Variable Precision Analysis for FPGA Synthesis

Mark L. Chang

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2004

Program Authorized to Offer Degree: Electrical Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Mark L. Chang

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

_____

Scott Hauck

Reading Committee:

_____

Scott Hauck

_____

Carl Ebeling

_____

Eve Riskin

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Variable Precision Analysis for FPGA Synthesis

by Mark L. Chang

Chair of Supervisory Committee:

Professor Scott Hauck
Electrical Engineering

FPGAs and reconfigurable computing platforms have become popular solutions for applications requiring high speed and reprogrammability. Yet developing for these platforms requires hardware design knowledge to exploit their full potential. In particular, retargeting algorithms conceived and prototyped on general-purpose processors for hardware devices often requires shifting from a floating-point arithmetic paradigm into a fixed-point one. Analyzing the tradeoffs between area consumption, algorithmic precision, output error, and overall circuit performance when making this paradigm shift is a difficult, time-consuming, and error-prone task with little to no assistive tool support.

This dissertation details my research in enabling an easier transition from floating-point to fixed-point arithmetic for FPGAs. This includes a detailed background on FPGAs, an in-depth discussion of precision analysis and data path optimization, a review of past and current precision analysis research efforts, and motivation for my contribution to research in this area. We also present Précis, a novel user-centric precision optimization tool that serves as a software vehicle to demonstrate and verify our methodologies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

One thing that cannot be captured in a dissertation is the human element. Over my many years of graduate school I have made several new friends and collegues that I would like to recognize, because for me, this dissertation is a reminder of the human thread that binds this text.

One of the greatest aspects of being in academia is having smart people to kick around ideas with. My fellow graduate students Ken Eguro, Mark Holland, Chandra Mulpuri, Paul Nimbley, Shawn Phillips and Akshay Sharma have all faithfully played that role. Filling white boards, fielding emails, phone calls, and having endless discussions is what makes this whole process worthwhile.

To the two that came before me, Andrew Kamholz and Agnieszka Miguel. They showed me how it was done and that there was indeed life after a doctorate. I follow in their footsteps still.

Special thanks must be given to one fellow student. When I first started graduate school, Katherine Compton welcomed me to the lab and to the city. We learned how to be graduate students together, for better or worse, and without her, I would have had to chart many waters alone.

Beyond my own advisor, Prithviraj Banerjee, Carl Ebeling, and Eve Riskin have provided challenges, direction, and priceless professional advice. To them I am grateful.

Financial support, a luxury that many graduate students do not have, has made this research possible. This work was funded in part by a Royal E. Cabell Fellowship, an Intel Foundation Graduate Fellowship, the Defense Advanced Research Projects

Agency, the National Aeronautics and Space Administration, and the National Science Foundation.

Finally, I will pay a lifelong debt to my advisor, Professor Scott Hauck. A friend and mentor, he has done more than any other advisor I've seen to make my life just that—a life. While I'm sure he doesn't want to get the reputation of not being a hard-driving advisor, he has made it a priority that graduate school be a fulfilling life experience and not just a purely academic endeavor. He is an advisor in every sense of the word, and one worth emulating.

DEDICATION

To my parents for bearing me, to Scott for bearing down on me, and to my beloved wife Caryn, for bearing with me.

Chapter 1

INTRODUCTION

The proliferation of the personal computer has revolutionized the average person's ability to perform calculations and computations. The relative ease with which a general-purpose processor can be programmed and reprogrammed is in part responsible for this revolution. Software, as the name suggests, is almost infinitely flexible, allowing general-purpose processor-based machines to perform almost any computational task.

However, even today's powerful computers cannot be used for every application. Often, a combination of criterion such as cost, performance, power consumption, and size, dictate the use of custom hardware, known as Application Specific Integrated Circuits (ASICs). Unfortunately, in most cases, the design, testing, and fabrication of an ASIC is out of reach, both in terms of expertise as well as cost. Even if the developer did not face these obstacles, an ASIC is *application specific* by nature, meaning it is inherently fixed-function silicon designed solely to perform a fixed number of functions while fitting within some set of criterion. Regardless, this is a tradeoff many systems have to make in order to meet performance constraints.

In recent years, reconfigurable hardware devices have matured to a point that they pose a challenge to the ASIC along many evaluation axes. Reconfigurable hardware has been a growing area of research in both industry and academia, promising high speed, high throughput, low power consumption, and infinite reprogrammabil-

ity. They essentially offer hardware that is both flexible like software, and as fast as hardware, a tantalizing combination.

At the heart of many reconfigurable hardware platforms is the field-programmable gate array, or FPGA [11,12,71]. The most popular type of FPGA is the "island-style" FPGA, which is a regular fabric of interconnected reprogrammable computational units whose interconnection fabric is also reprogrammable. Typical computational units consist of some number of lookup tables into which boolean equations can be mapped, and a state-holding element, such as a flip-flop. These units have their inputs and outputs routed to an interconnection fabric that connects every individual "cell" together, forming a two-dimensional mesh. The lookup table contents and the control for the interconnection switching points are both controlled by static-RAM, meaning that an entire FPGA's configuration—and therefore its function—can be quickly and infinitely reprogrammed, much like a memory.

Even though an FPGA obviates the need to fabricate a custom chip for every unique set of tasks, the FPGA is still a hardware device that requires hardware design expertise, methodologies, and tools. The design prerequisites for FPGAs are often very similar to designing ASICs. Perhaps the most striking difference between developing software versus hardware is that developers must think spatially as well as procedurally. The performance of a circuit (on an FPGA or in ASIC form) is strongly influenced by the physical layout and size of the circuits that perform the computation. Consequently, the area consumed by an implementation is a critical design consideration.

One important design tradeoff is in the widths of the data paths within a circuit. Unlike general-purpose processors, where data-type primitives such as `char, int,` and `float` provide abstractions of data path width, a hardware device like an FPGA works at the single-bit level. The developer can tune data paths to any word size desired, a level of customization that software developers rarely consider. The ability to tune data paths can result in complications; if too much width is allocated to the

data path and the computational elements along that data path, the area requirements increase, potentially increasing delay and power consumption, which decreases overall performance. On the other hand, reduction of the data path width and computational elements can be taken too far, introducing excessive errors in the output due to hardware behaviors such as overflow, roundoff, saturation, and truncation.

This paradigm shift in arithmetic and number format becomes an obstacle when taking an algorithm originally prototyped in software on a general-purpose processor and implementing it in hardware on an FPGA. The conversion from an infinite-precision conception of an algorithm utilizing floating-point arithmetic to the fixed-point arithmetic world of an FPGA is difficult. Unfortunately, this critical design task is addressed by few, if any, commercial tools that otherwise assist in this translation and implementation.

The work presented in this dissertation describes methodologies, practices, and tools for effective precision analysis to assist in the conversion from floating-point arithmetic into fixed-point arithmetic destined for implementation in a hardware device such as an FPGA. This dissertation is organized as follows:

- *Chapter 2: Field-Programmable Gate Arrays* provides technical background on common FPGA architectures and design methodologies.

- *Chapter 3: Precision Analysis* motivates the problem of precision analysis and describes previous research efforts in this area.

- *Chapter 4: Most-Significant Bit Optimization* describes a technique and tool for optimizing the position of the most-significant bit of a data path.

- *Chapter 5: Least-Significant Bit Optimization* compliments Chapter 4 and describes a novel technique for optimizing the position of the least-significant bit.

- *Chapter 6: Conclusions and Future Work* concludes this dissertation and presents directions for future work.

Chapter 2

FIELD-PROGRAMMABLE GATE ARRAYS

The FPGA is an increasingly popular device in reconfigurable systems. As the technology has matured, gate capacity has increased and costs have been reduced, allowing FPGAs to become useful and cost-effective in a wider variety of applications. This chapter will provide a technical background of FPGAs and common FPGA architectures, as well as insight into the intricacies of developing for FPGAs.

2.1 Architecture

By far the most popular architectural style for current FPGAs is the island-style lookup table-based FPGA. This architecture, depicted in Fig. 2.1, uses logic blocks (islands) that are tiled in a two-dimensional array and connected together through an interconnection network. The logic blocks are the computational medium within the FPGA fabric, while the interconnection network serves to route signals between cells. Both the function performed in the logic block and the routing of signals in the interconnection fabric are programmable. In most FPGAs destined for the consumer market, the programming mechanism is a static-RAM (SRAM) bit, each of which controls various multiplexers and switch blocks, providing quick and infinite reprogrammability. The aggregate of all the SRAM bits required to program an entire FPGA is called the configuration memory.

A typical FPGA logic block consists of a number of lookup tables into which boolean equations are mapped, and a state holding element, such as a flip-flop. An n-input lookup table (LUT), or an n-LUT, is simply a 2^n memory that is programmed

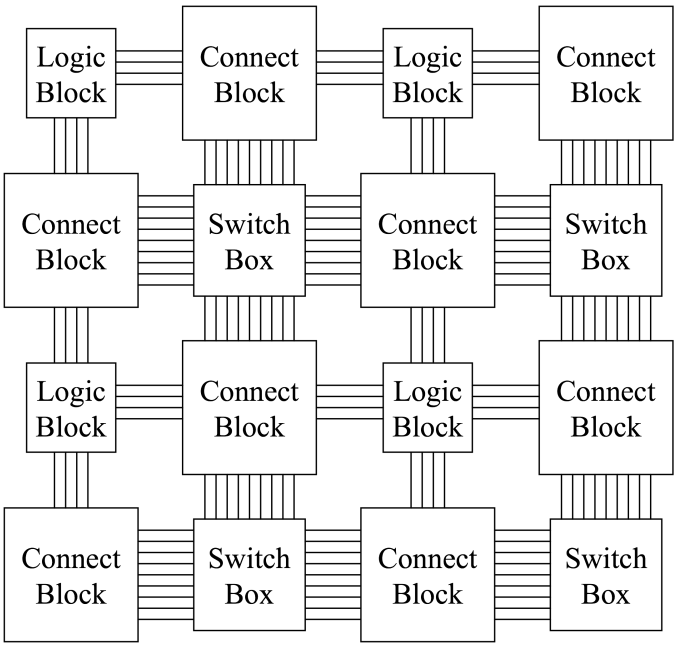

Figure 2.1: Typical island-style FPGA routing structure

with a truth table for the desired function. The use of lookup tables ensures generality, as any combinational digital circuit can be decomposed into a series of boolean equations that can then be expressed as a truth table. The presence of a state holding element allows the FPGA to implement sequential circuits as well as function as a rudimentary memory by aggregating several FPGA logic block resources together.

Connecting logic block inputs and outputs together is an interconnection fabric. This network can be simple, allowing only nearest-neighbor connections (Fig. 2.2), or more complex, with multiple switched routing channels and a hierarchical structure (Fig. 2.1, adapted from [24]).

In the more complex routing structure shown in Fig. 2.1, the inputs and outputs of logic blocks are connected to the general routing fabric through a connection block. This connection block utilizes programmable switch points to connect a particular input/output node of the logic block to a routing channel within the interconnection fabric. The switch boxes then allow the signals to travel to different parts of the

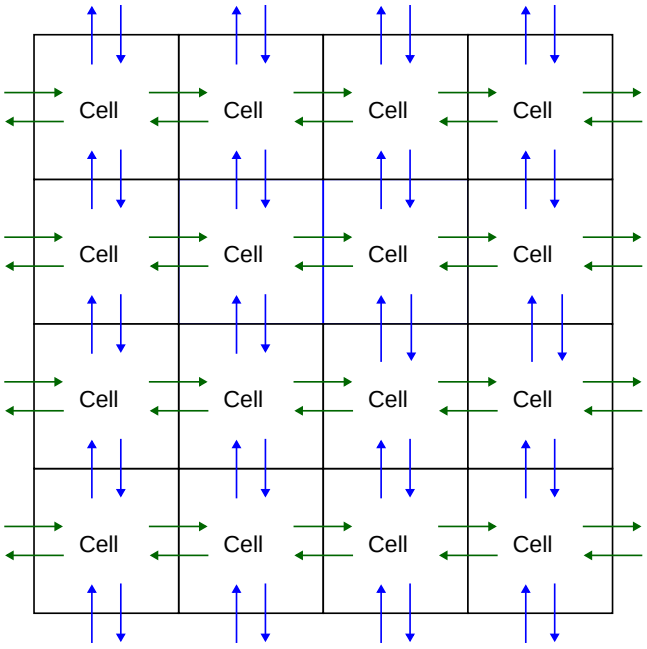

Figure 2.2: Simple nearest-neighbor routing [77]

FPGA, switching between different routing channels and resources, and turning corners. This routing structure is more flexible from an architectural standpoint, and is more typical of commercial FPGAs.

2.2 Case Study: The Xilinx Virtex-II FPGA

The architectural details described thus far have been very simple in nature and do not adequately describe the complexity and enhanced functionality present in modern FPGAs. To give an idea of how these basic models have been built upon by commercial FPGA vendors, consider the Xilinx Virtex-II [81], a good example of a modern island-style FPGA. The Virtex-II architecture is a lookup table-based FPGA utilizing a hierarchical interconnect structure. While the basic functional unit remains a lookup table, it is often useful to group logic and routing resources into hierarchical levels for abstraction purposes.

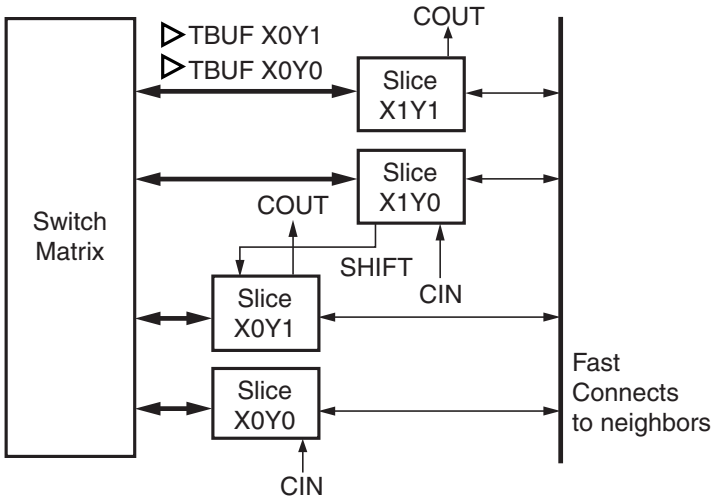

Figure 2.3: Xilinx Virtex-II Configurable Logic Block (CLB) [81]

2.2.1 Virtex-II Logic Resources

The largest hierarchical block is the Xilinx Configurable Logic Block (CLB), shown in Fig. 2.3. This block is the basic logic tile or logic cell of the Virtex-II FPGA. Several of these CLBs are organized in an array to form the core of the logic resources of the device. Tied to a switch matrix to access the general routing fabric, each CLB consists of four smaller "slices" with fast local routing.

Each Virtex-II slice (Fig. 2.4) contains two 4-input lookup tables, miscellaneous gates and multiplexers, two storage elements, and fast carry logic. As the diagram illustrates, the lookup tables can be configured and accessed in three different ways, including: 4-input LUT, 16 bits of memory, or a 16-bit variable-tap shift register element. The extra multiplexers (MUXFx and MUXF5 in Fig. 2.4) allow a single slice to be configured for wide logic functions of up to eight inputs. A handful of extra gates are present in each slice to provide additional functionality. These include: an XOR gate to allow a 2-bit full adder to be implemented within a single slice, an AND gate to improve multiplier implementations, and an OR gate to facilitate efficient

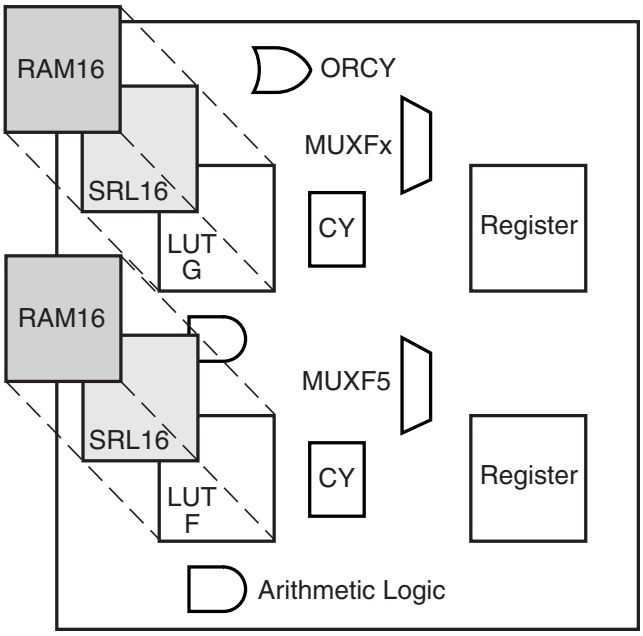

Figure 2.4: Xilinx Virtex-II Slice [81]

implementation of sum-of-products chains. Finally, the storage elements in each slice can be configured as either edge-triggered D-type flip-flops or level-sensitive latches.

2.2.2 Virtex-II Routing Resources

Connecting the CLB logic resources together is a hierarchical interconnect structure. These routing resources are located in horizontal and vertical routing channels between each switch matrix as depicted in Fig. 2.5, with details of the routing structure removed for clarity. The CLBs access the general routing fabric through the "Switch Matrix" boxes in Fig. 2.5. The blocks annotated IOB, DCM, and SelectRAM are beyond the scope of this dissertation, with more details being available in [81]. The actual nature of the interconnection fabric is shown in more detail in Fig. 2.6.

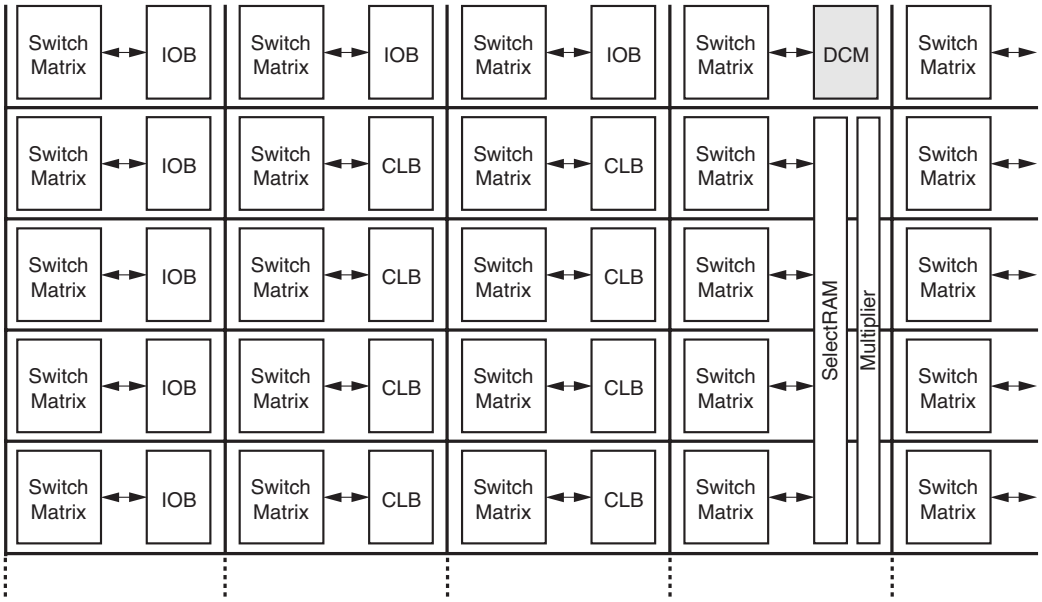

Figure 2.5: Xilinx Virtex-II global routing channels [81]

Much more flexible than the nearest-neighbor routing structure previously depicted in Fig. 2.2, the Virtex-II has a rich, hierarchical routing structure. The routing resources shown in Fig. 2.6 consist of:

- Long lines that span the full height and width of the device
- Hex lines that route to every third or sixth block away in all four directions
- Double lines that route to every first or second block away in all four directions
- Direct connect lines that route to all immediate neighbors
- Fast connect lines internal to the CLB, routing LUT outputs to LUT inputs

Finally, the Virtex-II has a number of routing resources dedicated to global clock nets, on-chip buses, fast carry chains, sum-of-products chains, and shift-chains. These dedicated resources free the general routing fabric from these more specific tasks.

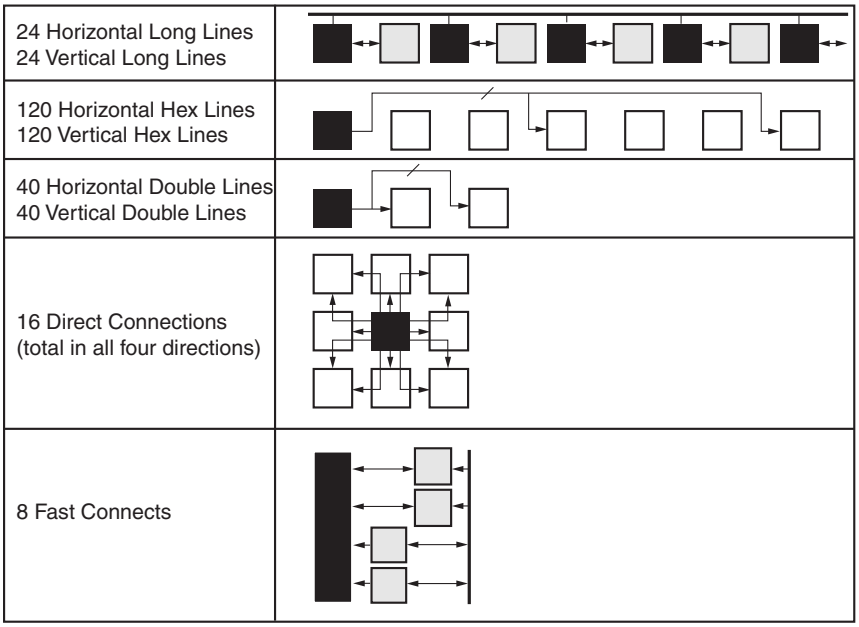

Figure 2.6: Xilinx Virtex-II routing detail [81]

In this brief description of the Virtex-II, several details have been omitted that are beyond the scope of this dissertation. More information is available in [81].

2.3 Performance Potential of FPGAs

Motivating the use of FPGAs in a wide variety of applications is the potential for great speedup compared to software implementations at a relatively low cost when compared to ASIC implementations. Like an ASIC, FPGA designs are built to exactly match the computational requirements of an algorithm. These designs do not have the overhead of performing instruction fetch and decode as in a processor, and can exploit simple, regular, and parallel computations to yield high performance. Unlike an ASIC, each FPGA design does not need to be separately fabricated. Rather, the FPGA is reconfigured with the design of a new algorithm, greatly reducing fabrication delay

and costs. Some examples of how FPGAs have been used to yield high performance are given in this section.

An image classification algorithm was accelerated by a factor of 16 versus an HP Unix workstation in [15, 16]. The SPIHT algorithm, a wavelet coding scheme, implemented on an FPGA platform in [36, 37], achieved a speedup of over 450 times versus a Sun Sparcstation 5 software implementation. Encryption algorithms are also good candidates for acceleration. In [30] the authors show that an FPGA implementation of the Rijndael encryption algorithm achieved an 11.9x speedup versus a 500MHz software implementation described in [39]. Code breaking can also benefit from FPGA devices as [72] demonstrates performance approximately 60 times that of a 1.5GHz Pentium 4 for an RC4 key search. Finally, the authors of [21] implement a two-dimensional finite-difference time-domain (FDTD) algorithm to model electromagnetic space and obtain a speedup of 24 compared to a 3.0GHz Pentium 4.

FPGAs have clearly demonstrated the ability to outperform their software and general-purpose processor counterparts on a wide variety of algorithmic classes. They often do so at a low cost in terms of price as well as power consumption compared to a workstation. As with custom hardware and ASICs, the larger hurdle to their adoption is the development cycle.

2.4 Developing for FPGAs

With the hardware in place, the motivation clear, and the algorithm at hand, developers need to create programs that can take advantage of FPGA resources. As the FPGA is programmed through the setting of a configuration memory, the actual programming information is simply a series of bits that define the configuration memory—commonly referred to as a configuration bitstream.

As an FPGA is a hardware device, the algorithms and applications take the form of circuits. At the lowest level the designer hand maps the desired circuit into the

target hardware. To do this, the designer evaluates each circuit element and its interconnection with other circuit elements, and derives the correct bits to configure the logic blocks and routing resources on the FPGA to implement these circuit elements. These bits together form the bitstream that is then used to program the FPGA. Quite clearly, this manual approach is difficult, time consuming, error prone, and scales poorly for large circuits. This path of development is rarely used now; aside from overwhelming difficulty, it requires intimate knowledge of the exact FPGA architecture, information that remains a closely guarded secret of commercial FPGA vendors.

Fortunately, as with integrated circuit (ASIC) development, tools exist to aid the developer in producing a bitstream with less manual labor and less detailed knowledge of the target hardware. These tools most often utilize a hardware description language (HDL) to specify the circuits to be implemented. A hardware description language is very similar to general-purpose processor programming languages in that it takes a human-readable description of a program and transforms it, through several steps, into something usable by the target platform. Whereas software programming languages such as C/C++ and Java generate a stream of instructions for a processor, the HDL tool chain generates a bitstream for an FPGA.

At the lowest level of abstraction, the developer provides a structural description of the circuit. Structural HDL specifies the target circuit using a library of building blocks supplied by the tool vendor. These building blocks are typically gates and storage elements that may be further abstracted into larger components such as adders and multipliers. While still a very manual approach, this first level of abstraction is one that many developers utilize in order to perform high levels of optimization to best utilize FPGA resources.

Structural HDL tool chains are available for most commercial FPGAs. It is a widely used method for generating efficient implementations as there are fewer levels of abstraction between the circuit description and the actual hardware generated.

The drawback of structural HDL is that it is more difficult to use than methods with higher levels of abstraction, especially when building complex circuits, because every component, input, and output, must be explicitly specified.

Further abstraction is possible through the use of behavioral HDL. Behavioral HDL is closer to conventional general-purpose software development in that it allows the designer to describe the desired program flow using more abstract data types than gate primitives, control and conditional statements, subroutines, and other familiar programming language constructs that define *behavior* as well as structure. Through a process called synthesis, the behavioral description is transformed into an equivalent structural description for further processing.

Up until this point in the development cycle, the specified circuits are technology independent—abstract in the sense that they can use any type of general logic elements. The next step in the tool chain is the technology mapper. The technology mapper transforms a technology independent description of the circuit into a technology dependent one by mapping the general circuit structures onto the specific resources within the fabric of the target FPGA. For LUT-based FPGAs, technology mapping generally consists of partitioning the target circuit into pieces small enough to be mapped into a lookup table. The result of the technology mapper is called a netlist. A netlist simply describes the FPGA resources required and how they are connected.

After mapping is complete, the resultant netlist must be placed and routed on the target FPGA. In the placement phase, each FPGA logic resource listed in the technology-dependent netlist is assigned to a specific physical FPGA resource. After placement of all the elements of the netlist, the routing phase determines exactly which physical routing resources are used to route the required signals between logic blocks. The particular algorithms and mechanisms by which these two steps are accomplished are beyond the scope of this dissertation. However, the references cited in this chapter provide sufficient background material.

Through the synthesis and place-and-route phases, the behavioral description is transformed into an equivalent circuit that can be run on the target FPGA. The behavioral HDL method requires the least amount of target FPGA hardware knowledge and pushes much of the responsibility of circuit generation onto the vendor-supplied tools. Fortunately, these tools do a reasonable job in achieving high-performance implementations of algorithms in FPGA hardware.

As an aside, an even higher level of abstraction is attainable through the use of compiler technologies that can generate structural and behavioral HDL from higher-level programming languages, such as C and C++. Some examples of these tools are described in [14, 35, 40, 70]; however their details will not be discussed in this dissertation.

Behavioral HDL has gained popularity as the effective number of gates on an FPGA reaches into the millions. With such high gate counts, it is becoming less feasible to use any of the more time-consuming methods described above to do full circuit implementation. Most developers utilize a behavioral specification to get an initial implementation, and use their knowledge of the target FPGA hardware along with structural HDL to optimize subsections to meet performance requirements. In this research, we target users of behavioral HDL for precision analysis of FPGA designs.

This brief introduction to FPGAs and FPGA architectures is sufficient to understand the context of the work in this dissertation. However, a more general and thorough survey of FPGAs and reconfigurable computing can be found in [24].

Chapter 3

PRECISION ANALYSIS

FPGAs are a viable alternative to both software implementations running on workstations and custom hardware in the form of ASICs. FPGAs can achieve hardware-like speeds while remaining reprogrammable, a feature typically reserved for software systems. While FPGAs are most certainly a high-performance implementation medium for many applications, developing and prototyping on a software platform remains simpler and less time-consuming. In many cases, hardware use is simply a necessity due to constraints that software platforms cannot meet.

3.1 The Hardware/Software Divide

Often times, algorithms and applications are prototyped in software in a high-level design environment such as MATLAB, Simulink, Java, or C/C++. These environments are comfortable, familiar, and provide a wealth of tools—high-level design tools, editors, compilers, performance analysis tools, automatic optimizers, debuggers, etc.—to aid in the development cycle. When an application fails to meet a particular constraint, such as power consumption, deployment cost, size, or speed, a hardware implementation should be considered. For example, DVD video encoding in MPEG-2 may perform the encoding procedure in real time on a workstation. Even if the workstation were fast enough to perform the encoding, it might be cost-prohibitive to sell a solution based on a software algorithm running on a high-performance workstation. Aside from cost, power and space requirements of the workstation solution might be considered unreasonable for an embedded application which was meant to function

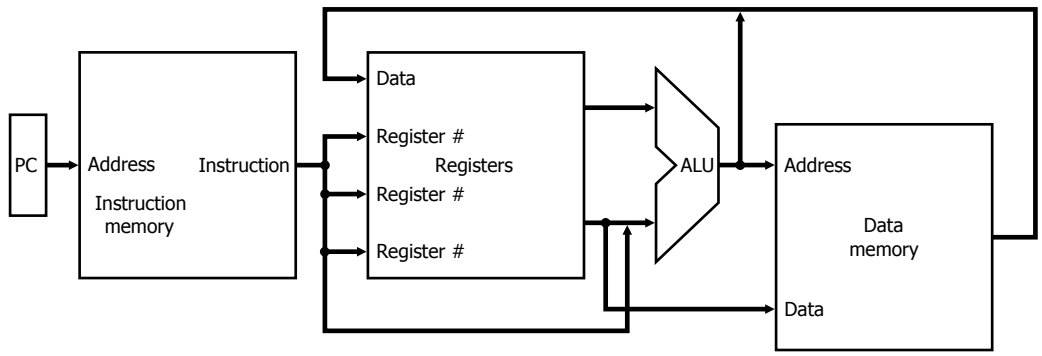

Figure 3.1: Simple GPP model with fixed-width data paths [42]

in a consumer's living room. For these reasons and many more, developers turn to hardware-based solutions.

In transitioning to hardware devices, it becomes difficult to find software developers that possess equivalent skills for hardware development. Therefore, when an algorithm or application must be implemented in a hardware device, the software developers typically relinquish control to hardware developers. Consequently, people who understand the algorithm best do not perform the implementation, while the hardware implementation is performed by individuals with the least involvement in developing the algorithm or application. This knowledge divide often leads to sub-optimal implementations.

With the advent of FPGAs, the promise of easier-to-use hardware is only partially realized. While hardware description languages and tools exist to aid in hardware development, and even though FPGAs allow for hardware implementation without the need or cost of custom silicon fabrication, hardware development essentially still requires hardware developers. This is partly due to the fundamental differences in the underlying computational structures between software and hardware execution.

3.2 Paradigm Shift

When developing software, the computational device commonly targeted is the general-purpose processor (GPP). A very simple model of a GPP is depicted in Fig. 3.1. A typical GPP contains fixed-width data buses and fixed-width computational units, such as an arithmetic logic unit (ALU). The general-purpose processor is designed to perform operations at the world level. The term "word level" refers to a quantization of storage elements into sizes of, typically, 8, 16, 32, and 64 bits. Supporting this paradigm, programming languages and compilers for general-purpose processors provide data type primitives that abstract these quantizations into storage classes, such as `char`, `int`, and `float`. These data types, along with a supporting compiler, abstract away the complications of using differently-sized storage elements in the fixed-width computational structure of a GPP, such as type compatibility, boundary alignment for arithmetic operations, packing and unpacking of large storage classes, and others.

The presence of these abstractions has the effect of allowing software developers to neatly ignore the problem of precision entirely. There is little benefit to using a narrower data type when possible—such as choosing the 16-bit `short` data type instead of the larger 32-bit `int` data type for a variable that doesn't require a large dynamic range—other than the possibility of a marginally reduced memory footprint and faster execution, both of which are highly dependent upon processor architecture and compiler actions. Ensuring correctness, however, through protecting against overflow and underflow is often more important, and therefore using larger-than-necessary data types is a common practice. This paradigm is even more prevalent in higher-level programming languages such as MATLAB [56, 57], where specifying data types is optional, and the assumed data type is the largest available—double-precision floating point, typically 64-bits in width. This mindset of worst-case data type selection is arguably prevalent in software designs.

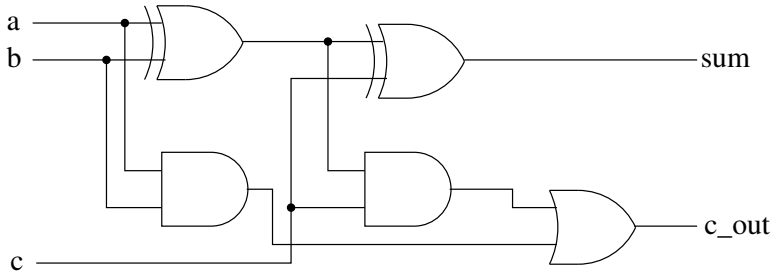

Figure 3.2: A single bit slice of a full-adder

In contrast, hardware devices such as an FPGA are bit-level devices that require bit-exact representations of the algorithm in circuit form. Therefore, if the algorithm requires a full-adder, a circuit such as the one shown in Fig. 3.2 needs to be constructed from FPGA resources. If a four-bit adder is required, it must be explicitly constructed from one-bit full-adders as in Fig. 3.3. Of course, this is just one way to implement adders in hardware and is meant only as an illustrative example. The strength of FPGAs and custom hardware is that the developer can tune the data paths to any word size desired, meeting the exact requirements of the algorithm. Unfortunately, this strength also exposes a difficult hurdle in hardware design: in order to create efficient circuits, the data paths *must* be tuned to match the algorithm, which is not a trivial task.

3.3 Data Path Optimization

If a naïve developer chose to emulate a general-purpose processor by selecting a wide data path, such as 32 bits, the implementation might be very wasteful in terms of hardware resources. The actual data propagating through the data path may never require all the bits allocated. In this scenario, the extra data path bits programmed into the FPGA structure would be wasted, and area would be consumed unnecessarily. On the other hand, if the designer implements the data path using too narrow of a bit

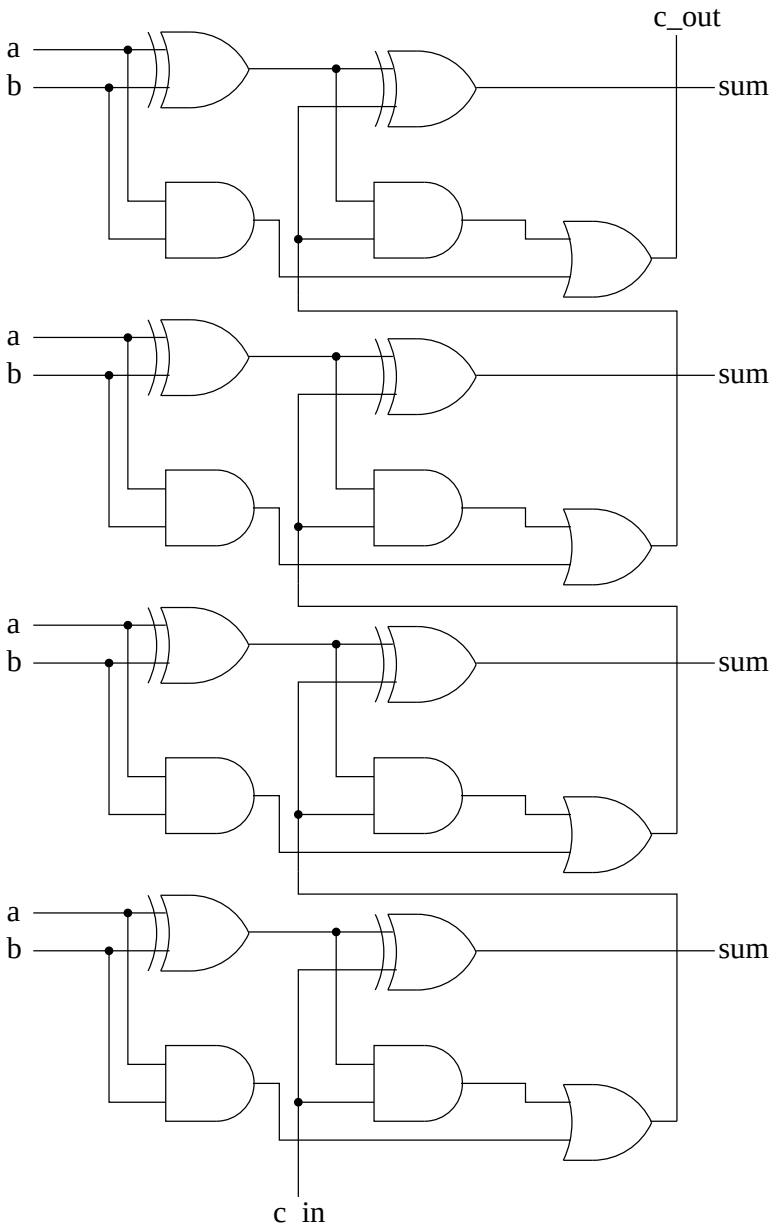

Figure 3.3: A four-bit adder built from four one-bit full-adders

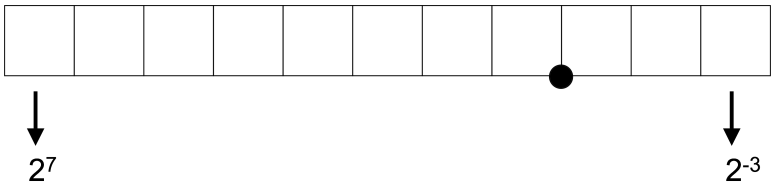

Figure 3.4: Example fixed-point data path

width, unacceptable levels of error may be introduced at run time through excessive quantization effects attributed to roundoff and/or overflow.

In determining the fixed-point representation of a floating-point data path, 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 data path may represent, otherwise 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 data path may attain. For example, as shown in Fig. 3.4, if the most-significant bit is at the 2^7 bit position, and the least-significant bit is at the 2^{-3} bit position, the maximum value attainable by an unsigned number will be 255.875, while the precision will be quantized into multiples of $2^{-3} = 0.125$. Values smaller than 0.125 cannot be represented in this example data path as the bits necessary to represent, for example, 0.0625, do not exist.

Having a fixed-point data path leads to results that may exhibit some quantity of error compared to their floating-point counterparts. This quantization error can be introduced in both the most-significant and least-significant sides of the data path. If the value of an operation is larger than the maximum value than can be represented by the data path, 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 data path if the value requires greater precision than the data path can represent, resulting in truncation or round-off error. The obvious

tradeoff to using a narrower data path is that the area impact of data path operators is reduced, yielding smaller and perhaps faster circuits while incurring some error in the computation.

The developer must tune the width of the data path for an acceptable amount of error while maintaining an area efficient implementation. The decision of data path size is one that has a great impact on overall circuit performance. This optimization is difficult due to the intricate dependencies between data path operators. Since each data path modification is likely to impact other data path operators in terms of area and error, the optimization space is large.

3.4 A Design Time Problem

The developer is faced with a critical system-level design problem: for each signal and each computation, how wide (or narrow) must the data paths be in order to conserve area and achieve performance constraints while maintaining an acceptable level of quantization error? This question is not often raised in software algorithm development due to the neat abstractions afford by double-precision floating-point arithmetic. However, in hardware development, every bit along the data path has an impact in terms of area and error. Therefore, careful thought must be given to the conversion of floating-point data paths into fixed-point ones when implementing algorithms in hardware.

This multi-faceted problem traditionally had very little hardware vendor tool support. The tools supplied by the FPGA chip vendors do not provide any means to analyze the precision requirements of an algorithm, nor do they allow the developer to determine the effects of varying the data path precision in terms of performance and correctness in an automated fashion.

The goal of this research is to fill the gap in design-time tools. We seek to provide methodologies for precision analysis that would be useful to a developer at *design time*.

These tools would guide the developer during the data path precision optimization process.

3.5 Previous Research

The related work in this area can be loosely grouped into analytical approaches, simulation-based approaches, and a hybrid of the two techniques. The approaches can be further categorized in at least three major ways. One is by the amount of user interaction required to perform an analysis and optimization of the data path; second is the amount of feedback they provide to the user for their own manual data path optimization; and third, the level of automation they achieve.

3.5.1 Analytical Approaches

M. Stephenson, et. al., introduce a compiler-based analytical approach to data path optimization in [65, 66]. The authors implemented their compiler, *Bitwise* within the SUIF compiler infrastructure [75], yielding a completely automatic approach to bitwidth analysis. This frees the programmer from performing any manual analysis. The authors have implemented various techniques to deduce bit widths within standard C programs. These techniques include:

- Data-range propagation, forward and backward, through the program's control flow graph using a set of transfer functions for support operators.
- Sophisticated loop-handling techniques using closed-form solutions pioneered in [38] to facilitate propagating through loop structures.
- Integration with the *DeepC Silicon Compiler* [5] to apply *Bitwise* to hardware designs.

Together, the *Bitwise* compiler reduced logic area by 15-86%, improved clock speed by 3-245%, and reduced power by 46-73% on a variety of benchmarks. This work

differs from the work presented here in that it is a purely analytical approach, while our work is a hybrid of techniques. Our work allows developers to interactively analyze the tradeoffs between area and error when performing a hardware implementation of a software-prototyped algorithm. This user-interactivity can allow for more aggressive optimization than is possible in a purely analytical approach and is arguably better suited to the naturally iterative nature of hardware development.

A similar approach is used by S. Mahlke, et. al. [55]. These researchers utilized bitwidth analysis to optimize the hardware cost of synthesizing custom hardware from loop nests specified in the C programming language. The authors acknowledge that their work is of narrower scope than that of M. Stephenson, et. al., cited previously. The bitwidth analysis in [55] consists of forward and reverse propagation of bitwidth information performed upon an assembly-level representation of a C-language program loop. This restriction alleviates S. Mahlke, et. al., from having to perform many of the more complex analyses detailed in [65, 66], such as loop traversal and pointer following. The results of this work show an average total cost reduction of 50% when compared to hardware synthesis without bitwidth optimization. As with the work presented by M. Stephenson, et. al., [55] represents a purely analytical approach to bitwidth analysis that does not incorporate any user-interactive optimization. The work described in this dissertation is a hybrid of techniques, both analytical and simulation based, which leverages user knowledge at design time to aid in the optimization of data path widths.

A third analytical approach taken by A. Nayak is detailed in [59]. It is similar to our own work in that it utilizes MATLAB as an input algorithm specification and attempts to handle the position of both the most-significant and least-significant bits. Their approach is to first mimic the value-range propagation of M. Stephenson [65] to determine the position of the most-significant bit. This phase is followed by an error analysis phase to find the level of precision needed in the data path and subsequently the position of the least-significant bit.

For the error analysis phase in A. Nayak's work, the user can either specify the amount of tolerable error in the pixels of the output image, or a default resolution of 4 fractional bits will be assumed. This information is backwards-propagated from the output node (assumed or explicitly defined) to obtain the required number of fractional bits at each intermediate node in the graph.

The results show an average of five times fewer FPGA resources required and 35% faster execution after optimization compared to unoptimized hardware. This work differs from our own in that it is, again, a purely analytical approach that makes several assumptions on the type of algorithm being implemented (image/signal processing). There is no opportunity for manual intervention for further or more aggressive optimization, nor is there any hint from the system as to alternative implementations or less aggressive optimizations.

Finally, Constantinides, et. al. [27, 28], concentrate on developing algorithms for nearly fully-automatic wordlength optimization. G. Constantinides introduces the optimization of linear time-invariant (LTI) systems through the use of error modeling. The optimization is first performed as a Mixed-Integer Nonlinear Programming (MINP) problem in order to find the optimal single wordlength for all signals in the system given an error constraint specified by the user while minimizing area estimated using models. This phase is followed by a heuristic algorithm that further refines the individual wordlengths based on a user-specified "goodness" function, typically signal-to-noise ratio on the output. The approach is to greedily reduce candidate signal word lengths by one bit until they violate the given output noise constraint. This creates a very automated system that unfortunately is limited to only LTI systems, a small class of problems. The authors demonstrate a 45% area reduction and up to 39% speed increase on selected DSP benchmarks. The work presented in [25] extends their previous efforts by using perturbation analysis to linearize non-linear data path operators. This allows the techniques described in [27, 28] to be used on non-LTI sys-

tems. Additionally, the authors show the effect of wordlength optimization on power reduction.

This work differs from our own in that it, like other previous work, does not consider much of the user's expertise within the optimization cycle. The only piece of information the user can supply is the output error constraint. There is no mechanism for the developer to reveal the area and error effects of modifying individual signals and operators.

3.5.2 Simulation-Based Approaches

Departing from purely analytical approaches are the solely simulation-based approaches. Sung, et. al. [45, 46, 67] introduced a method and tool for word-length optimization targeting custom VLSI implementations of digital signal processing algorithms. The initial work [50] from the group utilized an internal and proprietary VHDL-based simulation environment. This software was subsequently released as a commercial tool, "Fixed-Point Optimizer" [2, 68]. and further developed to target digital signal processing algorithms written in C/C++ in [45, 46, 67].

Sung and Kim's work consists of two utilities, a *range estimator* and the *fixed-point simulator*. The range estimator utility determines statistical information of internal signals through floating-point simulation with real (assumed typical) inputs. The fixed-point simulator converts a floating-point program into a fixed-point equivalent through the use of a fixed-point data class.

In discussion regarding the range estimation utility, the authors note that analytical methods used to determine the range of intermediate variables in a data path produce very conservative estimates. Additionally, analytical methods are difficult to use with adaptive or non-linear systems. Thus, the authors choose to focus on a simulation-based method to estimate the ranges during actual operation of the algorithm using realistic input data. In order to maintain functional equivalence with the original floating-point C or C++ algorithm, a new data class, **fSig** is introduced

that can track variable statistics. By simply replacing **float** types with **fSig** types in the original program, one can gather range information as easily as compiling and running the candidate program.

The statistics gathered through the use of this new data class include the summation of past values, the square of the summation of past values, the absolute maximum value (**AMax**), and the number of modifications during simulation. After the simulation is complete, the mean (μ) and the standard deviation (σ) are calculated from the stored statistics and the statistical range of a **fSig** variable x can be estimated as (taken from [45]):

$$R(x) = max\{|\mu(x)| + n * \sigma(x), \mathbf{AMax}\}, \tag{3.1}$$

where n is a user specified value, typically from 4 to 16. This value of n governs the aggressiveness of the estimation function, where larger values lead to a more conservative estimate. As this estimation takes into account the mean value of the variable as well as the standard deviation, it describes a range of values that the variable will most likely fall within during the lifetime of the simulation. With the presence of **AMax**, the estimation will encompass any outliers in terms of maximum absolute value.

With the range estimator yielding a picture of how the algorithm with typical input data behaves at each intermediate variable, it is left to the fixed-point simulation utility to model the algorithm operating in a fixed-point environment. This is done through the same mechanism as the range estimator: creating a new data class, **gFix** that encapsulates how values behave in a fixed-point sense. Using information gathered with the range estimator, word lengths can be explicitly set for each **gFix** variable and a fixed-point simulation can be performed to gauge the impact of roundoff and truncation error.

While the work described in [45, 46] yielded interesting C and C++ libraries, it differs from the work presented here as it has no notion of automatic or assisted optimization, nor does it have any methods for optimizing while minimizing hardware

costs. It was not until [67] that any cost model was integrated to facilitate automation of word length optimization.

In [67] a reference system is designed to be a correctness benchmark. Second, a fixed-point performance measure is developed that can quantify the fixed-point effects. An example they give of one such measure is a signal-to-quantization-noise ratio (SQNR) block. This block is designed such that it returns a positive value when the quantization effects on the output are within acceptable limits.

Combined with hardware estimation models from a commercial VLSI standard cell library, this optimization method seeks to automatically minimize hardware cost while maintaining a usable implementation. While the current word-lengths of the system do not satisfy the system performance required by the evaluation block, word-lengths are increased. The word-lengths which increase are the subject of either an exhaustive or simple heuristic search through all signals in the system, given hardware cost estimates.

The results on two benchmarks show that the system can effectively reduce the total gate count cost while meeting the requirements imposed by the fixed-point performance measure block. Kim and Sung's work employs a fully automated approach, while the work presented in this dissertation uses user-interactive as well as fully-automated approaches. No effort is made to include the developer in the optimization decision loop. How the system evaluates the correctness is automated through the creation of a "goodness function" block created by the developer. This single evaluation does not allow for a comprehensive exploration of the tradeoffs between hardware cost and quantization error, as, in many cases, the quality of a result and the gate cost of the implementation are inseparable metrics.

3.5.3 Hybrid Approaches

Seeing the deficiencies in using only one approach, several researchers have used a hybrid approach to tackling the problem of data path optimization.

In [74], W. Willems, et. al. summarize the FRIDGE project. The authors present a tool that allows for fixed-point design utilizing an interpolative approach—essentially blending the analytical and simulation approaches previously discussed. The four-step interpolative approach is:

1. Local annotation
2. Fixed-point simulation
3. Interpolation
4. Fixed-point simulation

For the local annotation, new ANSI-C parameterizable fixed-point data types, *fixed* and *Fixed* are introduced. These types provide a framework to perform fixed-point simulation of the algorithm by simply making changes to the source algorithm. The designer begins with a floating-point program which is then annotated with known fixed-point information (e.g. the inputs and outputs to the system). The information about each variable that can be passed to the simulation and interpolation environments include range, mean, variance, maximum absolute acceptable error, and maximum acceptable relative error. Simulation is then performed to check whether the annotated program operates correctly and meets all design criteria when compared to the floating-point program. If not, modifications to the annotations must be done manually.

If the original annotations are not too aggressive, the simulation will reveal correct behavior and move into the interpolation phase. The interpolation phase is where an analytical approach is utilized. Through estimation using dependency analysis, conditional structure analysis, and loop structure analysis similar in spirit to [65], all remaining floating-point variables are converted into annotated fixed-point data types.

Willems' work is similar to ours in that it is a design time analysis that incorporates some user input. In this case, it is only at the initial local annotation phase that the user has any input. From that point forward, there is little chance besides the final re-simulation phase for the user to override the deductions made during the interpolation phase. Another crucial difference between this work and the work presented in this dissertation is that no suggestions are made to the user outside of the automatic interpolation phase.

Finally, R. Cmar, et. al. [23] provide possibly the closest analog to the work presented in this dissertation. Their work provides a strategy for fixed-point refinement that utilizes both a simulation-based approach as well as an analytical approach. The simulation, as in several other efforts, utilizes C++ overloading and custom libraries. The analytical approach infers wordlengths of variables and operators from source code structure, local annotations, and interpolation. This effort also introduces error monitoring for the least-significant side of the data path. Utilizing the difference between the simulated floating- and fixed-point implementations, errors at each node are quantified and aggregated for each signal output. This makes it easier for the developer to see the effects of quantization error and determine if the precision optimization was too aggressive. This method, like those presented before, does not make any attempt to offer suggestions to the user as to where optimizations should take place.

3.6 Summary

There has been a wealth of work in this area in addition to this contribution. Optimization of data path widths using a range of techniques has been explored. Unfortunately, few approaches offer the developer a way to use their expertise at more than one point in the system. Few easily allow more aggressive optimizations to be profiled in terms of area and error. Finally, the ability to automatically and accurately place

the position of the least-significant bit is something that has been ignored except for the brief treatments in [23, 59].

In the chapters to follow we consider the optimization of the most-significant bit position (Chapter 4) and the least-significant bit position (Chapter 5). Together, these techniques offer a unified approach to design-time, developer-centric data path optimization.

Chapter 4

MOST-SIGNIFICANT BIT OPTIMIZATION

4.1 Designer-centric Automation

We begin our discussion of data path optimization with techniques for most-significant bit position optimization. As previously mentioned, much of the existing research focuses on fully-automated optimization techniques. While these methods can achieve good results, it is our belief that the developer should be kept close at hand during all design phases, as they possess key information that an automatic optimization methodology simply cannot account for or deduce.

In order to guide an automatic precision optimization tool, a goodness function must be used to evaluate the performance of any optimization steps. In some cases, such as two-dimensional image processing, a simple signal-to-noise ratio (SNR) may be an appropriate goodness function. In other cases, the goodness function may be significantly more complex and therefore more difficult to develop. In either case, the developer still has the burden of implementing a goodness function within the framework of the automatic optimization tool.

By simulating a human developer's evaluation of what is an appropriate tradeoff between quality of result and hardware cost, the automatic optimization tool loses a crucial resource: the knowledgeable developer's greater sense of context in performing a goodness evaluation. Not only is this valuable resource lost, for many classes of applications an automatically evaluated goodness function may be difficult or even impossible to implement. In other words, for many applications, a knowledgeable developer may be the best, and perhaps only, way to guide precision optimization.

Therefore, there are many instances where a fully-automatic precision optimization tool should not or cannot be used.

In a departure from previous work utilizing fully-automatic methods, we approach this problem by providing a "design-time" precision analysis tool that interacts with the developer to guide the optimization of the hardware data path.

4.2 Optimization Questions

In performing manual data path optimization, one finds that the typical sequence of steps requires answering four questions regarding the algorithm and implementation:

1. What are the provable precision requirements of the algorithm?

2. What are the effects of fixed-precision on the results?

3. What are the actual precision requirements of the data sets?

4. Where along the data path should optimizations be performed?

By repeatedly asking and answering these questions, hardware designers can narrow the data paths within their circuits. The tradeoffs between area consumption and accumulated error within the computation need to be manually analyzed—a time consuming and error prone process with little tool support.

In order to fill the gap in *design-time* tools that can aid in answering these precision questions, we introduce our prototyping tool, Précis. Algorithms written in the MATLAB language serve as input to Précis. MATLAB is a very high-level programming language and prototyping environment popular in the signal and image processing communities. More than just a language specification, MATLAB [56, 57] is an interactive tool that allows developers to manipulate algorithms and data sets

to quickly see the impact of changes on algorithm output. The ease with which developers can explore the design space of their algorithms makes it a natural choice to pair with Précis to provide a design-time precision analysis environment.

Précis aids developers by automating many of the more mundane and error-prone tasks necessary to answer the four precision analysis questions. This is done by providing several integrated tools within a single application framework, including: a constraint propagation engine, MATLAB simulation support, variable range gathering support, and a slack analysis phase. It is designed to complement the existing tool flow at *design time*, coupling with the algorithm before it is translated into an HDL description and pushed through the vendor back-end bitstream generation tools. It is designed to provide a convenient way for the user to interact with the algorithm under consideration. The goal is for the knowledgeable user, after interacting with our tool and the algorithm, to have a much clearer idea of the precision requirements of the data paths within their algorithm.

4.3 Précis

The front-end of Précis comes from Northwestern University in the form of a modified MATCH compiler [6]. The MATCH compiler understands a subset of the MATLAB language and can transform it into efficient implementations on FPGAs, DSPs, and embedded CPUs. It is used here primarily as a pre-processor to parse MATLAB codes. The MATCH compiler was chosen as the basis for the MATLAB code parsing because no official grammar is publicly available for MATLAB. The tool is not constrained to using the MATCH compiler, though, and may be updated to accommodate an alternate MATLAB-aware parser.

The main Précis application is written in Java, in part, due to its relative platform independence and ease of graphical user interface creation. Précis takes the parsed MATLAB code output generated from the MATCH compiler and displays a GUI that

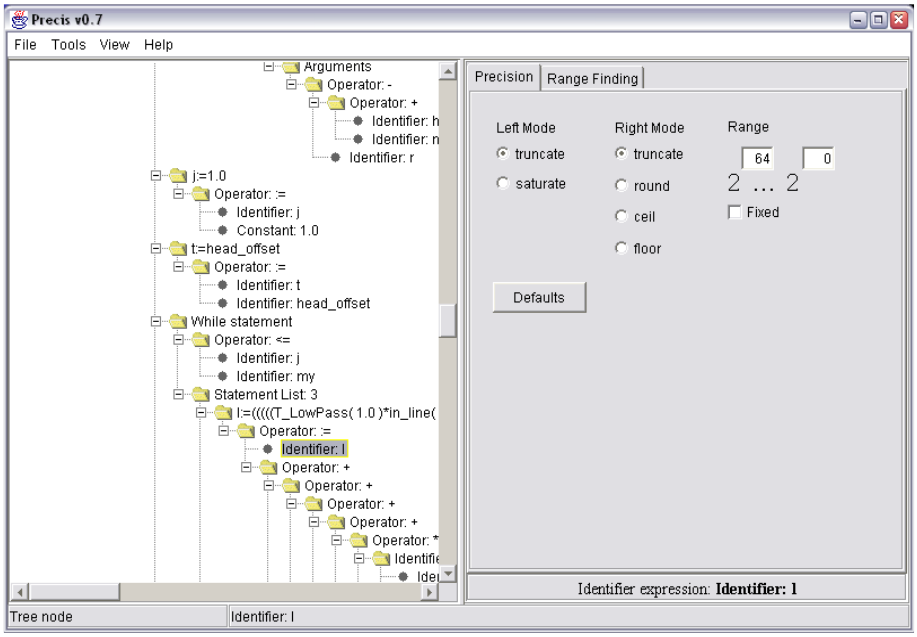

Figure 4.1: Précis screenshot

formats the code into a tree-like representation of statements and expressions. An example of the GUI in operation is shown in Fig. 4.1. The left half of the interface is the tree representation of the MATLAB code. The user may click on any node and, depending on the node type, receive more information in the right panel. The right panel displayed in the figure is an example of the entry dialog that allows the user to specify fixed-point precision parameters, such as range and type of truncation. With this graphical display the user can perform the tasks described in the following sections.

4.4 Propagation Engine

A core component of the Précis tool is a constraint propagation engine. The purpose of the constraint propagation engine is to answer the first of the four precision-analysis questions: *what are the provable precision requirements of my algorithm?* By learning

how the data path of the algorithm under question grows in a worst-case sense, we can obtain a baseline for further optimization as well as easily pinpoint regions of interest—such as areas that grow quickly in data path width—which may be important to highlight to the user.

The propagation engine works by modeling the effects of using fixed-point numbers and fixed-point math in hardware. This is done by allowing the user to (optionally) constrain variables to a specific precision by specifying the bit positions of the most significant bit (MSB) and least significant bit (LSB). Variables that are not manually constrained begin with a default width of 64 bits. This default width is chosen because it is the width of a double-precision floating-point number, the base number format used in the MATLAB environment. It is important to note that a 64-bit fixed-point value has a much narrower dynamic range than its 64-bit floating-point counterpart. However, a default width must be chosen for variables not annotated by the user. Typically, a user should be able to provide constraints easily for at least the circuit inputs and outputs.

The propagation engine traverses the expression tree and determines the resultant ranges of each operator expression from its child expressions. This is done by implementing a set of rules governing the change in resultant range that depend upon the input operand(s) range(s) and the type of operation being performed. For example, in the statement $a = b + c$, if b and c are both constrained by the user to a MSB position of 2^{15} and a LSB position of 2^0, 16 bits, the resulting output range of variable a would have a range of 2^{16} to 2^0, 17 bits, as an addition conservatively requires one additional high order bit for the result in the case of a carry-out from the highest order bit. Similar rules apply for all supported operations.

The propagation engine works in this fashion across all statements of the program, recursively computing the precision for all expressions in the program. This form of propagation is often referred to as value-range propagation. An example of forward and backward propagation is depicted in Fig. 4.2.

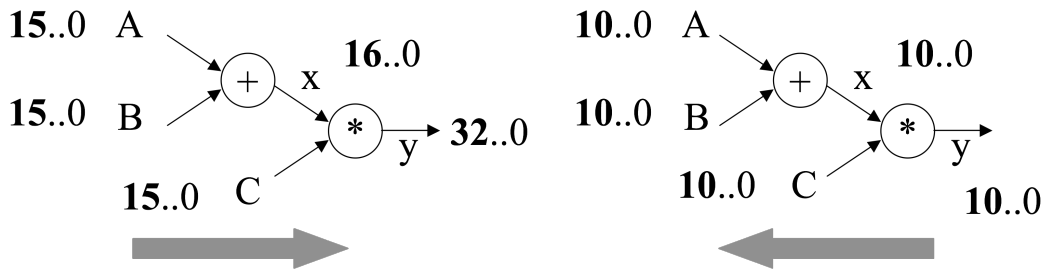

Figure 4.2: Simple propagation example

In this example, assume the user sets all input values (a, b, c) to utilize the bits [15,0], resulting in a range from $2^{16} - 1$ to 0. Forward propagation would result in x having a bit range of [16, 0] and c having a range of [31, 0]. If, after further manual analysis, the user notes that the output from these statements should be constrained to a range of [10, 0], backwards propagation following forward propagation will constrain the inputs (c and x) of the multiplication to [10, 0] as well. Propagating yet further, this constrains the input variables a and b to the range [10, 0] as well.

The propagation engine is used to get a quick, macro-scale estimate of the growth rate of variables through the algorithm. This is done by constraining the precision of input variables and a few operators and performing the propagation. This allows the user to see a conservative estimate of how the input bit width affects the size of operations downstream. While the propagation engine provides some important insight into the effects of fixed-point operations on the resultant data path, it forms a conservative estimate. For example, in an addition, the propagation engine assumes that the operation requires the carry-out bit to be set. It would be appropriate to consider the data path widths determined from the propagation engine to be worst-case results, or in other words, an upper bound. This upper bound, as well as the propagation engine, will become useful in further analysis phases of Précis.

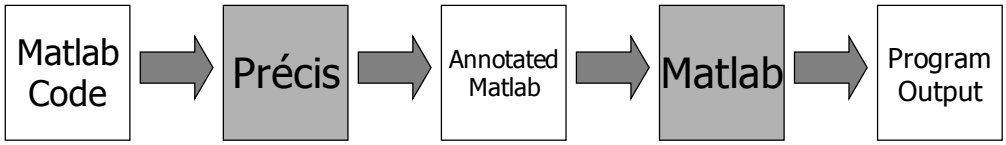

Figure 4.3: Flow for code generation for simulation

4.5 Simulation Support

To answer the second question during manual precision analysis: "*what are the effects of fixed-precision on my results?*" the algorithm needs to be operated in a fixed-point environment. This is often done on a trial-and-error basis, as there are few high-level fixed-point environments. To aid in performing fixed-point simulation, Précis automatically produces annotated MATLAB code. The developer simply selects variables to constrain and requests that MATLAB simulation code be generated. The code generated by the tool includes calls to MATLAB helper functions that were developed to simulate a fixed-point environment, alleviating the need for the developer to construct custom fixed-point blocks. The simulation flow is shown in Fig. 4.3.

In particular, a MATLAB support routine, `fixp` was developed to simulate a fixed-point environment. Its declaration is

```
fixp(x,m,n,lmode,rmode)
```

where `x` denotes the signal to be truncated to $(m - n + 1)$ bits in width. Specifically, m denotes the MSB bit position and n the LSB bit position, inclusively, with negative values representing positions to the right of the decimal point. The remaining two parameters, `lmode` and `rmode` specify the method desired to deal with overflow at the MSB and LSB portions of the variable, respectively. These modes correspond to different methods of hardware implementation. Possible choices for `lmode` are `sat` and `trunc`. Saturation sets the value of the variable to $2^{(MSB+1)} - 1$ while truncation

MATLAB Input	Annotated MATLAB
`a = 1;`	`a=1;`
`b = 2;`	`b=2;`
`c = 3;`	`c=3;`
`d = (a+(b*c));`	`d=(fixpp(a,12,3,'trunc','trunc')+ (b*c));`

Figure 4.4: Sample output generated for simulation, with the range of variable a constrained

removes all bits above the MSB position. For the LSB side of the variable, there are four modes: `round, trunc, ceil`, and `floor`. `Round` rounds the result to the nearest integer, `trunc` truncates all bits below the LSB position, `ceil` rounds up to the next integer level, and `floor` rounds down to the next lower integer level. These modes correspond exactly to the MATLAB functions with the exception of trunc, and thus behave as documented by Mathworks. `Trunc` is accomplished through the MATLAB modulo operation which allows easy truncation of any unwanted higher-order MSB-side bits. An example of output generated for simulation is shown in Fig. 4.4.

After the developer has constrained the variables of interest and indicated the mechanism by which to control overflow of bits beyond the constrained precision, Précis generates annotated MATLAB. The developer can then run the generated MATLAB code with real data sets to determine the effects of constraining variables on the correctness of the implementation.

If the developer finds the algorithm's output to be acceptable, then constraining additional key variables might be considered, further reducing the eventual size of the hardware circuit. On the other hand, if the output generates unusable results, the user knows that the constraints were too aggressive and that the width of the data paths used by some of the constrained variables should be increased.

During this manual phase of precision analysis, it is typically not sufficient to merely test whether the fixed precision results are identical to the unconstrained precision results, since this is likely too restrictive. In situations such as image processing, lossy compression, and speech processing, users may be willing to trade some result quality for a more efficient hardware implementation. Précis, by being a designer assistance tool, allows the designer to create their own "goodness" function, and make this tradeoff as they see fit. With the Précis environment, this iterative development cycle is shortened, as the fixed-point simulation code can be quickly generated and executed, allowing the user to view the results and the impact of error without the tedious editing of algorithm source code.

4.6 Range Finding

While the simulation support described above is very useful on its own for fixed-point simulation, it is much more useful if the user can accurately identify the variables that they feel can be constrained. This leads to the third question that must be answered in order to perform effective data path optimization: "*what are the actual precision requirements of the data sets?*" Précis helps answer this question by providing a range finding capability that helps the user deduce the data path requirements of intermediate nodes whose ranges may not be obvious. The development cycle utilizing range finding is shown in Fig. 4.5.

After the MATLAB code is parsed, the user can select variables they are interested in monitoring. Variables are targeted for range analysis and annotated MATLAB is generated, much like the simulation code is generated in the previous section. Instead of fixed-point simulation, Précis annotates the code with another MATLAB support routine that monitors the values of the variables under question.

This support routine, `rangeFind`, monitors the maximum and minimum values attained by the variables. The annotated MATLAB is run with some sample data

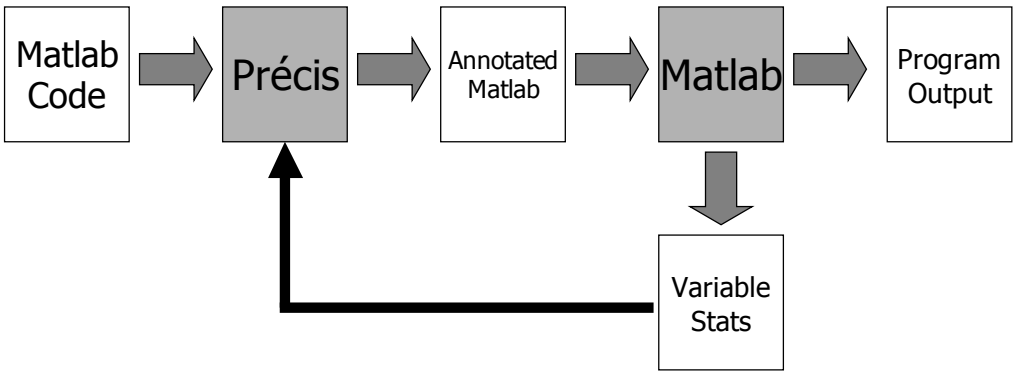

Figure 4.5: Development cycle for range finding analysis

MATLAB Input	Range Finding Output
`a = 1;`	`a=1;`
`b = 2;`	`b=2;`
`c = 3;`	`c=3;`
`d = (a+(b*c));`	`d=(a+(b*c));`
	`rangeFind(d,'rfv_d');`

Figure 4.6: Sample range finding output

sets to gather range information on the variables under consideration. The user can then save these values in data files that can be fed back into Précis for a further analysis phases. An example of the annotated MATLAB is shown in Fig. 4.6.

The developer then loads the resultant range values discovered by `rangeFind` back into the Précis tool and (optionally) constrains the variables. The range finding phase has now given the user an accurate profile of what precision each variable requires for the data sets under test. Propagation can now be performed to conservatively estimate the effect these data path widths have on the rest of the system.

The propagation engine and the range finding tools work closely together to allow the user to build a more comprehensive picture of the precision requirements of the algorithm than either of the tools could do alone. The propagation engine, with user-knowledge of input and perhaps output variable constraints, achieves a first-order estimation of the data path widths of the algorithm. Using the range finding information allows for significant refinement of this estimation; the discovered variable statistics allow for narrower data path widths that more closely reflect the true algorithmic precision requirements.

Another useful step that can be performed is to constrain variables even further than suggested by the range-finding phase. Subsequent simulations are performed to see if the error introduced, if any, is within acceptable limits. These simulations, as before, are easily generated and executed within the Précis framework.

The results from this range finding method are data set dependent. If the user is not careful to use representative data sets, the final hardware implementation could still generate erroneous results if the data sets were significantly different in precision requirements. Data sets that exercise the full expected range of precision (common cases as well as extreme cases) should be used to allow the range finding phase to gather meaningful and robust statistics.

It is useful, therefore, to consider range-gathered precision information to be a lower bound on the precision required by the algorithm. As the data sets run have been observed to exercise a known amount of data path width, any further reduction in the precision will likely incur error. Given that the precisions obtained from the propagation engine are conservative estimates, or an upper bound, manipulating the difference between these two bounds leads us to a novel method of user-guided precision analysis—slack analysis.

4.7 Slack Analysis

One of the goals of this work is to provide the user with "hints" as to where the developer's manual precision analysis and hardware tuning efforts should be focused. This is the subject of the fourth precision analysis question: *where along the data path should I optimize?*. Ultimately, it would be extremely helpful for the developer to be given a list of "tuning points" in decreasing order of potential overall reduction of circuit size. With this information, the developer could start a hardware implementation using more generic data path precision, such as a standard 64 or 32-bit data path, and iteratively optimize code sections that would yield the most benefit. Iteratively optimizing sections of code or hardware is a technique commonly used to efficiently meet constraints such as time, cost, area, performance, or power. We believe this type of "tuning list" would give a developer effective starting points for each iteration of their manual optimization, putting them on the most direct path to meeting their constraints.

Recall that if the developer performs range finding analysis and propagation analysis on the same set of variables, the tool would obtain a lower bound from range analysis and an upper bound from propagation. We consider the range analysis a lower bound because it is the result of true data sets. While other data sets may require even lower amounts of precision, at least the ranges gathered from the range analysis are needed to maintain an error-free output. Further testing with other data sets may show that some variables would require more precision. Thus, if the design is implemented with the precision found, we might encounter errors on output, thus the premise that this is a lower bound.

On the other hand, propagation analysis is very conservative. For example, in the statement $a = b + c$, where b and c have been constrained to be 16 bits wide by the user, the resultant bit width of a may be up to 17 bits due to the addition. But in reality, both b and c may be well below the limits of 16 bits and an addition

might never overflow into the 17th bit position. For example, if $c = \lambda - b$, the range of values a could ever take on is governed by λ. To a person investigating this section of code, this seems very obvious when c is substituted into $a = b + c$, yielding $a = b + \lambda - b$, however these types of more "macroscopic" constraints in algorithms can be difficult or impossible to find automatically. It is because of this that we can consider propagated range information to be an upper bound.

Given a lower and upper bound on the bit width of a variable, we can consider the difference between these two bounds to be the slack. The actual precision requirement is most likely to lie between these two bounds. Manipulating the precision of nodes with slack can yield gains in precision system-wide, as changes in any single node may impact many other nodes within the circuit. The reduction in precision requirements and the resultant improvements in area, power, and performance can be considered gain. Through careful analysis of the slack at a node, we can calculate how much gain can be achieved by manipulating the precision between these two bounds. Additionally, by performing this analysis independently for each node with slack, we can generate an ordered list of "tuning points" that the user should consider when performing manual iterative optimization.

A reduction in the area requirements of a circuit is a gain. In order to compute the gain of a node with respect to area, power and performance, we need to develop basic hardware models to capture the effect of precision changes upon these parameters. For this work a simple area model serves as the main metric. For example, an adder has an area model of x, indicating that as the precision decreases by one bit, the area reduces linearly and the gain increases linearly. In contrast, a multiplier has an area model of x^2, indicating that the area reduction and gain achieved are proportional to the square of the word size. Intuitively, this would give a higher overall gain value for bit reduction of a multiplier than of an adder, which is in line with the implementations that are familiar to hardware designers. Using these parameters,

our approach can effectively choose the nodes with the most possible gain to suggest to the user. We detail our methodology in the next section.

4.8 Performing Slack Analysis

The goal of slack analysis is to identify the nodes that could be constrained by the user that will yield the greatest impact upon the overall circuit area. While we do not believe it is realistic to expect users to constrain all variables, most users would be able to consider how to constrain a few "controlling" values in the circuit.

Our method seeks to optimize designer time by guiding them to the next important variables to consider for constraining. Précis can also provide a stopping criterion for the user: we can measure the maximum possible benefit from future constraints by constraining all variables to their lower bounds. The user can then decide to stop further investigation when the difference between the current and lower bound area is no longer worth further optimization.

Our methodology is straightforward. For each node that has slack, we set the precision for only that node to its range-find value—the lower bound. Then, we propagate the impact of that change over all nodes and calculate the overall gain in terms of area for the change, system-wide. We record this value as the effective gain as a result of modifying that node. We then reset all nodes and repeat the procedure for the remaining nodes that have slack. We then sort the resultant list of gain values in decreasing order and present this information to the user in a dialog window. From the graphical user interface, the user can easily see how and which nodes to modify to achieve the highest gain. It is then up to the designer to consider these nodes and determine which, if any, should actually be more tightly constrained than suggested by Précis. Pseudo-code for the slack analysis procedure is shown in Fig. 4.7.

```
PerformSlackAnalysis
 1 constrain user-specified variables
 2 perform propagation
 3 baseArea ← calculateArea()
 4 load range data for some set of variables n
 5 listOfGains ← ∅
 6 foreach m in n
 7     reset all variables to baseline precision (from line 1)
 8     constrain range of m to the range analysis value
 9     perform one pass of forward then reverse propagation
10     newArea ← calculateArea()
11     if (newArea < baseArea) then
12         listOfGains ← (m, baseArea − newArea)
13 sort listOfGains by decreasing gain
```

Figure 4.7: Slack analysis pseudo-code

4.9 Benchmarks

In order to determine the effectiveness of Précis, we utilized the tool to optimize a variety of image and signal processing benchmarks. To gauge how effective the suggestions were, we constrained the variables the tool suggested in the order they were suggested to us, and calculated the resulting area. The area was determined utilizing the same area model discussed in previous sections, i.e. giving adders a linear area model while multipliers are assigned an area model proportional to the product of their input word sizes. We also determined an asymptotic lower bound to the area by implementing all suggestions simultaneously to determine how quickly our tool would converge upon the lower bound.

4.9.1 Wavelet Transform

The first benchmark we present is the wavelet transform. The wavelet transform is a form of image processing, primarily serving as a transformation prior to applying a compression scheme, such as SPIHT [36]. A typical discrete wavelet transform runs a high-pass filter and low-pass filter over the input image in one dimension. The results are down sampled by a factor of two, effectively spatially compressing the wavelet by a factor of two. The filtering is done in each dimension, vertically and horizontally for images. 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.

This algorithm was hand-mapped to hardware as part of work done by Thomas Fry [36]. Significant time was spent converting the floating-point source algorithm into a fixed-point representation by utilizing methodologies similar to those presented here. The result was an implementation running at 56MHz, capable of compressing 8-bit images at a rate of 800Mbits/sec. This represents a speedup of nearly 450 times as compared to a software implementation running on a Sun SPARCStation 5.

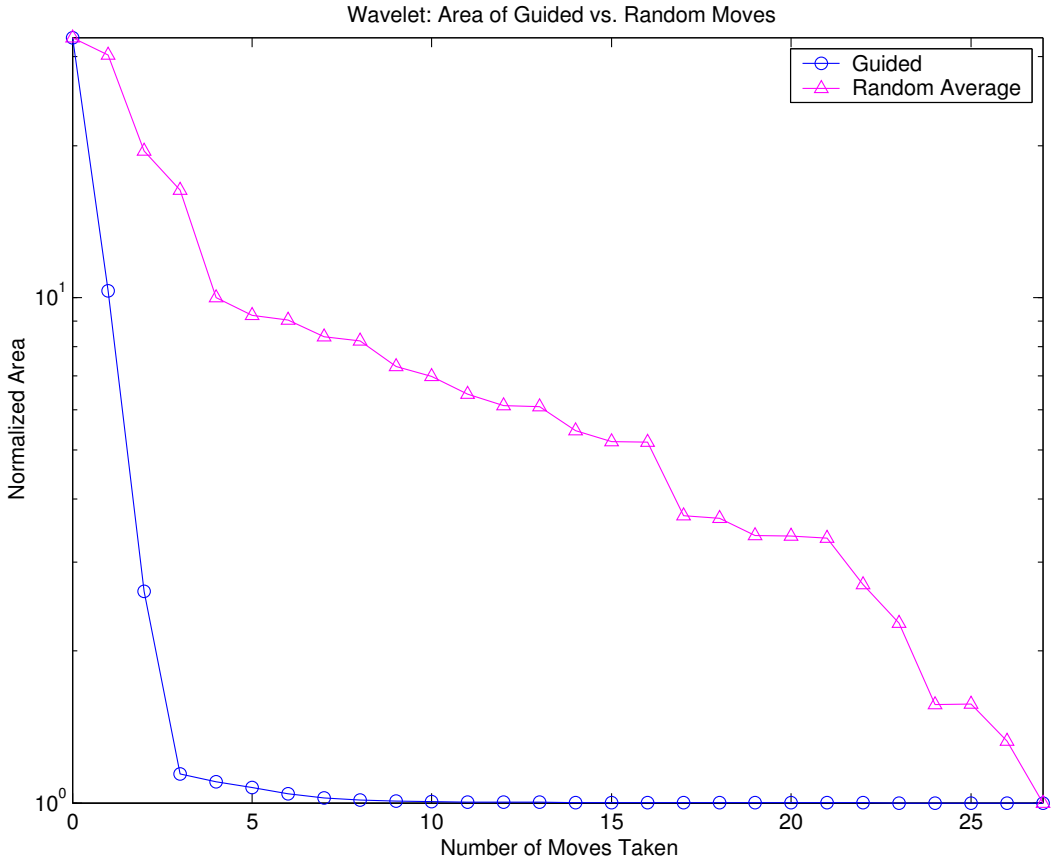

Figure 4.8: Wavelet area vs. number of optimization steps implemented

The wavelet transform was subsequently implemented in MATLAB and optimized in Précis. In total, 27 variables were manually selected to be constrained. These variables were then marked for range-finding analysis and annotated MATLAB code was generated. This code was then run in the MATLAB interpreter with a sample image file (Lena) to obtain range values for the selected variables. These values were then loaded into Précis to obtain a lower bound to be used during the slack analysis phase. The results of the slack analysis are shown in Fig. 4.8.

These results are normalized to the lower bound, which was obtained by setting all variables to their lower bound constraints and computing the resulting area. The slack analysis results suggested constraining the input image array, then the low and

high pass filter coefficients, and finally the results of the additions in the multiply-accumulate structure of the filtering operation.

By iteratively performing the optimization "moves" suggested by the Précis slack analysis phase, we were able to reach within fifteen percent of the lower bound system area in three moves. By about seven moves, the normalized area was within three percent of the lower bound, and further improvements were negligible. At this point a typical user may choose to stop optimizing the system.

To determine if this methodology is sound, we compared the suggested optimization steps to the performance if we were optimizing randomly. We performed four optimization runs where the nodes selected for optimization were randomly chosen from the set of nodes with slack. The same values for the upper and lower precision bounds as the guided optimization scheme were used. The average area of these random passes is plotted versus the guided slack-analysis approach in Fig. 4.8. As shown, the guided optimization route suggested by Précis approaches the lower bound much more quickly than the random method. The random method, while still improving with each optimization step, does so much more slowly than the guided slack analysis approach. From this we conclude that our slack analysis approach provides useful feedback in terms of what nodes to optimize in what order to make the largest gains in the fewest number of optimization steps. This same comparison to random moves is done for all following benchmarks.

It is important to note that the area values obtained by Précis are calculated by reducing the range of a number of variables to their range-found lower bounds. This yields what could be considered the "best-case" solution only for the input data sets considered. In reality, using different input data with these range-found lower bounds might introduce errors into the system. Therefore it is important to continue testing the solution with new data sets even after optimization is complete. This testing step is made easier with the automatic generation of annotated simulation code for use in MATLAB.

4.9.2 CORDIC

The next benchmark investigates the CORDIC algorithm [73], an acronym that stands for **CO**ordinate **R**otation **DI**gital **C**omputer. The algorithm is novel in that it is an iterative solver for trigonometric functions that requires only a simple network of shifts and adds, and produces approximately one additional bit of accuracy for each iteration. A more complete discussion of the algorithm, as well as a survey of FPGA implementations, can be found in [4].

The CORDIC algorithm can be utilized in two modes: rotation mode and vectoring mode. For this benchmark we utilized rotation mode, which rotates an input vector by a specified angle while simultaneously computing the sine and cosine of the input angle. As in [4], the difference equations for rotation mode are:

$$\begin{aligned} x_{i+1} &= x_i - y_i * d_i * 2^{-i} \\ y_{i+1} &= y_i + x_i * d_i * 2^{-i} \\ z_{i+1} &= z_i - d_i * \tan^{-1}(2^{-i}) \end{aligned}$$

where

$$d_i = \begin{cases} -1 & \text{if } z_i < 0 \\ +1 & \text{otherwise} \end{cases}$$

The MATLAB implementation of CORDIC was unrolled into twelve stages. In order to obtain a variety of variable range information during the range finding phase of the analysis, a test harness was developed that swept the input angle through all integer angles between 0° and 90°. The results were then passed into Précis and all 41 intermediate nodes were chosen for slack analysis. The results are shown in Fig. 4.9, truncated to the first 21 moves suggested by the tool, and are consistent with those in the wavelet benchmark.

The suggested moves do not converge upon the lower bound as quickly as the wavelet benchmark, taking until the eighth move to reach the lower bound area. This

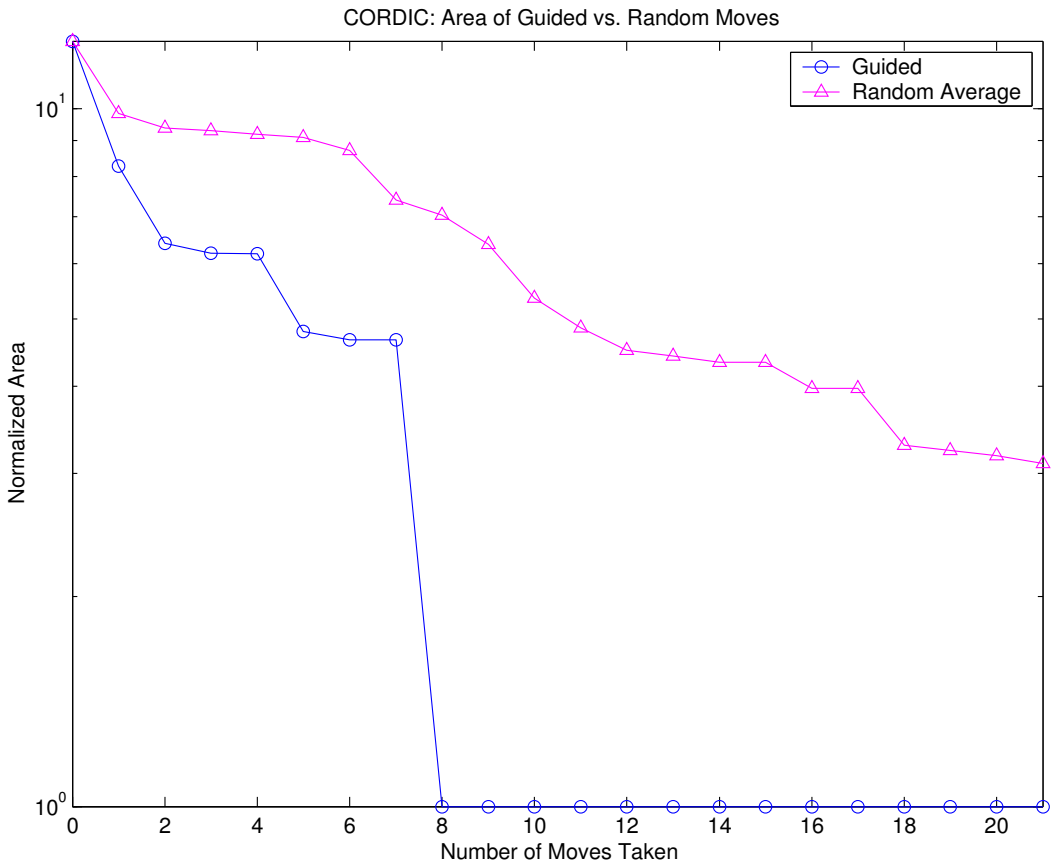

Figure 4.9: CORDIC benchmark results

can be attributed to the fact that the slack analysis algorithm is greedy in nature. The first few proposed moves all originate at the outputs. Only after these are constrained does the slack analysis suggest moving to the input variables. This behavior is in part due to the depth of the adder tree present in the twelve-stage unrolling of the algorithm. The gain achieved by constraining the outputs is greater than the limited impact of constraining any one of the inputs because the output nodes are significantly larger. Shortly thereafter, though, all the input variables are constrained, giving us the large improvement in area after the seventh suggested move, at which point the very linear data path of the CORDIC algorithm has been collapsed to near the lower bound.

4.9.3 Gaussian Blur

The third benchmark is a Gaussian blur implemented as a spatial convolution of a 3x3 Gaussian kernel with a 512x512 greyscale input image. We ignore rescaling of the blurred image for simplicity. The Gaussian blur algorithm was input into Précis and 14 intermediate nodes were chosen for the slack analysis phase. The results are shown in Fig. 4.10. The slack analysis prompted us to constrain first the Gaussian kernel followed by the input image. This led to the largest area improvement—within 28 percent of the lower bound in three moves, and within eight percent in 5 moves. Again, the tool makes good choices for optimization and achieves performance near the lower bound in many fewer optimization steps than the random move approach.

4.9.4 1-D Discrete Cosine Transform

The next benchmark is a one-dimensional discrete cosine transform. The DCT [1] is a frequency transform much like the discrete Fourier transform, but using only real numbers. It is widely used in image and video compression. Our implementation is based upon the work done by [54] as used by the Independent JPEG Group's JPEG

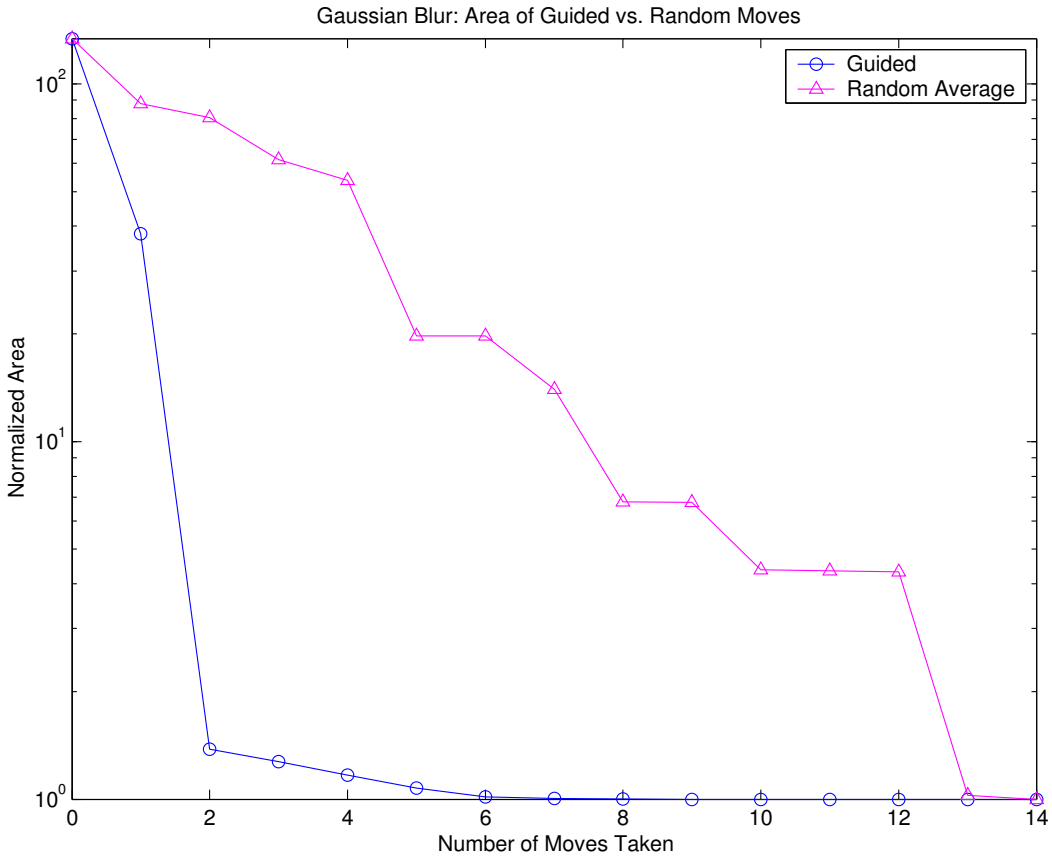

Figure 4.10: Gaussian blur results

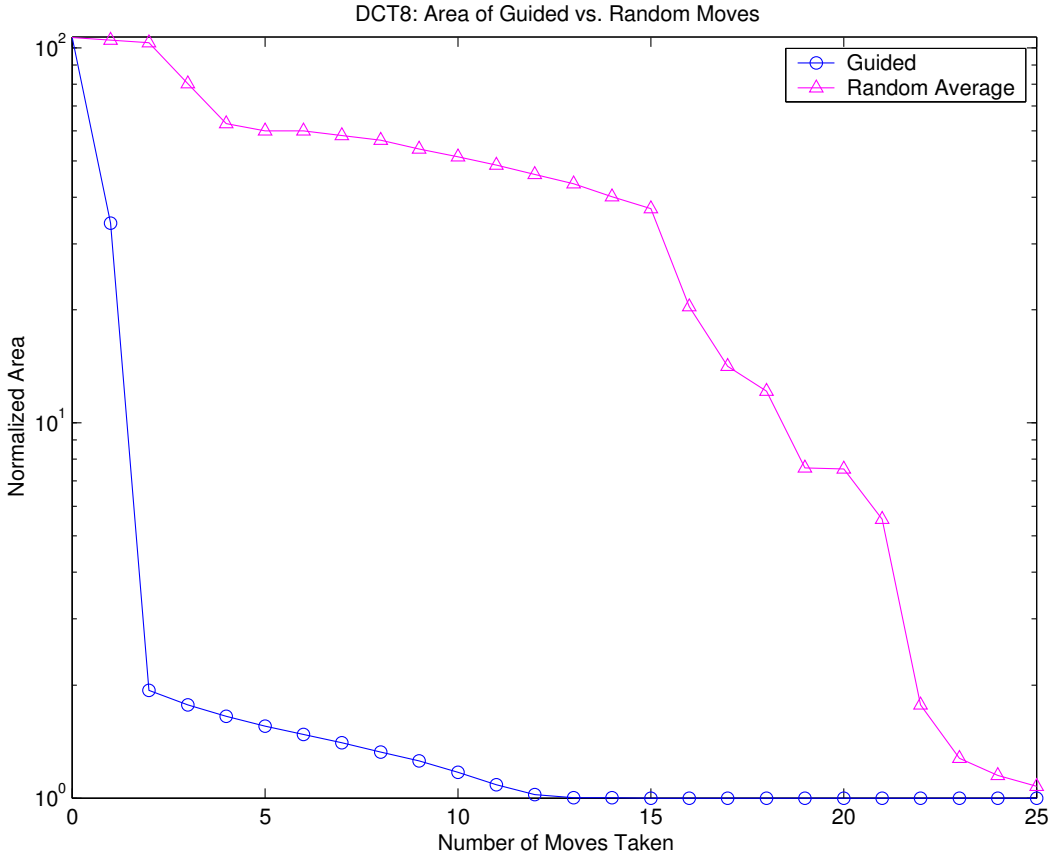

Figure 4.11: 8-point 1-D DCT results

software distribution [51]. This implementation requires only 12 multiplications and 32 additions.

Our MATLAB implementation performed an 8-point 1-D DCT upon a 512x512 input image. The results for all 25 nodes chosen for slack analysis are shown in Fig. 4.11. To get within a factor of two of the lower bound, the input image and DCT input vector are constrained. The suggested moves achieve within 50 percent of the lower bound within six moves, and within two percent in 12 moves.

4.9.5 Probabilistic Neural Network

The final benchmark we investigated was a multi-spectral image-processing algorithm, similar to clustering analysis and image compression, designed for NASA satellite imagery. The goal of the algorithm is to use multiple spectral bands of instrument observation data to classify each image pixel into one of several classes. For this particular application, these classes define terrain types, such as urban, agricultural, rangeland, and barren. In other implementations, these classes could be any significant distinguishing attributes present in the underlying dataset. This class of algorithm transforms the multi-spectral image into a form that is more useful for analysis by humans.

One proposed scheme to perform this automatic classification is the Probabilistic Neural Network classifier [22]. In this implementation, each multi-spectral image pixel vector is compared to a set of training pixels or weights that are known to be representative of a particular class. The probability that the pixel under test belongs to the class under consideration is given the formula depicted below.

$$f(\overrightarrow{X}|S_k) = \frac{1}{(2\pi)^{d/2}\sigma^d} * \frac{1}{P_k} * \sum_{i=1}^{P_k} \exp\left[- \frac{(\overrightarrow{X} - \overrightarrow{W_{ki}})^T(\overrightarrow{X} - \overrightarrow{W_{ki}})}{2\sigma^2} \right]$$

Here, $\overrightarrow{X}$ is the pixel vector under test, $\overrightarrow{W_{ki}}$ is the weight i of class k, d is the number of spectral bands, k is the class under consideration, σ is a data-dependent "smoothing" parameter, and P_k is the number of weights in class k. This formula represents the probability that pixel $\overrightarrow{X}$ belongs in the class S_k. This comparison is then made for all classes and the class with the highest probability indicates the closest match.

This algorithm was manually implemented on an FPGA board and described in greater detail in [15]. Like the wavelet transform described earlier, significant time and effort was spent on variable range analysis, with particular attention being paid to the large multipliers and the exponentiation required by the algorithm. This

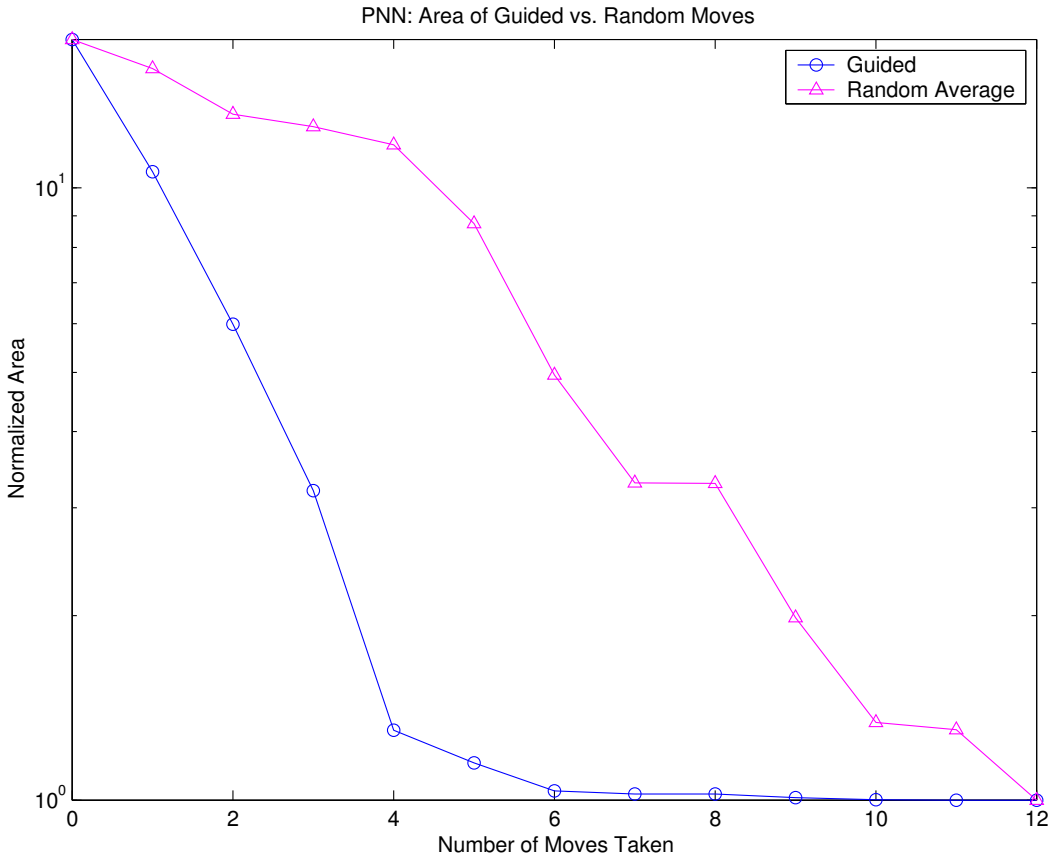

Figure 4.12: PNN area vs. number of optimization steps implemented utilizing only range-analysis-discovered values

implementation obtained speedups of 16 versus a software implementation on an HP workstation.

The algorithm was implemented in MATLAB and optimized with Précis. Twelve variables were selected and slack analysis was run as in the previous benchmarks. Again, all results were normalized to the lower bound area. As shown in Fig. 4.12, the tool behaved consistently with other benchmarks and was able to reach within four percent of the lower bound within six moves, after which additional moves served to make only minor improvements in area.

For a seasoned developer with a deeper insight into the algorithm, or for someone that already has an idea of how the algorithm would map to hardware, the range-analysis phase sometimes returns results that are sub-optimal. For example, the range-analysis of the PNN algorithm upon a typical dataset resulted in several variables being constrained to ranges such as $[2^0, 2^{-25}]$, $[2^8, 2^{-135}]$, $[2^0, 2^{-208}]$, and so on. This simply means that the range-finding phase discovered values that were extremely small and thus recorded the range as requiring many fractional bits (bits right of decimal point) to capture all the precision information. The shortcoming of the automated range-analysis is that it cannot determine at what point values become too small to affect follow-on calculations, and therefore might be considered unimportant. With this in mind, the developer would typically restrict the variables to narrower ranges that preserve the correctness of the results while requiring fewer bits of precision.

Précis provides the functionality to allow the user to make these decisions in its annotated MATLAB code generation. In this case, the user would choose a narrower precision range and a method by which to constrain the variable to that range, consistent with how they implement the operation in hardware—truncation, saturation, rounding, or any of the other methods presented in previous sections. Then, the developer would generate annotated MATLAB code for simulation purposes, and re-run the algorithm in MATLAB with typical data sets. This would allow the user to determine how narrow a precision range would be tolerable and thus constrain the variables in Précis accordingly. The user would then be able to continue the slack analysis phase, optionally reconstraining variables through use of simulation as wider-than-expected precision ranges were encountered.

This user-guided method was performed by reconstraining the variables suggested by the slack analysis phase to more reasonable ranges. For instance, the third variable suggested by the slack analysis phase, **classTotal**, had a range-found precision of $[2^{10}, 2^{-60}]$, far too wide to implement in an area-efficient manner. This value was

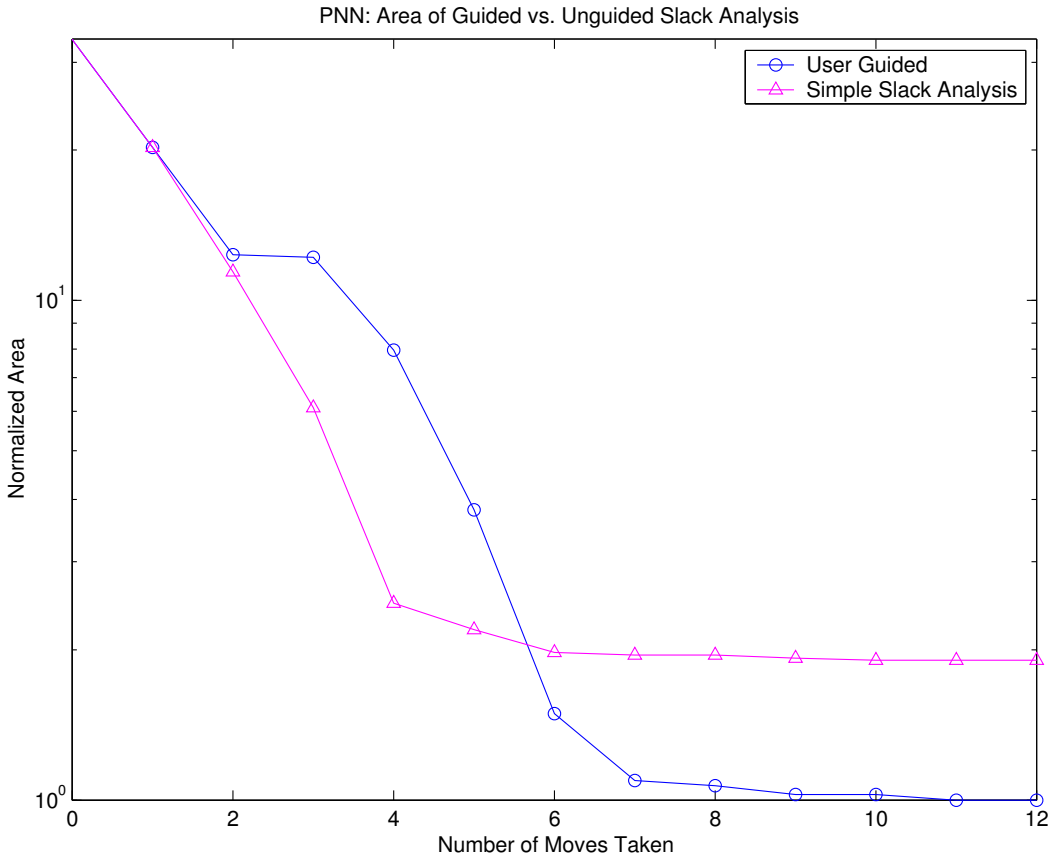

Figure 4.13: PNN area with user-defined variable precision ranges

reconstrained to $[2^{37}, 2^0]$, which includes an implicit scaling factor. This type of reconstraining was performed in the order the variables were suggested by Précis. The results from this experiment are shown in Fig. 4.13, normalized to the lowest bound between the standard and "user-guided" approaches.

At first glance, one can see that both methods provide similar trends, approaching the lower bound within five to seven moves. This behavior is expected and is consistent with the results of the other benchmarks. The results show that the user-guided approach, when reconstraining variables during slack analysis to narrower ranges, achieves a lower bound that is almost 50 percent lower than slack analysis without

user guidance. As expected, the unguided slack analysis approach does not improve further as the number of optimization steps is increased.

The intuition of the hardware developer is used in this case to achieve a more area-efficient implementation than was possible with the unguided slack analysis optimization. The ability to keep the "user in the loop" for optimization is crucial to obtaining good implementations, something that Précis is clearly able to exploit.

4.10 Limitations

The results of the previous sections have shown encouraging results for guided optimization of the position of the most-significant bit using the approaches presented in this chapter. However, there are a number of limitations that present themselves when the entire scope of interactive iterative optimization is taken into account.

In its current implementation, Précis uses a very simplistic area model to perform the area estimation. While appropriate for the complexity of algorithms presented in this dissertation, significantly larger and more complex algorithms may benefit from more accurate models. This limitation is corrected in Chapter 5 where more accurate models are introduced.

Another limitation is the fact that manual optimization is still necessary for final implementation. Since the tool possesses the data path width information, it could perform automatic behavioral HDL generation, providing many potential benefits. Some benefits include: the developer would have a basic HDL starting point for further refinement; if synthesis and place and route are performed, actual area and timing numbers could be factored into the optimization process; simulation could take place on actual hardware at hardware speeds, allowing for more testing to take place. While not a trivial task, automatic HDL may be a useful route to examine in the future.

Finally, as previously mentioned, the slack analysis approach depends upon precision slack between an upper and lower bound found through using the propagation engine and range finding analysis, respectively. The quality of result that the slack analysis technique yields is data dependent, a shortcoming inherent in the hybrid analytical/simulation techniques employed in Précis.

One can partially alleviate this shortcoming through careful data set selection and increased testing. Data sets that exercise the full expected range of precision, both common and extreme cases, should be used to gather meaningful information from the range finding phase of analysis. Then, once an implementation is obtained, simulation with new data sets might be prudent to determine if the data path width is too aggressive or too conservative. Unfortunately, this places the burden on the user and requires more simulation time to verify the correctness of an implementation.

A more fundamental change may also alleviate some of the data dependency. By expanding the amount of statistical information gathered in the range finding phase of analysis it may be possible to predict the likelihood of other data sets falling above or below the monitored ranges. By using information such as mean and variance, this likelihood information can be propagated back to the user as the risk of error being introduced into the output. By incorporating a statistical component to the lower bound, the range finding phase can be made less sensitive to extreme cases in the data set.

While the results shown in this chapter are consistently positive over a range of benchmarks, limitations to the techniques presented do exist and should be addressed. Doing so would improve upon the results shown and enhance the usefulness of Précis as an optimization tool.

4.11 Summary

In this chapter we presented Précis, a tool that enables semi-automatic, user-centric, design-time precision analysis and data path optimization. Précis combines an automatic propagation engine, a fixed-point simulation environment with automatic MATLAB code generation, MATLAB support routines with automatic code generation for variable statistics gathering, and a slack analysis phase. Together, this tool chest addresses a major shortcoming of automated data path optimization techniques: leaving the developer out of the optimization. We have demonstrated an effective methodology for guiding the developer's eventual manual optimization toward those regions of the data path that will provide the largest area improvement.

Précis aids new and seasoned hardware developers in answering the four basic questions needed to perform data path optimization at a very high level, before HDL is generated. At this time, small design changes almost always lead to large differences in performance of the final implementation. Thus, it is crucial to have assistive tools from the very beginning of the design cycle, in particular, data path optimization. Unfortunately, there are few commercial and academic tools that provide this level of support, highlighting the importance of this contribution.

Chapter 5

LEAST-SIGNIFICANT BIT OPTIMIZATION

The previous chapter discussed methodologies and tools that allow us to perform guided most-significant bit position optimization in a convenient and novel fashion. The slack analysis procedure, while effective in dealing with the position of the most-significant bit, does not attempt to optimize the position of the least-significant bit. In this chapter we investigate methods that allow optimization of the least-significant bit position in an automated fashion given user-defined area and error constraints.

As discussed in previous chapters, increasing the relative bit position of the least-significant bit (toward the most-significant end) reduces the maximum precision that the data path may attain. A side-effect of this precision reduction is a likely error accumulation through truncation or round-off error when a data path value requires greater precision than the data path can represent. After performing the optimization for the most-significant bit position as described in the previous chapter, an area/error tradeoff analysis must be performed to optimize the position of the least-significant bit. This tradeoff analysis and its automation are the focus of this chapter.

5.1 Constant Substitution

In order to quantify the benefit of the optimization techniques we present in this chapter, we introduce our metrics: error and area. Error is measured as the net distance from the correct answer, or $|expectedValue - obtainedValue|$, while area is estimated in terms of number of FPGA logic blocks consumed.

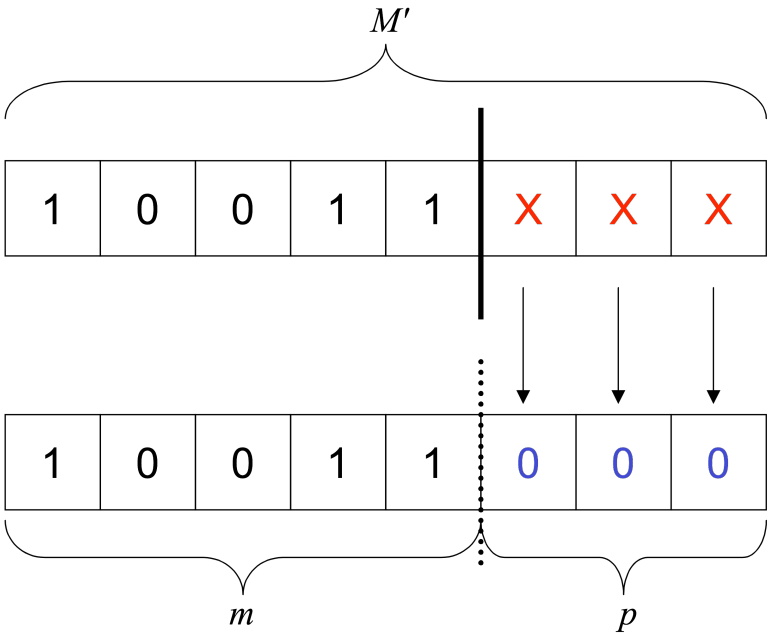

Figure 5.1: Constant substitution

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 constants, in this case zeros (Fig. 5.1). Therefore, for an M'-bit value, we will use the notation $A_m 0_p$. This denotes a word that has m correct bits and p zeros inserted to signify bits that have been effectively truncated, resulting in an $M' = m + p$-bit word.

We begin the analysis of this fundamental change in data path sizing technique by developing models for area and error estimation of a general island-style FPGA.

5.2 Hardware Models

By performing substitution rather than immediate truncation, we introduce a critical difference in the way hardware will handle this data path. Unlike the case of immediate truncation, the implementation of downstream operators does not need to change to handle different bit-widths on the inputs. If the circuit is specified in a behavioral fashion using a hardware description language (HDL), the constant substitution becomes a wire optimization that is likely to fall under the jurisdiction of vendor tools such as the technology mapper and the logic synthesizer.

For example, in an adder, as we reduce the number of bits on the inputs through truncation, the area requirement of the adder decreases. The same relationship exists when we substitute zeros in place of variable bits on an input, because wires can be used to represent static zeros or static ones, so the hardware cost in terms of area is essentially zero. The area impact similarity to truncation should be obvious, as both methods remove bits from the data path computation. This allows us to use constant substitution in place of truncation for many data path operators, as their semantics are identical. In the next sections we outline the area models used to perform area estimation of the data path.

5.2.1 Adder Hardware Model

One of the most simple FPGA logic block architectures is the two-input lookup table (2-LUT). Its simplicity allows for ease of abstraction in terms of an area model. In a 2-LUT architecture, a half-adder can be implemented with a pair of 2-LUTs. Combining two half-adders together and 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. 5.2.

Starting at the least-significant side, all bit positions that overlap with zeros need only wires since the inputs can be wired directly to the outputs. The next most

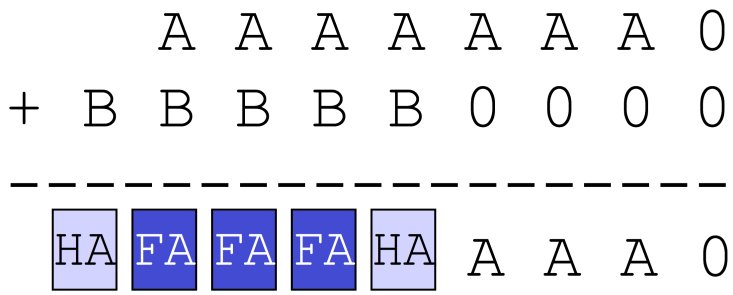

Figure 5.2: Adder hardware requirements

significant bit will only require a half-adder, as there can be no carry-in from any lower bit positions. For the rest of the overlapping bit positions, we require a regular full-adder structure, complete with carry propagation. 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 hardware impact of substitution can be generalized using the formulae in Table 5.1 and the notation previously outlined. This allows an analytic estimation of the area requirement of constant-substituted adder structures to be performed. For the example in Fig. 5.2, we have the following notation 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.

5.2.2 Multiplier Hardware Model

The same approach is used to characterize the multiplier. A multiplier consists of a multiplicand (top value) multiplied by a multiplier (bottom value). The hardware

Table 5.1: Adder Area

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

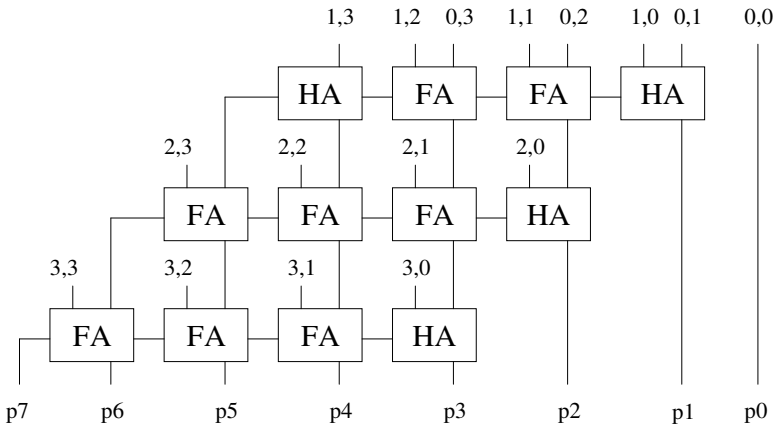

Figure 5.3: Multiplication structure

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. 5.3.

Referring to the example in Fig. 5.4, each bit of the input that has been substituted with a constant 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 a column from the partial product array. For each bit of the multiplier that is zero, we

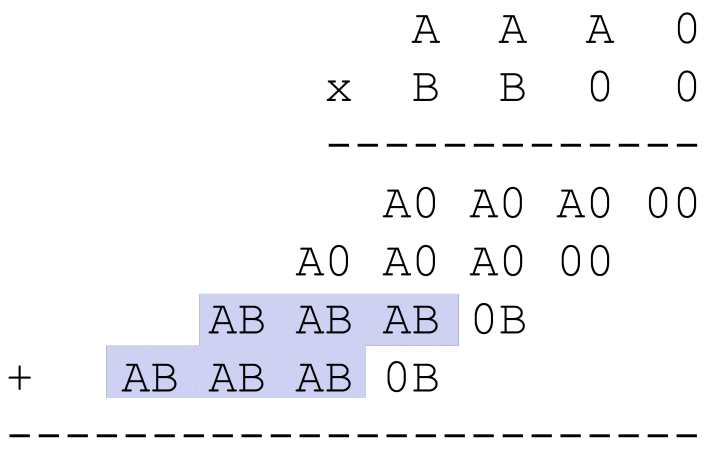

Figure 5.4: Multiplication example

remove a 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, shown as the shaded portion of Fig.5.4. This requires two half-adders, one full-adder, and six AND gates, for a total of 15 2-LUTs. This behavior has been generalized into formulae shown in Table 5.2.

5.2.3 Model Verification

To verify the developed hardware models against real-world implementations, we implemented both the adder and multiplier structures in Verilog on the Xilinx Virtex FPGA using the vendor-supplied place and route tools, *Xilinx Foundation*.

For the adder structure, we observe in Fig. 5.5 that the model closely follows the actual implementation area, being at worst within two percent of the actual Xilinx Virtex hardware implementation. The number of bits substituted at the two adder inputs was the same within each data point.

Table 5.2: Multiplier Area

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

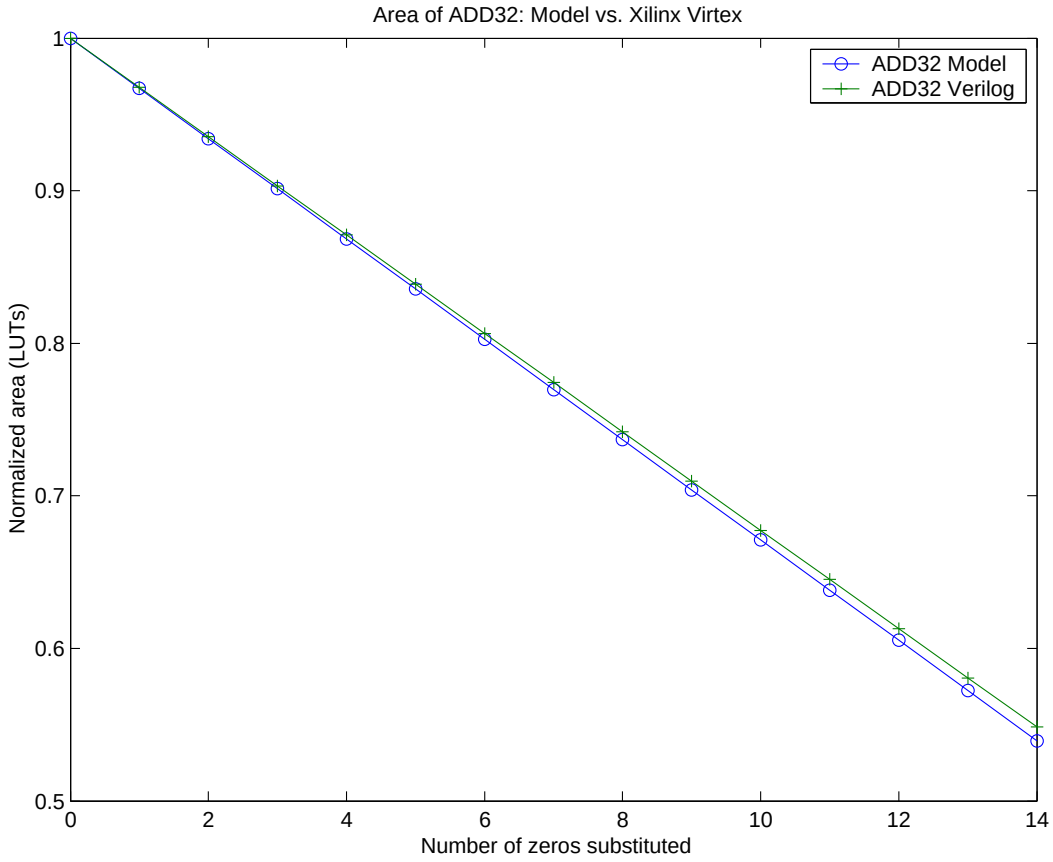

Figure 5.5: Adder model verification

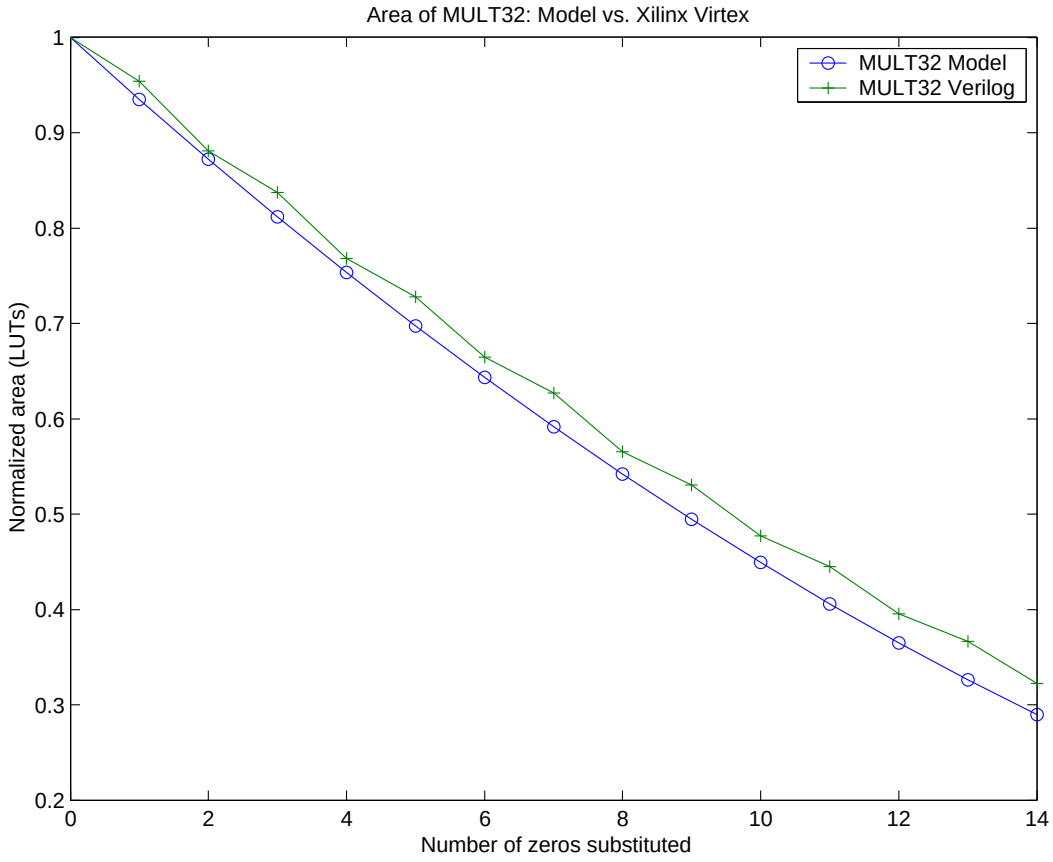

Figure 5.6: Multiplier model verification

The multiplier in Fig. 5.6 has a similar result to the adder, being at worst within 12 percent of the Xilinx Virtex implementation. These results support the use of the simple 2-LUT approximation of general island-style FPGAs to within a reasonable degree of accuracy.

5.3 Error Models

Area is only one metric upon which we will base optimization decisions. Another crucial piece of information is the error introduced into the computation through the quantization error of a fixed-point data path.

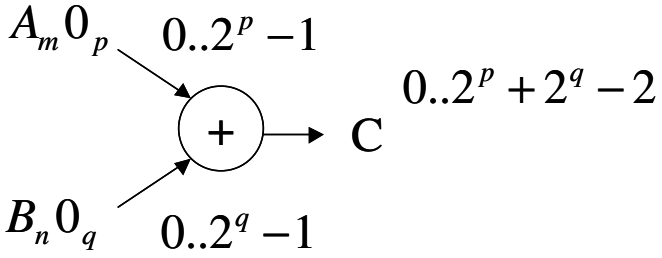

Figure 5.7: Error model of an adder

Having performed a reduction in the precision that can be obtained by this data path with a substitution of zeros, we have introduced a quantifiable amount of error into the data path. For an A_m0_p value, substituting p zeros for the lower portion of the word gives us a maximum error of $2^p - 1$. This maximum error occurs when the bits replaced were originally *ones*, making this result too low by the amount $2^p - 1$. If the bits replaced were originally *zeros*, we will have incurred no error. We will use the notation $[0..2^p - 1]$ to describe this resultant error range produced by the substitution method.

As with the area models, using constant zeros at the least-significant end of a word is functionally equivalent, in terms of error, to truncating the same number of bits for many types of data path operators. This is intuitive in operations such as addition and multiplication where truncation removes bits from the computation just as using constant zeros does.

This abstract error model is used in the optimization methodology to estimate the effective error of combining quantized values with arithmetic operations. We discuss the details of implementing this error model for adder and multiplier structures.

5.3.1 Adder Error Model

The error model for an adder is shown in Fig. 5.7. The addition of two possibly quantized values $A_m0_p + B_n0_q$, results in an output, C, which has a total of

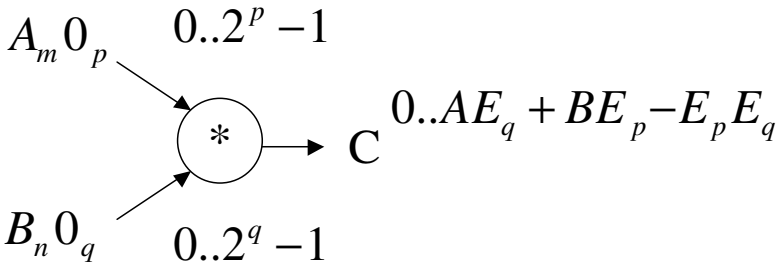

Figure 5.8: Error model of a multiplier

$\max(M', N') + 1$ bits. Of these bits, $\min(p, q)$ of them are substituted zeros at the least-significant end. In an adder structure, the range of error for the output, C, is the sum of the error ranges of the two inputs, A and B. This results in an output error range of $[0..2^p + 2^q - 2]$.

5.3.2 Multiplier Error Model

We use the same approach to derive an error model 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. Here, $p + q$ of them are substituted zeros at the least-significant end. This structure is shown in Fig. 5.8.

The output error is 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, but unlike the adder, multiplying these two inputs together requires us to multiply the error terms as well, as shown in (5.1).

$$\begin{aligned} C &= A * B \\ &= (A - (2^p - 1)) * (B - (2^q - 1)) \\ &= AB - B(2^p - 1) - A(2^q - 1) + (2^p - 1)(2^q - 1) \end{aligned} \tag{5.1}$$

The first line of (5.1) 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 (5.1) yields the simpler (5.2):

$$\begin{aligned} C &= AB - BE_p - AE_q + E_pE_q \\ &= AB - (AE_q + BE_p - E_pE_q) \end{aligned} \tag{5.2}$$

From (5.2) 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.

5.4 Optimization Methods

Using the models described in the previous sections, we can now quantify the tradeoffs between area and error of various optimization methodologies, some of which exploit the use of constant substitution to derive benefits over simple truncation.

5.4.1 The Nature of Error

Looking at the error introduced into a data path using the standard method of simple truncation, we see that the error is skewed, or biased, only in the positive direction. As we continue through data path elements, if we maintain the same truncation policy to reduce the area requirement of our circuits, the lower-bound error will remain zero while the upper bound will continue to skew toward larger and larger positive values. This behavior also holds true for the zero-substitution policy in Fig. 5.7 and Fig. 5.8.

This error profile does not coincide with our natural understanding of error. In most cases we consider the error of a result to be the *net distance from the correct value*, implying that the error term can be either positive or negative. Unfortunately,

neither straight truncation nor the zero-substitution policy, as defined in previous sections, matches this notion of error. These methods only utilize *half* of the available area, as they do not exploit the negative region of error. Fortunately, substituting constants for the least-significant bits allows us to manipulate their static values and capture this more intuitive behavior of error. We call this process *renormalization.*

5.4.2 Renormalization

It is possible for us to capture the more natural description of error with constant-substitution because the least-significant bits are still present. We can use these bits to manipulate the resultant error range. An example of renormalization in an adder structure is shown in Fig. 5.9. 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, an error range whose net distance from zero is achieved that is half that if zero substitution was used. If instead truncation were performed, no further shaping of the error range would be possible, leaving us with a positively skewed error range not consistent with our natural notion of error.

For example, in Fig. 5.7, a substitution of p, q zeros results in an error range of $[0..2^p + 2^q - 2]$. By using 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, $[-(2^p - 1)..2^q - 1]$, if $p == q$, as shown in Fig. 5.9.

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 be biased. Fig. 5.10 depicts a normalization centered near zero by substituting ones

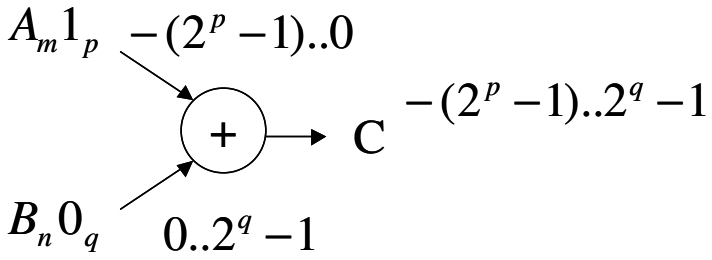

Figure 5.9: Normalized error model of an adder

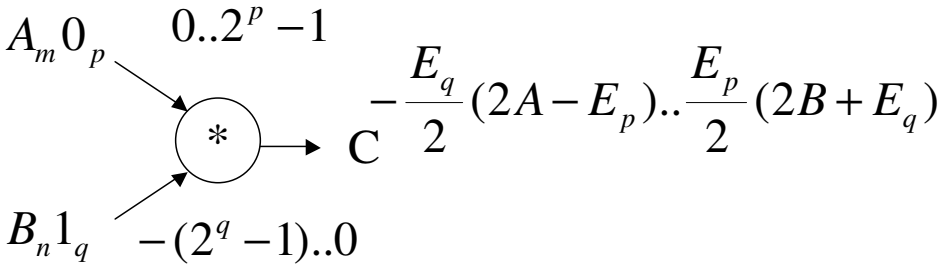

Figure 5.10: Normalized error model of a multiplier

instead of zeros for input B, and if $p == q$. The derivation of the resultant error range is as follows in (5.3):

$$\begin{aligned} C &= (A - E_p)(B + Eq) \\ &= AB + AE_q - BE_p - E_pE_q \\ &= AB + AE_q - (BE_p + E_pE_q) \\ &= AB + AE_q - \frac{E_pE_q}{2} - \left(BE_p + \frac{E_pE_q}{2}\right) \\ &= AB + \frac{E_q}{2}(2A - E_p) - \frac{E_p}{2}(2B + E_q) \end{aligned} \tag{5.3}$$

Performing in-line renormalization requires modifying substitution constants on primary inputs. A variation of in-line renormalization that can accomplish the same error biasing without requiring the manipulation of inputs is "active renormalization". By inserting a constant addition, we can again deterministically bias the output error

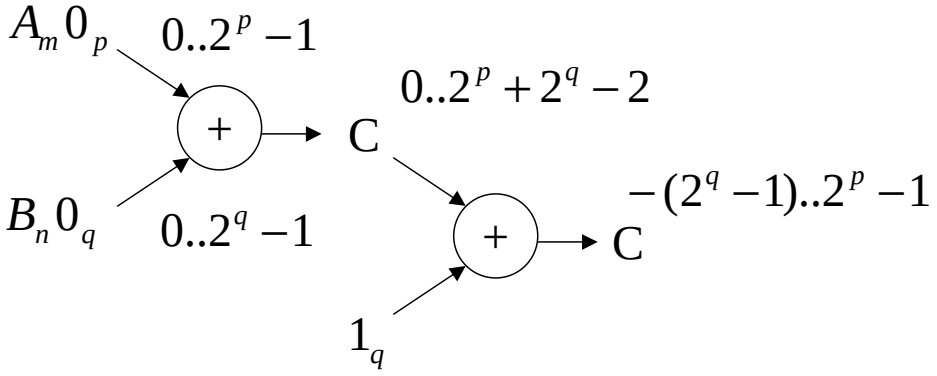

Figure 5.11: Using addition to perform active renormalization

range. An example is shown in Fig. 5.11. Active renormalization is not so much a different technique from in-line renormalization, but rather should be considered a different implementation of the same optimization technique. Active renormalization is useful when intermediate nodes may have very imbalanced error ranges that cannot be corrected by manipulating the inputs. Active renormalization can also be implemented with little area overhead within existing arithmetic structures, as will be discussed in the next section, where the distinction between in-line and active renormalization becomes less rigid.

5.4.3 Renormalization Area Impact

The benefits of renormalization can come very cheaply in terms of area. The adder structure example from Fig. 5.2 originally requires 19 2-LUTs and has an error range of [0..16]. From here, we can perform a variety of optimizations that achieve different, and perhaps better, area-to-error profiles.

To perform in-line renormalization, we simply substitute a "1" for the least-significant bit on one of the inputs. This would yield an output error range of [−1..15]. While not particularly biased, it doesn't incur any area penalty as the newly substi-

tuted "1" lines up with a zero from the other input, requiring no computational hardware.

Even when substituted ones and zeros on the inputs completely overlap, consideration must be made for downstream operations, as we now have ones in the least-significant bit positions which may need to be operated upon in subsequent operations. This may adversely impact the overall area of the circuit, at which point "active" renormalization should be considered as an alternative that can be implemented cheaply later in the data path to "fix up" the error range using a constant bias.

We can achieve a balanced output error range of $[-8..8]$ by flipping the constant zero substituted at the 2^3 bit position of input B to a constant one, as shown in 5.12. With the presence of a constant one, a different calculation must be performed for that bit position by replacing the original wire with an inverter structure. Using FPGAs, which are lookup table based computational devices, this inverter structure is simply a change in how the lookup table(s) this signal eventually reaches are computed, and therefore requires no additional hardware. In order to handle the possible carry-out from this pseudo-half-adder at the 2^3 bit position, the half-adder originally at the 2^4 bit position must be replaced with a full-adder. These two changes have an area impact of +3 2-LUTs and result in a balanced error range, $[-8..8]$, half that of the original error range of $[-16..16]$.

Alternatively, we can obtain a completely negative bias of $[-16..0]$ with zero area penalty by modifying the structure of the half-adder at the 2^4 bit position to have a constant carry-in of 1 as shown in Fig. 5.13. This is a simple change to the lookup table implementation of the half-adder that effectively adds a constant value of 16 to the addition without incurring an area penalty. This has the same effect as using the "active renormalization", where an explicit addition is performed to change the error bias of the data path.

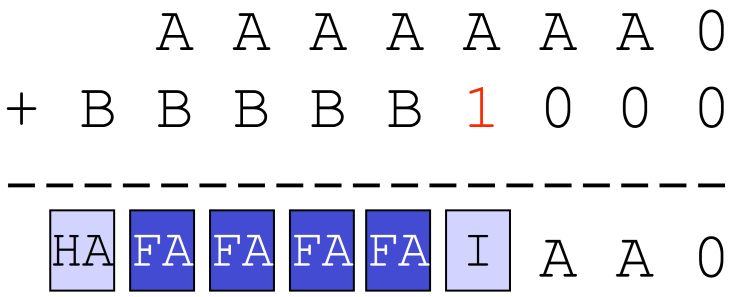

Figure 5.12: Renormalization by changing input constants achieves balanced output error with +3 area penalty

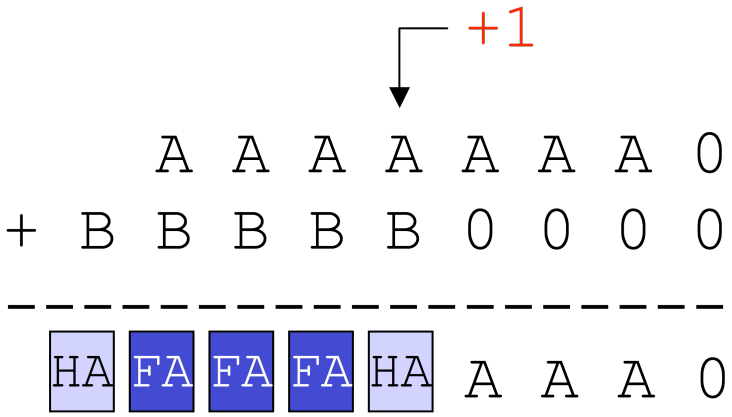

Figure 5.13: Renormalization by modification of adder structure achieves completely negative output error range with zero area penalty

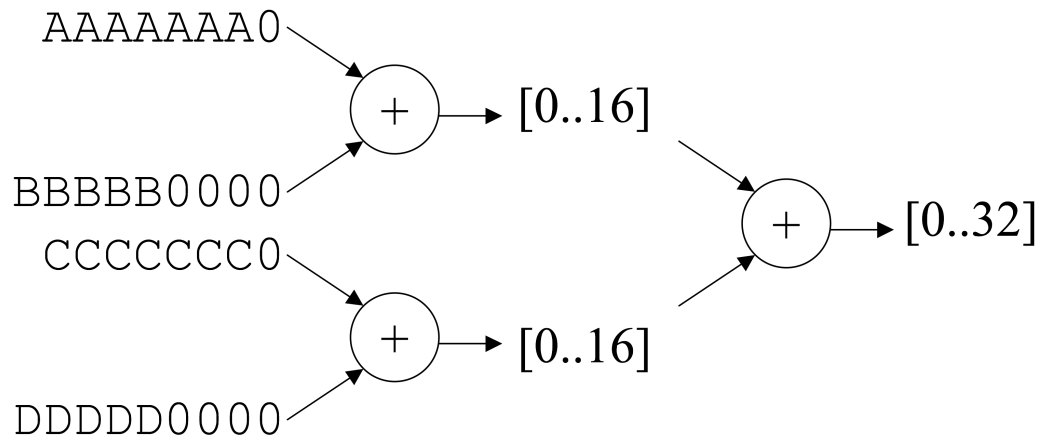

Figure 5.14: Simple three adder data path before renormalization has maximum error range of 32

While at first glance the resultant error range of $[-16..0]$ may not be any narrower than the original error range, having a variety of error range biases can prove very useful when considering a larger number of data path operators. Consider the simple data path comprised of three adders in Fig. 5.14.

The two leftmost adders are the same as the adder example of Fig. 5.2. Each of the first level adders produces an output error range of [0..16]. The output of the sum of these two first level adders yields an output error range at the second level adder of [0..32]. If the modification depicted in Fig. 5.13 is applied to the $C + D$ first level adder, the output error range will be completely negatively biased to $[-16..0]$. This optimized implementation is shown in Fig. 5.15. The output error range of the second level adder will now be $[-16..16]$ instead of [0..32]. Renormalization has given a result that has half the overall maximum error range with zero area penalty.

The behavior of renormalization in multiplier structures is equally interesting. As can be seen in Fig. 5.4, zeros substituted at the least-significant end of either the multiplier or the multiplicand "fall" all the way through to the result. For the multiplication $A_m 0_p * B_n 1_q$, p zeros will be present at the least-significant end of the result,

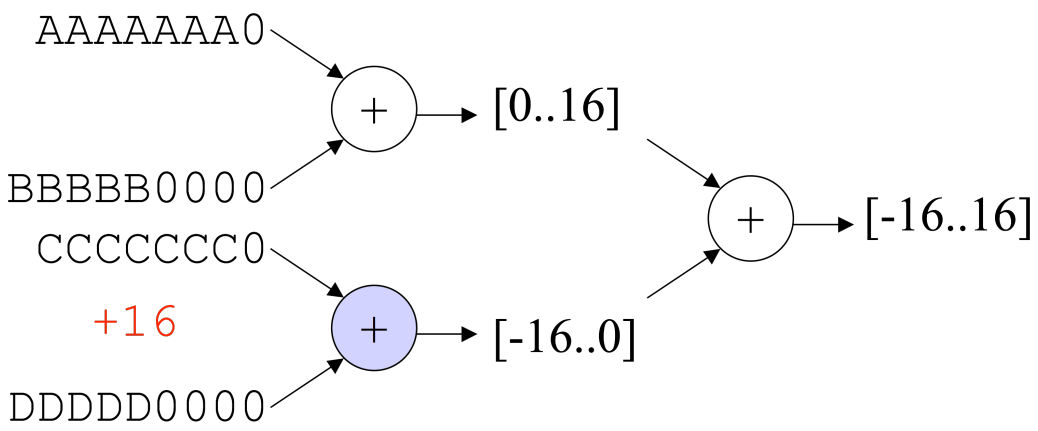

Figure 5.15: Simple three adder data path after renormalization has maximum error range of 16

having "fallen through" from the p zeros substituted on input A. This is advantageous compared to the adder because a renormalized error result can be obtained while still providing zero-substituted bit positions that will not have to be operated upon in downstream operations. This is important in providing opportunities for area savings throughout the data path. As with the adder structure, we pay a penalty for this renormalization. For the multiplier, we must put back an inner row and column for each one-substitution present in the multiplier and multiplicand, respectively.

5.4.4 Alternative Arithmetic Structures

As discussed in previous sections, the 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 [53]. This work, and the work of others [61, 76], focuses on removing a number of least-significant columns of the partial-product array in order to provide a different area-to-error profile.

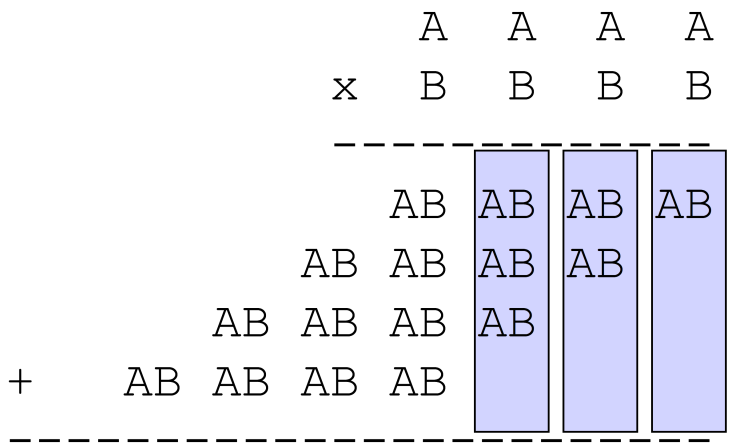

Figure 5.16: Truncated multiplier: removal of shaded columns reduces area and increases error

As described in [61], by removing the n least-significant columns from an array-multiplier multiplication, we save (for $n \geq 2$) $\frac{n(n+1)}{n}$ AND gates, $\frac{(n-1)(n-2)}{2}$ full adders, and $(n-1)$ half adders. This column removal, depicted in Fig. 5.16, reduces the area requirement of a truncated multiplier. This method has a different area-to-error tradeoff profile than a traditional array multiplier, and is illustrated in Fig. 5.17 for a 32-bit multiplier. In this figure, each marker represents one column truncated from the partial product array, or one pair of input bits substituted with constant zeros, using truncated or zero-substituted multipliers, respectively. The right-most markers represent one column and one bit of zero substitution, increasing toward the left.

In the work presented by Lim, Schulte and Wires [53, 61, 76], it was assumed that the inputs to the truncated multipliers were full precision, in other words, without constant substitutions made. If this limitation were to be carried into the framework presented in this chapter, it would require all calculations prior to the use of a truncated multiplier to be full precision (i.e. no constant substitutions), possibly negating any area gain by requiring larger, full-precision operators upstream. This would make

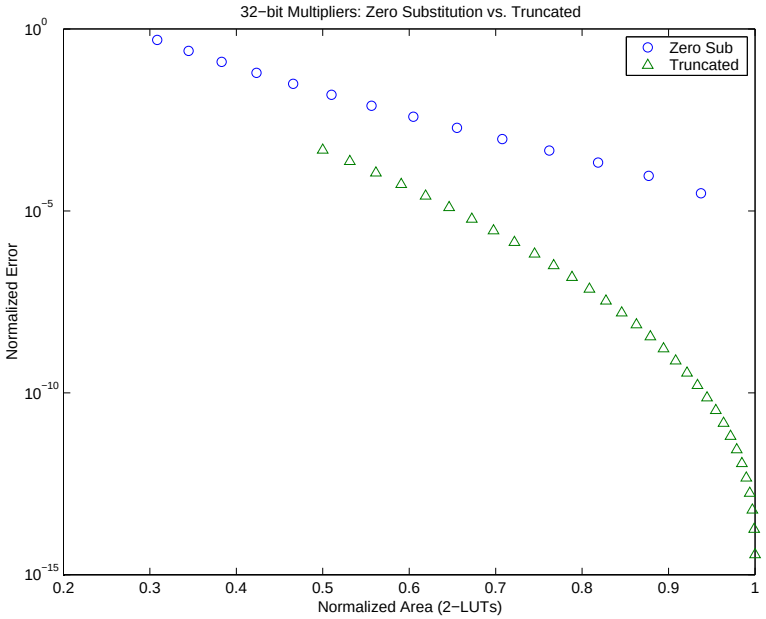

Figure 5.17: Error to area profile of zero-substitution 32-bit multiplier and truncated 32-bit multiplier

truncated multipliers more valuable in multiplications occurring closer to the inputs than those closer to the outputs.

However, it is possible to combine constant substitution with truncation, as each method has a well-defined impact upon area and error. In this work, we allow for both constant substitution as well as truncation in multiplier structures.

5.5 Automated Optimization

We have presented several optimization methods designed to allow more control of the area/error profile of the data path. Due to the strongly interconnected nature of data paths and data flow graphs in general, it is hard to analytically quantify the impact of each method on the overall profile of the system. Making a small change, such as increasing the number of zero-substituted bits at a particular primary input, will impact the breadth of possible optimizations available at every node.

Fortunately, we have provided a model that can accurately determine the area and error of each node within the data path. With these measurements and optimization "moves", we can utilize simulated annealing [47] to choose how to use our palette of optimizations to achieve an efficient implementation area while meeting a user-specified error constraint.

5.5.1 Simulated Annealing

Simulated annealing is a technique commonly used to solve complex combinatorial optimization problems. It is a heuristic technique that leverages randomness to find a good global solution while avoiding local minima. The name and inspiration for simulated annealing originates in the process of annealing in metallurgy. An ancient process, annealing begins by heating a metal to mobilize the atoms within the crystalline lattice. Through uniform and controlled cooling, these mobile atoms find lattice configurations that have lower energy states and are in general more "orderly". The result is a metal that has fewer defects and larger crystals.

A typical simulated annealing algorithm requires four ingredients as described in [47]. A description of the configuration of the system to be optimized, a random generator of "moves" for elements in the system, an objective function to evaluate the quality of any configuration, and a cooling schedule to guide the annealing process.

The annealer begins with a candidate, often random, configuration of the problem. At an initially high temperature, a number of random moves are attempted. Moves that yield an improvement in the configuration are accepted while moves that degrade the configuration are probabilistically accepted based upon the current temperature and the severity of the degradation. At high temperatures bad moves are more likely to be accepted than at lower temperatures, giving the algorithm an opportunity to navigate away from local minima. The duration of the annealing process as well as the acceptance ratio profile is governed by a carefully crafted cooling schedule.

Simulated annealing is a popular solution technique for problems that are otherwise intractable through more "traditional" optimization methods. The difficulty in effective implementation is tuning it for a particular problem. Inventing appropriate moves, a good objective function, and an appropriate cooling schedule requires as much trial-and-error as analytical sense, making a quality implementation somewhat of an art. Even so, studies have shown that good solutions can be obtained with a computational effort that grows very slowly relative to problem size. Simulated annealing is very widely used in hardware design algorithms, in particular placement of cells in a physical layout, therefore we use it here in order to automatically optimize the tradeoffs between area and error in a fixed-point data path.

5.5.2 Simulated Annealing for Automatic LSB Optimization

We have developed an automated optimization approach using simulated annealing principles similar to those found in [9] to area-optimize a data flow graph.

The possible moves in the system are the various optimization methods. At each temperature we choose randomly between altering the amount of zero-substitution at the inputs and changing multiplier structures. The cost function for determining the quality of moves is determined by the area estimate of the entire data path combined with a user-specified error constraint. This error constraint is identified as an error range at a particular node, dubbed the error node. The cost function is defined in (5.4), where *error* is the absolute value of the difference between the maximum error and the target error at the error node. We have determined through experimentation that $\beta = 0.25$ gives a good balance between an area efficient implementation and meeting the error constraint.

$$cost = \beta * area + (1 - \beta) * error \tag{5.4}$$

When modifying an input, we allow the annealer to randomly choose to increase or decrease the number of bits substituted with constants by one bit. Thus, an input $A_5 0_2$ can move to $A_6 0_1$ or $A_4 0_3$.

When modifying the structure of a multiplier, we randomly choose a multiplier and adjust its degree of truncation. As with the inputs, we allow the annealer to increase or decrease by one the number of columns truncated from the partial product array. This allows a smooth transition from the traditional array multiplier to a highly-truncated multiplier.

At each temperature, after the move has been completed, we perform a greedy renormalization. Recalling from previous sections, there are several instances where the effect of renormalization can be achieved without an area impact. For each adder that may be renormalized without area penalty, we perform renormalization and observe the impact on the error node of interest. The adder that exhibits the most reduction in maximum error at the error node through renormalization is renormalized. This process is repeated until either the list of candidate adders is exhausted, or there can be no error improvement through renormalization. After the annealer has finished, we optionally apply active renormalization at the error node if it yields a lower overall implementation cost.

5.6 Experimental Results

We have implemented automated optimization of the LSB position as a subset of our design-time tool presented in [17]. To test the effectiveness of our methodologies, we have used the techniques described here to optimize several benchmark image processing kernels. These include a matrix multiply, wavelet transform, CORDIC, and a one-dimensional discrete cosine transform.

A typical use of our methods would begin with the user performing basic truncation. As mentioned before, while basic truncation does afford an area savings

throughout the data path, there is very little guidance as to which inputs to manipulate, and how changes might affect the overall performance of the implementation. The starting points we have used in these experiments are truncating zero, one, and two bits from every input. These can be seen in Figs. (5.18-5.21) as the "Basic Truncation" points on the plots.

From these initial estimates of area and error, we performed the automated optimization using these points as guidelines for error constraints. The flexibility of the constant-substitution method allows us to choose any error constraint, giving us far more area/error profiles to consider for implementation. As can be seen in the plots, the automated optimization method is able to obtain better area/error tradeoffs than the basic truncation method. The overall results show an average of 44.51% better error for the same amount of area, and an average 14.66% improvement in area for the same level of error versus the simple truncation technique. The overall trend is an improvement over truncation except in a few spurious cases. We attribute this to the need for further tuning of some of the parameters in our simulated annealing algorithm to obtain more robust results. In particular, tuning the β parameter to adjust the weighting of meeting the error constraint vs. obtaining an area-efficient data path. In the future, perhaps this parameter could be influenced by the user.

5.7 Limitations

While the results presented are consistently good, there are a number limitations that warrant discussion. The current implementation of error calculation is limited in that the error calculated at later nodes has only a range of error from previous nodes available for consideration. More specifically, the error contributed by previous nodes is propagated only *by value* rather than *by reference*. The result is that correlations and relationships between error contributing nodes are not preserved. For example, in the simple statement $y = x - x$, the current implementation would not be able to

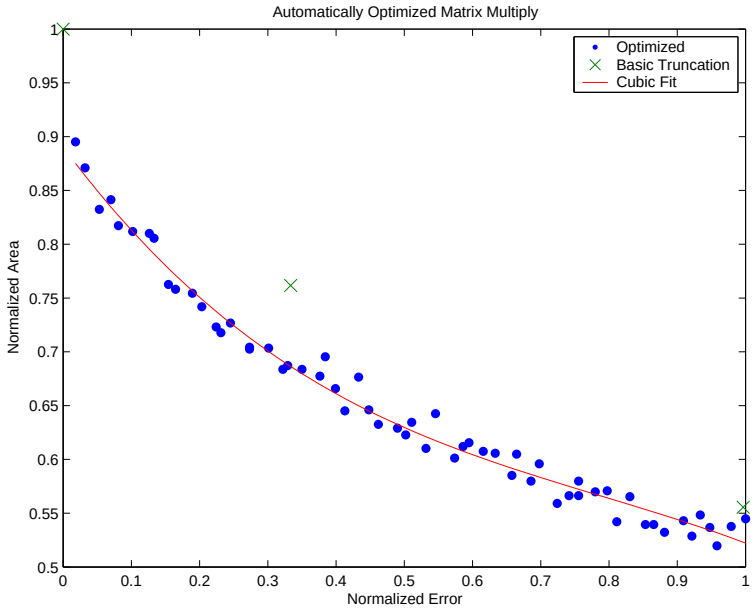

Figure 5.18: Optimized results for matrix multiply

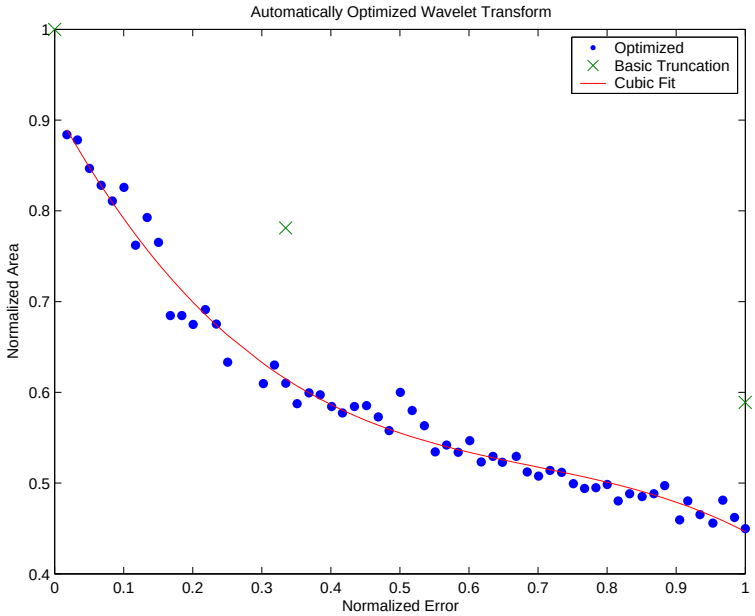

Figure 5.19: Optimized results for wavelet transform

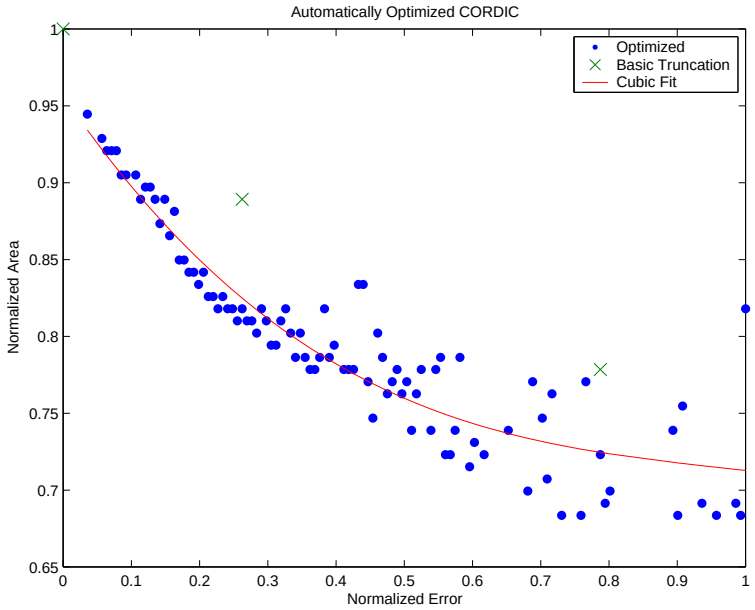

Figure 5.20: Optimized results for CORDIC

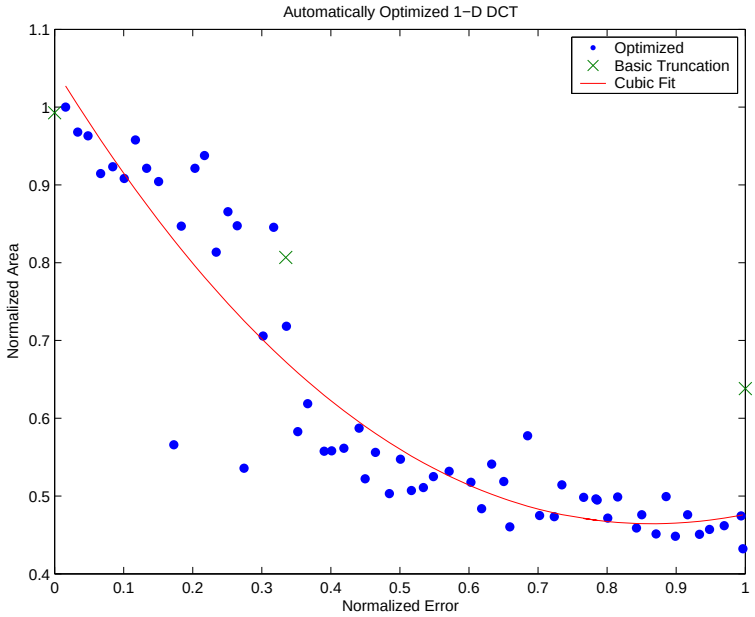

Figure 5.21: Optimized results for 1-D discrete cosine transform

realize that the result y should be zero, regardless of the level of error, or uncertainty, of the input operand x. This is a shortcoming of the interval analysis technique employed to propagate error through the system. This same shortcoming has plagued other work, e.g. M. Stephenson's BitWise compiler [65, 66]. Recently, Fang, et. al. [32–34] have demonstrated a technique called affine arithmetic to overcome some of the limitations of the interval analysis technique. By preserving the correlations between variables it is possible expose these macro-scale relationships and arrive at a better error estimate.

Another limitation is in the small number of optimization options presented. While the results shown remain positive, perhaps more alternative arithmetic structures could be proposed that would yield more varied area/error profiles. Further, no sense of timing or performance is included in the optimization. While the presented adder and multiplier structures generally have delay proportional to their area requirements, this may not be true for other arithmetic structures. Bit- and digit-serial arithmetic structures, for instance, can yield higher precision at the cost of performance. These types of structures cannot be adequately evaluated without considering the timing of the circuit.

The formulation of the error as a value constraint may be limited in usefulness for real-world algorithms. Possible alternatives could include signal-to-noise ratio, bit-error rate, or any number of statistical measures. The impact of a different error constraint on the performance of the automated optimization is an open question.

The ability of the user to specify an area constraint instead of, or in addition to, an error constraint is another possible enhancement to the techniques presented in this chapter. By having both constraints, the automated optimization may be able to prioritize one constraint over another depending on the ease of reaching either constraint. If the area constraint is used instead of an error constraint, the automated technique should then yield an error minimizing implementation. Both of these options would widen the usability of the tools presented.

Finally, another consideration is in use of simulated annealing to automate the optimization process. Other algorithms may be more suited to automating the optimization, but have not been investigated. One possible candidate is the use of a genetic algorithm approach. The objective function is similar to the simulated annealing cost function; the genetic representation, genome, can be the implementation details, including input bit substitutions, operator selection, and renormalization factors; and genetic mutators can be defined similarly to the moves in the simulated annealing approach. This method has not been investigated, but provides an interesting alternative to the simulated annealing approach.

5.8 Summary

In this chapter we have motivated the need to investigate the optimization of the least-significant bit position within a data path. While most-significant bit optimization in the previous chapter only used area to judge the quality of an implementation, area and error *both* play an important role in picking the position of the least-significant bit. How a developer makes the area-to-error tradeoff over a large number of data path nodes is a difficult question to answer manually, leading to the development of the methodologies and tools described in this chapter to support the automatic optimization of the LSB position.

A novel alternative to truncation—constant-substitution—was presented. Constant-substitution affords the developer more control over the area-to-error tradeoff during data path optimization than simple truncation. Models were presented to facilitate area and error estimation of data paths for automated optimization using a number of optimization techniques. These include primary input constant substitution, area-efficient renormalization, and alternative arithmetic structures. Simulated annealing was used to automate the optimization of a data path subject to a user-supplied error constraint using these optimization techniques. Experimental results show that the

automated optimization can achieve better area-to-error performance than simple truncation. Improvements average 44.51% better error for the same area requirement, and 14.66% improvement in area for the same level of error. More importantly, the techniques and tools presented in this chapter give the developer a simple and automatic way to achieve a specific error bound while maintaining an area-efficient implementation, leading to a broader range of options to consider while implementing an algorithm in a fixed-point system.

Chapter 6

CONCLUSIONS AND FUTURE WORK

This dissertation has offered a disciplined look at the problem of precision analysis as it applies to data path optimization for reconfigurable hardware systems such as FPGAs. This includes an in-depth review of the current state of research in precision and bit-width analysis, along with investigations into areas which were previously ignored, including designer-centric assistive optimization tools and techniques for least-significant bit position optimization.

Chapter 3 presented an in-depth look at one of the more difficult hurdles in hardware development—data path optimization. The notion of bit-level operations versus the general-purpose processor model of word-level operations, and the shift from the time-multiplexed fixed-resource computational system of the GPP, to the spatial computation paradigm of hardware devices such as FPGAs is often difficult to grasp. Chapter 3 also provided an overview of previously documented research efforts categorized into three basic approaches: analytical, simulation-based, and hybrids. We point out the common components missing from these previous works, iterative assistive tools that attempt to optimize the position of the least-significant bit.

In Chapter 4 we introduced our designer-centric tool, Précis. A prototyping tool for our most-significant bit optimization methodologies, Précis is unique: it enables semi-automatic, user-centric, design-time precision analysis and data path optimization. Combining a propagation engine, range finding, fixed-point simulation support, and a novel slack analysis phase, the tool keeps the developer in the loop during optimization. By providing feedback and hints on where the user's manual optimiza-

tion efforts should focus, Précis demonstrated effective guidance toward area-efficient implementations, shortening the design iteration time.

In Chapter 5 we introduced novel methods to optimize the position of the least significant bit. Through the use of constant-substitution instead of truncation, the developer retains much more tunability of the area-to-error profile of the data path. This method, combined with area-efficient renormalization and alternative arithmetic structures, optimizes the least-significant bit position given a user-specified error constraint using simulated annealing. Experimental results are promising, and show improvement when compared to simple truncation.

In presenting our methods for optimizing both ends of the data path, more than just area and error improvements have been achieved. The goal of this work was to provide an assistive technology that helps developers implement software-prototyped algorithms in hardware devices. Précis, through the slack analysis method, gives the user important feedback on performing the data path implementation, and what order achieves the largest gains in area for the smallest number of optimization iterations. This falls directly in line with the iterative nature of hardware development and is immediately useful. The techniques introduced in Chapter 5 yield not only better results when compared directly to truncation, they give the developer a useful alternative to truncation that can be used to automatically generate implementation details given an error constraint.

While this work and the work of others have focused on casting floating-point algorithms in a fixed-point paradigm, there are several alternative avenues to FPGA arithmetic that have yet to be investigated. As the capacity of FPGAs continues to grow, floating-point operations implemented directly in the FPGA fabric are becoming a distinct reality. Several authors have investigated implementation of floating-point arithmetic units in FPGAs fabrics [8, 31, 52]. Implementation through parameterized generators or commercially-available floating-point arithmetic cores is only part of the solution. How developers should optimize the use of these resource-hungry com-

putational units is still a largely unanswered question. With floating-point operators added to the palette of arithmetic units, the range of possible implementations increases. Just as a tool is necessary to investigate these tradeoffs in the fixed-point case, it would be even more useful with floating-point operators. Extension of the techniques presented in this dissertation to include area and error models of floating-point operators is one clear avenue to answering this question.

The natural counterpoint to developing floating-point tools for fixed-point FPGAs is developing next-generation FPGA architectures that are tuned for floating-point arithmetic. Several FPGA vendors already provide versions of their devices that include fixed-function silicon "hard cores", ranging from multiply-accumulate structures to embedded processors [3, 29, 79–81]. These fixed-function blocks can be interfaced with the general FPGA fabric and have been proven to be effective in reducing general fabric area consumption by alleviating the need to implement area-consumptive functions in the generally less-efficient FPGA fabric. A perfect candidate for fixed-function implementation are floating-point arithmetic operators.

An FPGA architecture with "real" fixed-function floating-point blocks should yield great performance increases in many applications. Naturally, this architecture would be able to increase the accuracy of computations by providing a true floating-point, rather than an error-inducing fixed-point, data path. As a positive side-effect, not requiring the emulation of floating-point within the FPGA fabric will free a lot of area for additional processing. Combined, these benefits could dramatically increase accuracy, performance, and utilization.

Perhaps more importantly, providing a floating-point capable FPGA brings about a critical turning point in the *usability* of FPGAs. By not having to consider the tradeoffs between floating- and fixed-point arithmetic, hardware implementation could be greatly simplified through the use of smart, high-level synthesis tools that exploit the floating-point data path. Of course, the current research into precision and bit

width analysis would still provide useful avenues for optimal use of what will most likely be scarce floating-point resources.

FPGAs have the potential to bring about a revolutionary change in the way computing workloads are partitioned. The line between hardware and software grows less distinct with each advance in FPGA tools and technology. Research in both FPGA architectures and the software that supports them is crucial in bringing about the wider adoption of FPGAs within all computing communities.

BIBLIOGRAPHY

[1] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. *IEEE Transactions on Computer*, C-23:90–93, January 1974.

[2] Alta Group of Cadence Design Systems, Inc. *Fixed-Point Optimizer User's Guide*, August 1994.

[3] Altera Corporation, San Jose, California. *Excalibur Devices, Hardware Reference Manual, Version 3.1*, November 2002.

[4] Ray Andraka. A survey of CORDIC algorithms for FPGAs. In *ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, 1998.

[5] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, 1999.

[6] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden. MATCH: A MATLAB compiler for configurable computing systems. Technical Report CPDC-TR-9908-013, Northwestern University, ECE Dept., 1999.

[7] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB compiler for distributed, heterogeneous, reconfigurable

computing systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 39–48, 2000.

[8] Pavle Belanovic and Miriam Leeser. A library of parameterized floating point modules and their use. In *International Conference on Field Programmable Logic and Application*, June 2002.

[9] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Seventh International Workshop on Field-Programmable Logic and Applications*, pages 213–222, 1997.

[10] William Joseph Blume. Symbolic analysis techniques for effective automatic parallelization. Master's thesis, University of Illinois at Urbana-Champaign, 1995.

[11] Stephen Brown and Jonathan Rose. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

[12] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, May 1992.

[13] W. Carter, K. Duong, R.H. Freeman, H.C. Hsieh, J.Y. Ja, J.E. Mahoney, L.T. Ngo, and S.L. Sze. A user programmable reconfigurable gate array. In *IEEE 1986 Custom Integrated Circuits Conference*, 1986.

[14] Celoxia. Handel-C Compiler, September 2003.

[15] Mark L. Chang. Adaptive computing in NASA multi-spectral image processing. Master's thesis, Northwestern University, Dept. of ECE, December 1999.

[16] Mark L. Chang and Scott Hauck. Adaptive computing in NASA multi-spectral image processing. In *Military and Aerospace Applications of Programmable Devices and Technologies International Conference*, 1999.

[17] Mark L. Chang and Scott Hauck. Précis: A design-time precision analysis tool. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 229–238, 2002.

[18] Mark L. Chang and Scott Hauck. Précis: A design-time precision analysis tool. In *Earth Science Technology Conference*, 2002.

[19] Mark L. Chang and Scott Hauck. Variable precision analysis for FPGA synthesis. In *Earth Science Technology Conference*, June 2003.

[20] Mark L. Chang and Scott Hauck. Automated least-significant bit datapath optimization for FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.

[21] Wang Chen, Panos Kosmas, Miriam Leeser, and Carey Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *ACM/SIGDA Twelfth ACM International Symposium on Field-Programmable Gate Arrays*, pages 213–222, 2004.

[22] Samir R. Chettri, Robert F. Cromp, and Mark Birmingham. Design of neural networks for classification of remotely sensed imagery. *Telematics and Informatics*, 9(3-4):145–156, 1992.

[23] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. A methodology and design environment for DSP ASIC fixed point refinement. In *Design Automation and Test in Europe*, 1999.

[24] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[25] George Constantinides. Perturbation analysis for word-length optimization. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[26] George A. Constantinides, Peter Y.K. Cheung, and Wayne Luk. Heuristic datapath allocation for multiple wordlength systems. In *Design Automation and Test in Europe*, 2001.

[27] George A. Constantinides, Peter Y.K. Cheung, and Wayne Luk. The multiple wordlength paradigm. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[28] George A. Constantinides, Peter Y.K. Cheung, and Wayne Luk. Optimum wordlength allocation. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[29] Altera Corporation. *Stratix Device Handbook, Volume 1.* Altera Corporation, San Jose, California, 2003.

[30] Adam J. Elbirt, W. Yip, Brendon Chetwynd, and Christof Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on VLSI*, 9(4):545–557, 2001.

[31] Barry Fagin and Cyril Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on VLSI Systems*, 2(3), September 1994.

[32] Claire Fang Fang, Rob A. Rutenbar, and Tsuhan Chen. Floating-point bit-width optimization for low-power signal processing applications. In *International Conf. on Acoustic, Speech and Signal Processing*, 2002.

[33] Claire Fang Fang, Rob A. Rutenbar, and Tsuhan Chen. Fast, accurate static analysis for fixed-point finite precision effects in DSP designs. In *International Conf. on Computer Aided Design*, 2003.

[34] Claire Fang Fang, Rob A. Rutenbar, Markus Püschel, and Tsuhan Chen. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. In *Design Automation Conference*, 2003.

[35] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the Streams-C C-to-FPGA compiler: An applications perspective. In *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, pages 134–140, 2001.

[36] Thomas W. Fry. Hyperspectral image compression on reconfigurable platforms. Master's thesis, University of Washington, Seattle, WA, May 2001.

[37] Thomas W. Fry and Scott Hauck. Hyperspectral image compression on reconfigurable platforms. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 251–260, 2002.

[38] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[39] Brian Gladman. Implementation experience with aes candidate algorithms. In *Second AES Candidate Conference*, March 1999.

[40] Maya Gokhale, Jan Stone, and Jeff Arnold. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–56, 2000.

[41] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, 1991.

[42] John L. Hennessey and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface; 2nd edition*. Morgan Kaufmann, 1994. HEN j2 98:1 P-Ex.

[43] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[44] Jordan L. Holt and Jenq-Neng Hwang. Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, 42(3):281–290, March 1993.

[45] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. In *Workshop on VLSI and Signal Processing*. Osaka, 1995.

[46] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems II*, 45(11):1455–1464, Nov 1998.

[47] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 13 1983.

[48] Donald E. Knuth. *The TEXbook*. Addison–Wesley, third edition, 1986.

[49] Helmut Kopka and Patrick W. Daly. *A Guide to LATEX2e*. Addison–Wesley, second edition, 1995.

[50] Ki-Il Kum and Wooyong Sung. VHDL based fixed-point digital signal processing algorithm development software. In *Proceedings of the IEEE International Conference on VLSI CAD*, pages 257–260, November 1993.

[51] Tom Lane, Philip Gladstone, Lee Crocker Jim Boucher, Julian Minguillon, Luis Ortiz, George Phillips, Davide Rossi, Guido Vollbeding, and Ge' Weijers. The independent JPEG group's JPEG software library. http://www.ijg.org/files/jpegsrc.v6b.tar.gz, June 2004.

[52] Walter B. Ligon III, Scott McMillan, Greg Monn, Kevin Schoonover, Fred Stivers, and Keith D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[53] Y.C. Lim. Single-precision multiplier with reduced circuit complexity for signal processing applications. *IEEE transactions on Computers*, 41(10):1333–1336, October 1992.

[54] Christoph Loeffler, Adriaan Ligtenberg, and George S. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *Proc. Int'l. Conf. on Acoustics, Speech, and Signal processing*, pages 988–991, 1989.

[55] Scott Mahlke, Rajiv Ravindran, Michael Schlansker, Robert Schreiber, and Timothy Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1355–1371, November 2001.

[56] Cleve B. Moler. MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science, 1980.

[57] Cleve B. Moler. MATLAB user's guide. Technical report, University of New Mexico. Dept. of Computer Science, November 1980.

[58] Stefan Müller and Heinz Waller. Efficient integration of real-time hardware and web based services into MATLAB. In *11th European Simulation Symposium And Exhibition Simulation In Industry*, October 1999.

[59] Anshuman Nayak, Malay Haldar, et al. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. In *Design Automation & Test*, March 2001.

[60] Louis B. Rall. Automatic differentiation: Techniques and applications, 1981.

[61] Michael J. Schulte and Jr. Earl E. Swartzlander. Truncated multiplication with correction constant. In *VLSI Signal Processing VI, IEEE Workshop on VLSI Signal Processing*, pages 388–396, October 1993.

[62] Michael J. Schulte and James E. Stine. Reduced power dissipation through truncated multiplication. In *Proceedings of the IEEE Alessandro Volta Memorial International Workshop on Low Power Design*, pages 61–69, March 1999.

[63] Michael J. Schulte, Kent E. Wires, and James E. Stine. Variable-correction truncated floating point multipliers. In *Proceedings of the Thirty Fourth Asilomar Conference on Signals, Systems, and Computers*, pages 1344–1348, November 2000.

[64] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. Technical Report LCS-TM-602, Massachussets Institute of Technology, November 1999.

[65] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, June 2000.

[66] Mark William Stephenson. Bitwise: Optimizing bitwidths using data-range propagation. Master's thesis, Massachusetts Institute of Technology, May 2000.

[67] Wonyong Sung and Ki-Il Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE transactions on Signal Processing*, 43(12):3087–3090, December 1995.

[68] Wooyong Sung and Ki-Il Kum. Word-length determination and scaling software for signal flow block diagram. In *International Conference on Acoustic, Speech, and Signal Processing*, pages 457–460, Adelaide, Australia, 1994.

[69] Katsuharu Suzuki, Michael X. Wang, Zhao Fang, and Wayne W.M. Dai. Design of c++ class library and bit-serial compiler for variable-precision datapath synthesis on adaptive computing systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[70] Synopsys. Synopsys CoCentric SystemC Compiler, September 2003.

[71] Stephen M. Trimberger, editor. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, 1999.

[72] K.H. Tsoi, K.H. Lee, and P.H.W. Leong. A massively parallel rc4 key search engine. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–21, 2002.

[73] Jack Volder. The CORDIC trigonometric computing technique. *IRE Trans. Electronic Computing*, EC-8:330–334, September 1959.

[74] Markus Willems, Volker Bürsgens, Holger Keding, Thorsten Grötker, and Heinrich Meyr. System level fixed-point design based on an interpolative approach. In *Design Automation and Test in Europe*, 1997.

[75] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.W. Liao, C.W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), 1996.

[76] Kent E. Wires, Michael J. Schulte, and Don McCarley. FPGA resource reduction through truncated multiplication. In *Proceedings of the 11th International Conference on Field Programmable Logic and Applications*, pages 574–583, August 2001.

[77] Xilinx Inc. XC6200 field programmable gate arrays. Datasheet, April 24 1997. Version 1.10.

[78] Xilinx Inc. *The programmable logic data book 2000.* Xilinx Inc., San Jose, California, 2000.

[79] Xilinx, Inc., San Jose, California. *Virtex-II Pro^{TM}Platform FPGA User Guide*, March 2003.

[80] Xilinx, Inc. Virtex-II Pro$^{\mathrm{TM}}$Platform FPGAs: Introduction and Overview, March 2003.

[81] Xilinx, Inc., San Jose, California. *Virtex^{TM}-II Platform FPGAs: Detailed Description*, June 2003.

VITA

Education

Ph.D., Electrical Engineering, University of Washington, Seattle, WA, 2004.
Thesis: *Variable Precision Analysis for FPGA Synthesis.*
Advisor: Scott Hauck.

M.S., Electrical and Computer Engineering, Northwestern University, Evanston, IL, 2000.
Thesis: *Adaptive Computing in NASA Multi-Spectral Image Processing.*

B.S. with University and Departmental Honors, Electrical and Computer Engineering, The Johns Hopkins University, Baltimore, MD, 1997.

Research Interests

FPGA Architectures, Applications, and Tools; Reconfigurable Computing; Operating System Integration of Reconfigurable Computing; Computer Architecture; VLSI Design.

Awards

Intel Corporation: 2002-2003 Intel Foundation Graduate Fellowship.

University of Washington: Outstanding Graduate Research Assistant (2002), Nominated for the 2002 Yang Research Award.

Northwestern University: Royal E. Cabell Fellowship (1997), ECE Department Best Teaching Assistant Honorable Mention (1998/1999).

Johns Hopkins University: IEEE student chapter President (1996), Eta Kappa Nu chapter President (1996-1997), Tau Beta Pi, Dean's List, Electrical and Computer Engineering Chair Award.

National Merit Scholar, National Computer Systems Merit Scholarship.

Employment

Franklin W. Olin College of Engineering, Needham, MA
08/2004 – *Present* Assistant Professor.

University of Washington, Seattle, WA
07/2000 – 07/2004 Research assistant. Developing variable precision design tools for FPGAs.

Quicksilver Technologies, Inc., Seattle, WA
07/2001 – 10/2001 Java software developer. Assisted development of software development tools for Quicksilver's reconfigurable hardware.

Northwestern University, Evanston, IL
09/1997 – 06/2000 Research assistant developing FPGA implementations of NASA image processing applications.

National Computer Systems, Iowa City, IA
06/1997 – 08/1997 Customer service operator for the Department of Education.

Johns Hopkins University, Baltimore, MD
10/1994 – 06/1997 Undergraduate research assistant in the Parallel Computing and Imaging Laboratory.
03/1995 – 06/1997 Assistant System Administrator for the Center for Language and Speech Processing.
05/1995 – 01/1997 Maryland Space Grant Consortium webmaster.

Anton-Paar, GmbH, Graz, Austria
06/1996 – 07/1996 Participated in a cooperative internship with the Technical University of Graz, Austria. Developed embedded software for use in concentration determination instruments.

Products Unlimited, Corp., Iowa City, IA
Summers 1989–1994 Programmer and technician. Set up and maintained a network of PCs for a small office. Developed computer-aided testing facilities using IEEE-488 instruments and hardware.

Consulting

NetFrameworks, Inc. & Applied Minds, Inc. Consultant and primary software developer for proprietary groupware system. July – September, 2001.

Hunter Benefits Group. Lead software developer, September, 2000.

HumaniTree.com, LLC. Web developer and Java programmer. December, 1998 – March, 1999.

Publications

Mark L. Chang, Scott Hauck, "Automated Least-Significant Bit Datapath Optimization for FPGAs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, January, 2004.

Mark L. Chang, Scott Hauck, "Least-Significant Bit Optimization Techniques for FPGAs", poster presented at *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February, 2004.

Mark L. Chang, Scott Hauck, "Précis: A Design-Time Precision Analysis Tool", submitted to *IEEE Design and Test of Computers*, 2003.

Mark L. Chang, Scott Hauck, "Variable Precision Analysis for FPGA Synthesis", *Earth Science Technology Conference*, June, 2003.

Mark L. Chang, Scott Hauck, "Précis: A Design-Time Precision Analysis Tool", *Earth Science Technology Conference*, June, 2002.

Mark L. Chang, Scott Hauck, "Précis: A Design-Time Precision Analysis Tool", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 229–238, 2002.

Mark L. Chang, *Adaptive Computing in NASA Multi-Spectral Image Processing*, M.S. thesis, Northwestern University, Dept. of ECE, December, 1999.

Mark L. Chang, Scott Hauck, "Adaptive Computing in NASA Multi-Spectral Image Processing", *Military and Aerospace Applications of Programmable Devices and Technologies International Conference*, 1999.

P. Banerjee, A. Choudhary, S. Hauck, N. Shenoy, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, "MATCH: A MATLAB Compiler for Adaptive Computing Systems", *Northwestern University Department of Electrical and Computer Engineering Technical Report CPDC-TR-9908-013*, 1999.

Teaching

University of Washington, Seattle, WA

EE 471: Computer Design and Organization. Instructor, Winter 2003. Overall class evaluation rating 4.13/5.0.

Northwestern University, Evanston, IL

B01: Introduction to Digital Logic Design. Instructor, Summer 1999. Overall class evaluation rating 5.56/6.0.

C91: VLSI Systems Design. Teaching Assistant, Winter 1999.

C92: VLSI System Design Projects. Teaching Assistant, Spring 1998.