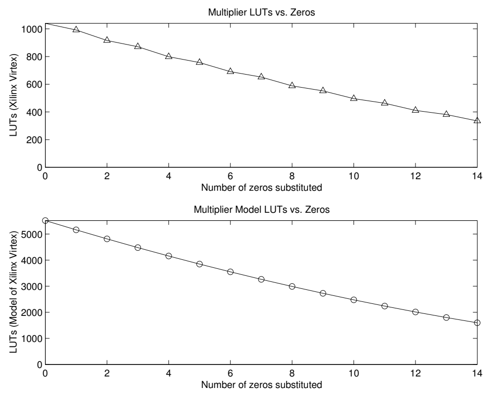

Fig. 13. Multiplier model verification

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.

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.

To illustrate the idea of error "steering", we present an illustrated example. Fig. 14 depicts 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.

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

B. 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:

$$A_2 0_2, B_2 0_2, C_3 0_1, D_3 0_1, E_4 0_0$$

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.

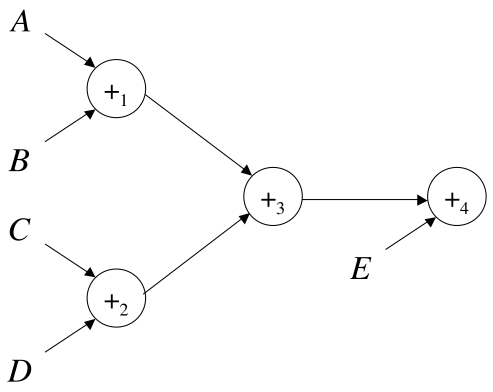

Fig. 14. Precision steering example

C. 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:

$$A_3 0_1, B_3 0_1, C_3 0_1, D_3 0_1, E_3 0_1$$

This achieves 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.

VI. Fixed-Point Extension

Our previous discussion encompassed only integer-valued datapaths. While this is a limitation, extending our format and methodology to a real-valued datapath is not difficult.

In a fixed-point, or real-valued datapath, the binary point may be anywhere. In our integer-valued datapath, the binary point was implicitly at the least-significant end of the word. This allowed us to ignore the task of alignment, as all words were already aligned.

Re-alignment of input words to an operator is a simple task, but has implications. Obviously, in order to align two words whose binary point location differs, we simply right-pad the word whose binary point location is further to the right. In other words, we pad zeros to the right of the word that has less precision. With aligned input words, we can perform operations as described in previous sections. Since the format of numbers in the system must be determined beforehand, alignment can either be factored into the input signal, or additional hardware can be inserted at points where alignment will be necessary.

VII. Guidelines and Complexities

In the previous sections, we demonstrated how steering the precision in even a simple datapath can have a significant impact on the area requirements of an implementation. Extending this to much larger circuits with more primary inputs, it becomes clear that guidelines to semi-automatic precision analysis should be discussed.

When performing steering of error at operator nodes, the area benefits are actually quantized to the number of whole-bit substitutions that can be made at the inputs. For example, an error range of $0..2$ cannot be fully utilized at a single input to an adder. This is because the error introduced at an input is governed by $0..2^p - 1$, as shown in Fig. 1. This only allows us to substitute a single constant at the LSB position, giving us an effective error contribution to the adder of $0..1$, which doesn't consume all the error allocated to this node. By contrast, an error range of $0..2$ can be fully utilized if split between two inputs, as was done in both §V-B and §V-C.

On the other hand, if the error cannot be distributed completely, we can reduce the overall output error by taking up the slack in error. While not shown explicitly in our examples, it is similar in effect to the example shown in §V-C. By starting with uniform error at the inputs, we achieve an output error range of $0..5$, better than the alloted $0..8$ that was tolerated by the user.

Another demonstrated guideline that can be applied to semi-automatic precision analysis is the fact that area benefits increase as error is steered toward longer branches. This follows from the fact that with more operator nodes, we have more opportunity to reduce the area of ancestor nodes provided that we can propagate whole-bit substitutions to the primary inputs as discussed.

VIII. Conclusions and Future Work

We have presented our methodology for utilizing area and error information in performing precision analysis. We have derived mathematical models for the area and error of operators that utilize reduced-precision inputs. We have shown how these models can be used during precision analysis. We have demonstrated the impact of performing this type of analysis on both area and error. Finally, we have given basic precision optimization guidelines for use when both area and error model values are at hand.

We are currently extending our design-time precision analysis tool [1] to incorporate some of the ideas in this paper. Some of the proposed enhancements include:

- Performing the calculation of area and error for a particular implementation automatically.
- Developing more models for common functions.
- Deriving default precision requirements of intermediate nodes from user-supplied output error tolerance and input error specifications.
- Providing an easy and intuitive method for adjusting precision and observing the impact.

Furthermore, we hope to continue the development and verification of our models. We also aim to implement real-world benchmarks that demonstrate, more concretely, the effectiveness of our methods.

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.