Imperial College London
Department of Computing

Application-Specific Number Representation

Haohuan Fu

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing
and the Diploma of Imperial College London, February 2009

Abstract

Reconfigurable devices, such as Field Programmable Gate Arrays (FPGAs), enable application-specific number representations. Well-known number formats include fixed-point, floating-point, logarithmic number system (LNS), and residue number system (RNS). Such different number representations lead to different arithmetic designs and error behaviours, thus producing implementations with different performance, accuracy, and cost.

To investigate the design options in number representations, the first part of this thesis presents a platform that enables automated exploration of the number representation design space. The second part of the thesis shows case studies that optimise the designs for area, latency or throughput from the perspective of number representations.

Automated design space exploration in the first part addresses the following two major issues:

- **Automation requires arithmetic unit generation.** This thesis provides optimised arithmetic library generators for logarithmic and residue arithmetic units, which support a wide range of bit widths and achieve significant improvement over previous designs.
- **Generation of arithmetic units requires specifying the bit widths for each variable.** This thesis describes an automatic bit-width optimisation tool called R-Tool, which combines dynamic and static analysis methods, and supports different number systems (fixed-point, floating-point, and LNS numbers).

Putting it all together, the second part explores the effects of application-specific number representation on practical benchmarks, such as radiative Monte Carlo simulation, and seismic imaging computations. Experimental results show that customising the number representations brings benefits to hardware implementations: by selecting a more appropriate number format, we can reduce the area cost by up to 73.5% and improve the throughput by 14.2% to 34.1%; by performing the bit-width optimisation, we can further reduce the area cost by 9.7% to 17.3%. On the performance side, hardware implementations with customised number formats achieve 5 to potentially over 40 times speedup over software implementations.

Acknowledgements

First of all, I would like to express my great thanks to my supervisors Dr. Oskar Mencer and Prof. Wayne Luk, who have provided priceless support and guidance during my PhD studies.

I'd like to thank Dr. Chak-Chung Cheung, Dr. Dong-U Lee, and Dr. Altaf Abdul Gaffar for their valuable advice on my research work. Their excellent efforts on bit-width optimisation provide a concrete base for my tool that supports bit-width analysis across multiple number representations. Dr. Dong-U Lee's great ideas on optimising hardware function evaluation provide me the basic techniques to develop arithmetic libraries for various number representations.

I want to thank Dr. Robert G. Clapp for his great expertise on both geophysics and computer science. During my research project in the Geophysics department of Stanford University, he provided me with a lot of guidance on using seismic software libraries, and taught me about the seismic concepts behind the numerous computations. His great idea on evaluating the accuracy of seismic images with prediction error filters enables my tool to automatically explore different number precisions in seismic imaging computations.

I also want to thank Dr. Tim Todman and Mr. Craig Court for proof-reading my thesis draft.

Great thanks to all my lovely colleagues in Imperial College London, Mr. Chun Hok Ho, Mr. Chi Wai Yu, Dr. Yuet-Ming Lam, Dr. Qiang Wu, Dr. Kubilay Atasu, Mr. Carlos Tavares, Mr. Lee Howes, Mr. Min Li, and Mr. Yizhou Wang. They have provided a lot of help, and brought a lot of happiness to my life at Imperial.

The financial support from Overseas Research Students Award Scheme and UK Engineering and Physical Sciences Research Council (grant number EP/C509625/1) is gratefully acknowledged.

Dedication

To my wife,

who has shared with me all the good and bad bits of life...

And to my parents,

who are always there to guide and support me with their love...

Abbreviations

ALU	Arithmetic and Logic Unit
ASA	Adaptive Simulated Annealing
ASC	A Stream Compiler
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM (a memory unit on commercial FPGAs)
BTFP	Better-Than-Floating-Point
NBTFP	Not-Better-Than-Floating-Point
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DCT	Discrete Cosine Transform
DI	Difference Indicator
DSCG	Digital Sine/Cosine Waveform Generator
DSP	Digital Signal Processor
DSR	Double Square Root
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FIX	Fixed-point number
FLP	Floating-point number
FPGA	Field Programmable Gate Arrays
GPP	General Purpose Processor
HMUL	Devoted Hardware Multipliers on Commercial FPGAs
IPP	Intellectual Property Protection
LNS	Logarithmic Number System
MAC	Multiplication and Accumulate Operation
MM	Matrix Multiplication
MR	Mixed-Radix
NaN	Not a Number
PEF	Prediction Error Filter
PML	Perfectly Matched Layer
QRD	QR Decomposition
RMCS	Radiative Monte Carlo Simulation
RNS	Residue Number System
ROM	Read-Only Memory
RTL	Register Transfer Level
SQRT	Square Root
ulp	unit in the last place
VHDL	VHSIC (Very-High-Speed Integrated Circuits) Hardware Description Language
XST	Xilinx Synthesis Technology

List of Publications

Journal Papers:

- H. Fu, W. Osborne, R. Clapp, O. Mencer, and W. Luk, "Accelerating Seismic Computations Using Customized Number Representations on FPGAs", submitted to *EURASIP Journal on Embedded Systems, Special Issues on FPGA Supercomputing Platforms, Architectures and Techniques for Accelerating Computationally Complex Algorithms*.
- H. Fu, O. Mencer, and W. Luk, "FPGA Designs with Optimized Logarithmic Arithmetic", submitted to *IEEE Transactions on Computers*.
- H. Fu, A. Gaffar, O. Mencer, and W. Luk, "Bit-width Analysis Across Multiple Number Representations", submitted to *IEEE Transactions on Very Large Scale Integration Systems*.

Conference Papers:

- H. Fu, O. Mencer, and W. Luk, "Optimizing Residue Arithmetic on FPGAs", *Proc. Field Programmable Technology (FPT)*, Best Paper Award, 2008.
- T. Todman, H. Fu, K. Tsoi, O. Mencer, and W. Luk, "Smart Enumeration: a Systematic Approach to Exhaustive Search", *Proc. International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*,
- H. Fu, W. Osborne, R. Clapp, and O. Pell, "Accelerating Seismic Computations on FPGAs: from the Perspective of Number Representations", extended abstract, *Proc. 70th EAGE Conference and Exhibition incorporating SPE EUROPEC*, 2008.
- T. Todman, H. Fu, O. Mencer, and W. Luk, "Improving Bounds for FPGA Logic Minimization", poster, *Proc. Field Programmable Technology (FPT)*, pages 245-248, 2007.
- H. Fu, O. Mencer, and W. Luk, "Optimizing Logarithmic Arithmetic on FPGAs", *Proc. Field-Programmable Custom Computing Machines (FCCM)*, pages 163-172, 2007.
- H. Fu, O. Mencer, and W. Luk, "Comparing Floating-point and Logarithmic Number Representations for Reconfigurable Acceleration", poster, *Proc. Field Programmable Technology (FPT)*, pages 337-340, 2006.

Contents

Part I An Automated Exploration Platform

Part II Effects of Application-Specific Number Representation

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Application-Specific Number Representation

For mathematicians, number representation is about different methods of expressing numbers with symbols. For computer scientists, who work with machines operating on '0's and '1's, number representation becomes a design problem about how to express different values with a limited number of bits.

Different number representations involve different rules to interpret the bits into mathematical values. The different interpretation rules lead to different ranges and accuracies for the represented value. Moreover, number representation does not only specify how to represent values, but also determines how to compute with the values, i.e. how to perform arithmetic operations. Thus, computations using different number representations lead to different operations on the bits, which produce different costs, performances, and accuracies.

An application is in essence a collection of computations that operate on certain number representations. For a target application, **application-specific number representation** refers to a number representation that is specifically customised to satisfy its cost, performance, and accuracy requirements.

This thesis presents my work on investigating application-specific number representation on

FPGAs. We develop an automated design space exploration platform that can automatically explore different number representations (Chapter 3). The exploration platform supports optimised arithmetic unit generation (Chapters 4 and 5) and bit-width optimisation for different number representations (Chapter 6). By applying the platform on practical applications or computation kernels (Chapters 7 and 8), we show that application-specific number representation can bring benefits to area cost, accuracy, and throughput of various applications. The exploration of different number representations also help us to understand the characteristics of different number systems, and the tradeoffs between different metrics and constraints.

1.2 Motivations

In the field of computer science, there are different categories of methods to implement a target application [CH02].

As shown in Figure 1.1, the first category of methods implements the algorithms with hardware, such as the Application-Specific Integrated Circuits (ASICs). With an electronic chip or a board made up of a number of components, the customised hardware solution can achieve almost the best performance for the target application. However, no modifications can be made to an ASIC after its fabrication. A circuit needs to be re-designed and re-fabricated in case of bugs or any change of requirements. Moreover, the cost for putting a specific application into a chip

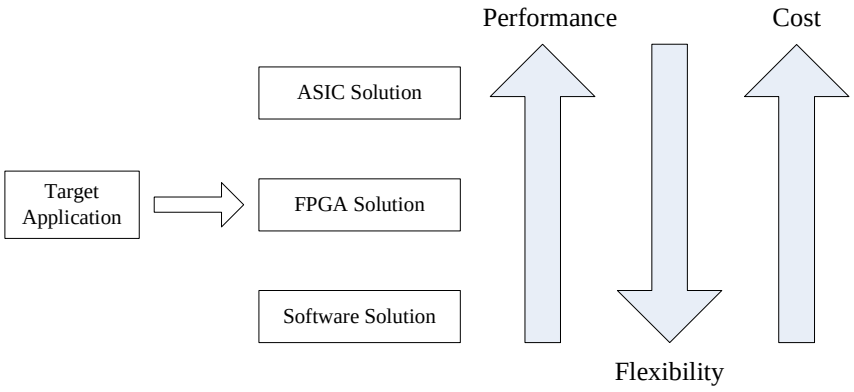


Figure 1.1: Different categories of methods to implement a target application.

or a circuit board is extremely expensive and usually takes a long period.

In contrast to an ASIC solution, we can also implement the design in software. With a processor that supports a large set of common instructions, the application can be implemented with software programs. As the processor is designed for a general use, but not customised for the specific application as in the hardware solution, the performance is poorer. However, the software solution provides complete flexibility when you need to make modifications and costs much less.

Reconfigurable devices, such as Field Programmable Gate Arrays (FPGAs), have emerged in the gap between the hardware and software approaches.

An FPGA contains an array of programmable logic components, such as 4-input or 6-input look-up tables. These logic components can be configured to perform arbitrary arithmetic or logic operations on the input bits. There is also a hierarchy of programmable interconnections, which can be used to connect the programmed logic components. By programming both the logic components and interconnections, the users can map various designs onto the FPGA.

With logics that are specially configured for a specific application, an FPGA can achieve much better performance than software. Meanwhile, it provides a more flexible interface than an ASIC. A new configuration can be loaded into the FPGA board in several seconds at run-time. The cost for design and implementation is also much less.

In general, ASIC solutions provide no re-programmability and incur high implementation costs, while software solutions are constrained by the inherent pre-defined number representation. Therefore, the FPGA becomes the only practical platform that supports customisation of number representations and enables application-specific number representation.

The FPGA users can apply different number systems, such as fixed-point, floating-point, logarithmic number system (LNS) [ABCC90], residue number system (RNS) [Tom06], and even hybrid number systems. The circuit designs on FPGAs also enjoy the freedom on bit widths. While users of common processors can only use predefined bit widths such as 16-bit, 32-bit, and 64-bit integers or floating-point numbers, the FPGA users can apply arbitrary bit widths

for each variable in the design.

While the FPGA provides the above design freedom on number representations, it also brings the problem of selecting a number representation that fits a given application the best. As different number representations lead to different complexities and architectures of the resulting FPGA implementation, choosing a more appropriate number representation may bring great benefits to the cost, performance, or accuracy of the application.

Based on the above observations, the major motivations of this thesis can be summarised as follows:

- employ reconfigurable devices, such as FPGAs, to explore the design options related to number representations.
- analyse the exploration results to study the characteristics of different number representations, as well as the various tradeoffs between different solutions to the same design.
- achieve benefits on cost, performance, or accuracy by using application-specific number representations.

1.3 Constraints and Metrics

When exploring an application with different number representations, various aspects need to be considered, either as a constraint to meet or as a metric to optimise in the design. In this work, the following four items are studied as the major constraints or metrics during the exploration process:

- **AREA** of the design. In the cases where the design only consumes look-up tables and registers on the FPGA, the area is simply described in the number of slices the design takes. If the design also consumes other categories of resources, such as hardware multipliers, block RAMs, and microprocessors, we generally apply certain ratios to convert the special-purpose units into a number of logic slices.

- **ACCURACY** of the design. In normal cases, the accuracy can be described by the error values of the design, such as the maximum/average absolute or relative errors. For the cases where the application produces images containing pattern information, the accuracy needs to be evaluated in some other ways (an example is shown in Section 8.3.3).
- **THROUGHPUT** of the design, described in the number of results the design produces every second. This item describes how fast the computation can be performed, and provides an indication of the performance of the design.
- **LATENCY** of the design, which is the delay between the time point that the input is fed in and the time point that the corresponding output comes out. While the throughput is the maximum frequency that the design can run at, the latency corresponds to the time gap between corresponding input and output signals.

There are still other metrics/constraints that also relate to the circuit design and vary for different number representations, such as the power consumption of the design [GCC06]. However, this work focuses on investigating the above four items.

In general, based on users' different requirements and applications' different characteristics, we can apply different constraints and metrics to figure out the optimal number representation system. For instance, area of the design is normally used as a constraint in practical FPGA designs, as the design shall at least fit into the board. Under the constraint of the available resources on an FPGA board, we can customise the number representation to achieve maximum throughput of the application. In some other cases, we may set the the accuracy as the constraint and require the absolute error of the output to be less than a certain value, while area, throughput or latency become the metrics that we strive to optimise.

1.4 Summary of My Work

Different number representations lead to different complexities and architectures of the resulting FPGA implementations. For engineers trying to implement various applications on FPGA, it

becomes a vital first step to determine the number representations for the variables. Choosing the most appropriate number format can generally bring significant benefits on area, accuracy, or performance of the design.

To investigate the number-representation design options, the first part of this thesis presents a platform that enables automated exploration of the number-representation design space. The platform addresses the following two major issues:

- **Automation requires arithmetic unit generation.** This thesis provides optimised arithmetic library generators for logarithmic and residue arithmetic units. The generators support a wide range of bit widths and in most cases provide significant area/latency reduction or performance improvement over previous arithmetic designs. When compared with previous LNS designs [HBUH05], our generated units typically provide 6% to 37% reduction in area and 20% to 50% reduction in latency (Section 4.3.6). For reverse converters from RNS to binary numbers, we propose a novel design that uses only n-bit additions and consumes 14.3% less area than previous work [WSAS02] (Section 5.2.3).

- **Generation of arithmetic units requires specifying the bit widths for each variable.** To address this problem, this thesis shows an automatic bit-width optimisation tool called R-Tool. R-Tool combines dynamic and static analysis methods to provide optimised bit widths for different guarantees of accuracy (Section 6.3).

We use A Stream Compiler (ASC) [Men06] as our hardware compilation tool. ASC supports automated generation of fixed-point and floating-point arithmetic units, which provide a competitive performance to commercial arithmetic cores. Combining ASC with our library generators, our platform is able to explore a given application for different implementations using different number representations.

Putting all the above tools and techniques together, the second part of the thesis explores the effects of application-specific number representation with practical benchmarks, such as radiative Monte Carlo simulations, and seismic imaging applications.

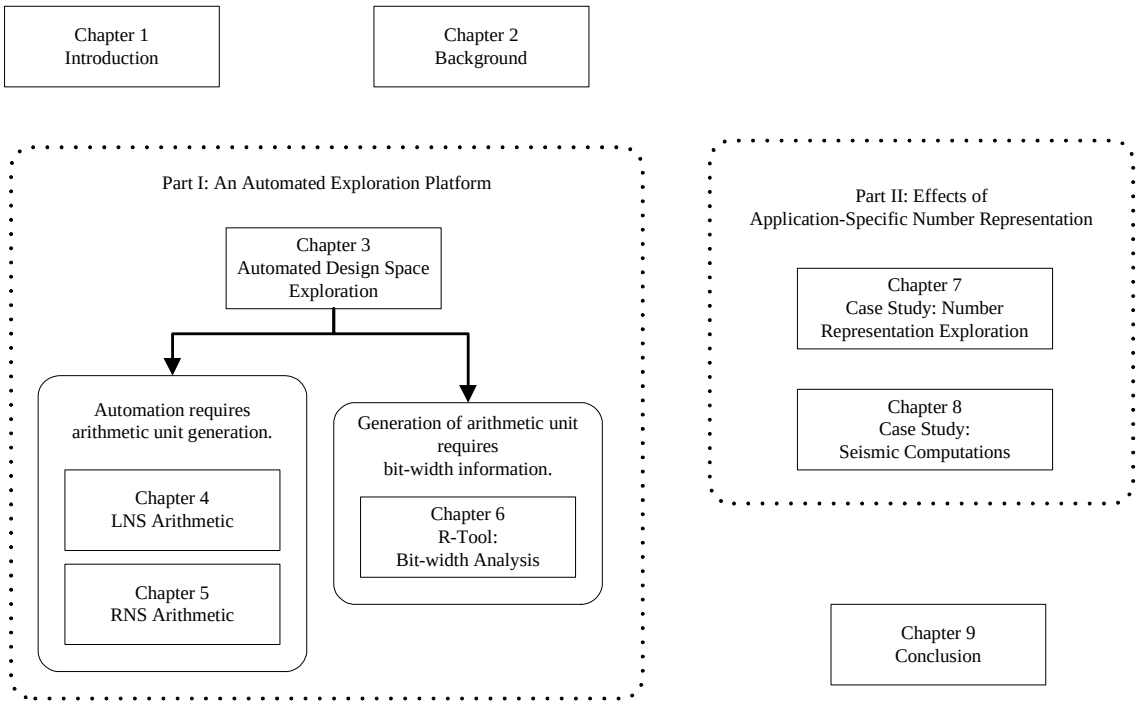


Figure 1.2: Structure of the thesis.

Experimental results show that customising the number representations brings benefits to applications on FPGAs: by selecting a more appropriate number format, such as switching from floating-point to LNS, or from fixed-point to RNS, we can reduce the area cost by up to 73.5% (Section 7.2) and improve the throughput by 14.2% to 34.1% (Sections 7.2 and 7.3); by performing bit-width optimisation, we can further reduce the area cost by 9.7% to 17.3% (Section 7.4). On the performance side, FPGA implementations with customised number formats achieve 5 to potentially over 40 times speedup over software implementations (Section 8.4).

As shown in Figure 1.2, the rest of the thesis is organised as follows:

Chapter 2 gives a thorough introduction to the related background and a literature review of the previous work.

Chapter 3, 4, 5, and 6 make up Part I of the thesis, which presents the automated exploration tool for number representations. Within them, Chapter 3 describes the general architecture and basic techniques. Chapter 4 and 5 present my work on optimised library generator for LNS and RNS arithmetic units. The automated bit-width optimisation tool called R-Tool is described

in Chapter 6. The tool targets streaming circuit designs with determined error requirements.

Part II of the thesis consists of Chapters 7 and 8, and shows the advantages of application-specific number representation with case studies of practical applications. Chapter 7 demonstrates case studies of comparing different number representations on practical applications, and bit-width optimisation in feed-forward circuit designs. Chapter 8 shows some real challenges for the computation bottlenecks in seismic processing, and the proposed FPGA solutions using customised number formats.

Chapter 9 concludes the thesis and discusses potential topics to investigate in future work.

Chapter 2

Background

2.1 Introduction

This chapter provides background information related to my work. Section 2.2 introduces the structure of FPGA devices, the tools for programming FPGAs, the applications that we can accelerate on them, and the different number representations we can use. Section 2.3 discusses the previous efforts on automation of design space exploration. Section 2.4 describes some more details about arithmetic unit generation, which is the first key element in our automated exploration of number representation. The other key element, bit-width optimisation, is discussed in Section 2.5. Section 2.6 summarises this chapter.

2.2 Reconfigurable Computing on FPGAs

2.2.1 General Hardware Architecture of FPGAs

Generally, an FPGA consists of reconfigurable function units, reconfigurable interconnections, and an interface that connects the reconfigurable device to the rest of the system [TCW+05]. According to the configuration bits the users set, the reconfigurable function units perform

the required processing operations while the reconfigurable interconnections produce required routes between the input and output ports of the functions units. The configured function units together with the configured routes make up the circuit design required by the users.

Reconfigurable Function Units: Fine-grained and Coarse-grained

Based on the flexibility and the number of bits they handle, the reconfigurable function units on the FPGA are classified as fine-grained or coarse-grained. Typically, fine-grained units provide processing of a small number (3 to 6) of bits, and can be configured to be any kind of function of the input bits. Examples are the 4-input or 6-input lookup tables widely used on the commercial FPGAs of Xilinx and Altera [Inc05], [Alt06]. On the other hand, coarse-grained units process a larger number of bits and the configurations of the function are less flexible. Coarse-grained units are usually used as Arithmetic and Logic Units (ALUs) and large storage units. Typical examples of coarse-grained units are the devoted 18-by-18 multipliers and 18-KBit Block RAMs on the commercial Xilinx FPGAs. The combination of both fine-grained and coarse-grained units provides balanced resources for users to implement common applications on FPGAs.

Reconfigurable Interconnections

Similar to function units, reconfigurable interconnections can also be categorised as fine-grained or coarse-grained according to the scale of bits they handle [TCW+05]. Fine-grained interconnections usually provide a configuration interface for each of the wires that connects every bit, while coarse-grained interconnections usually handle a number of wires together as an entire configurable bus. Most of the current commercial FPGAs use the fine-grained interconnection architecture.

Communication between the FPGA and the Processors

In practical applications, the FPGA is normally used to accelerate a computationally intensive part of the entire program. Thus, the FPGA needs to communicate with the other parts of the program that are executed on processors, such as CPUs or digital signal processors (DSPs). Due to the high communication latency, we generally try to avoid feedback between FPGAs and processors. The ideal case is to apply a streaming computation pattern, in which only

inputs and outputs are streamed in and out of the FPGA.

Nowadays, some commercial FPGAs already provide a microprocessor in the device, such as the PowerPC processor on the FPGA boards of Xilinx [Inc03b], or the MicroBlaze [Inc04] soft core simulated using the programmable slices. In this way, the programmable units and the processors are more tightly coupled, and the communication cost becomes acceptable.

2.2.2 FPGA Design Tools

The process of implementing an application on an FPGA can be divided into the following four basic steps:

- Describe the application using arithmetic or behavioural statements, such as additions, subtractions, multiplications and reading/writing memories.
- Synthesise the arithmetic or behavioural description into AND/OR gates and flip-flops.
- Map the gates and flip-flops into the programmable function units on the FPGA, such as fine-grained 4-input lookup tables and coarse-grained 18-by-18 hardware multipliers.
- Place and route the design onto the FPGA chip.

Traditionally, in the first step, users describe the design in a Register Transfer Level (RTL) hardware description language, such as VHDL or Verilog. However, there are also compilers that support high-level hardware description languages derived from C++/Java. Examples are Streams-C [GSAK00], Handel-C [Cel03], ASC [Men06]. Streams-C converts the C++ description into RTL VHDL, using its special libraries. Handel-C converts a C++-like description into structural VHDL, Verilog or a net-list file. ASC, which is the tool used in our platform, converts a subset of C++ code into a net-list file.

A more recent example of a high-level hardware compiler is Optimus [HKB+08], which describes the hardware design using a streaming language, called StreamIt [TKA02]. In this language, the basic computation modules are described as filters, while the connections between the filters

are described as pipelines. Similar to Streams-C, Optimus converts the structure of filters and pipelines into a description of hardware description language (HDL) for later synthesis.

If the tool already generates a net-list file, the synthesis is already done. Otherwise, a synthesis tool, such as XST (Xilinx Synthesis Technology) of Xilinx [Inc07], and Synplify of Synplicity [Syn06], can be used to synthesise the VHDL description into a net-list file.

The tools for mapping, placing and routing in the third and forth steps are generally provided by the merchandiser of the commercial FPGAs, such as the MAP and PAR programs of Xilinx [Inc03a].

2.2.3 Application Acceleration on FPGAs

In recent years, FPGAs have shown excellent performance on accelerating different applications in image processing, cryptography, and scientific computations.

M. Leeser et al. [LMY04] demonstrates a novel image processing architecture by connecting a high-quality video camera to an FPGA processing board. Using FPGA to accelerate the image processing, the two applications from medical image processing and computational fluid dynamics both exhibit speedups of more than 20 times compared to software implementations on a 1.5 GHz PC platform.

In cryptography applications, using a reconfigurable device of Xilinx Virtex-II XC2V6000 FPGA, the point multiplication used in elliptic curve cryptography can be computed at the speed of 66 MHz [CTLC05], while an optimised software implementation requires 196.71 ms. This illustrates a 540 times acceleration by using reconfigurable devices over software.

FPGAs also demonstrate good performance in scientific simulations. J. L. Tripp et al. [TMHG05] present a road traffic simulation of entire metropolitan areas with a reconfigurable architecture combining 64-bit processors and FPGAs. A 34.4 times speed up is achieved using one FPGA on the Cray XD1 supercomputer, compared with the software version on AMD microprocessors.

A more recent example is an FPGA solution for radiation dose calculation [WHYC06]. The

3-D convolution/superposition "collapsed cone" algorithm is implemented on an FPGA, with some modifications and novel design techniques. The implementation shows an overall speedup of 20.7 times over software versions.

As shown in the above examples, with reconfigurable function units and interconnections, FPGAs generally provide a more customised solution for computationally intensive applications, and bring a significant improvement in performance.

2.2.4 Different Number Representation Systems on FPGAs

The work presented in this thesis focuses on four major number representation systems on FPGAs: fixed-point, floating-point, logarithmic number system (LNS), and residue number system (RNS).

- Fixed-point numbers:

  A fixed-point number format has two parts, the integer part and the fractional part. It is in a format as follows:

integer part: m bits	fractional part: f bits
$x_{m-1}x_{m-2}...x_0$	$x_{-1}...x_{-f+1}x_{-f}$

  When it uses a sign-magnitude format (the first bit defines the sign of the number), its value is given by $(-1)^{x_{m-1}} \cdot \sum_{i=-f}^{m-2} x_i \cdot 2^i$. It may also use a two's complement format to indicate the sign, with the value given by $\sum_{i=-f}^{m-2} x_i \cdot 2^i - x_{m-1} \cdot 2^{m-1} + 1$.

- Floating-point numbers:

  According to IEEE-754 standard [otICS08], floating-point numbers can be divided into three parts: the sign bit, the exponent and the significand, shown as follows:

sign: 1 bit	exponent part: m bits	significand part: f bits
S	M	F

Their values are given by $(-1)^S \times 1.F \times 2^M$. The sign bit defines the sign of the number. The exponent part uses a biased format. Its value equals the sum of the actual exponent value and the bias, which is defined as $2^{m-1} - 1$. The extreme values of the exponent (0 and $2^m - 1$) are used to indicate special cases, such as values of zero and $\pm\infty$. The significand is an unsigned fractional number, with an implied '1' to the left of the radix point.

- LNS numbers:

  Unlike the floating-point numbers defined by IEEE 754 standard, there is no commonly accepted standard for LNS numbers. In this thesis, the signed-logarithmic representation format similar to [ABCC90] is used. The format consists of a sign bit and a fixed-point number to record the logarithmic value, shown as follows:

sign bit	integer part: m bits	fractional part: f bits
S	M	F

  Its value is given by $(-1)^S \times 2^{M.F}$, which provides a similar representation range to floating-point numbers with m-bit exponent, f-bit significand and one sign bit. Note that the integer part and the fractional part are combined into a fixed-point value. The fixed-point value indicates the logarithmic value, and has its own sign, i.e. $M.F$ can be either positive or negative.

- RNS numbers:

  Assume that $\{M_1, M_2, \cdots M_s\}$ is the moduli set which contains s different moduli that are pairwise co-prime to each other. The representation range of the moduli set is $M = M_1 \cdot M_2 \cdots M_s$. A number X in the range of $[0, M-1]$ can be represented by $\{x_1, x_2, \cdots x_s\}$, where x_i is the residue value of X mod M_i, denoted as $|X|_{M_i}$.

the first residue: n bits	the second residue: n bits	the third residue: $n+1$ bits
$x_1 = \|X\|_{2^n-1}$	$x_2 = \|X\|_{2^n}$	$x_3 = \|X\|_{2^n+1}$

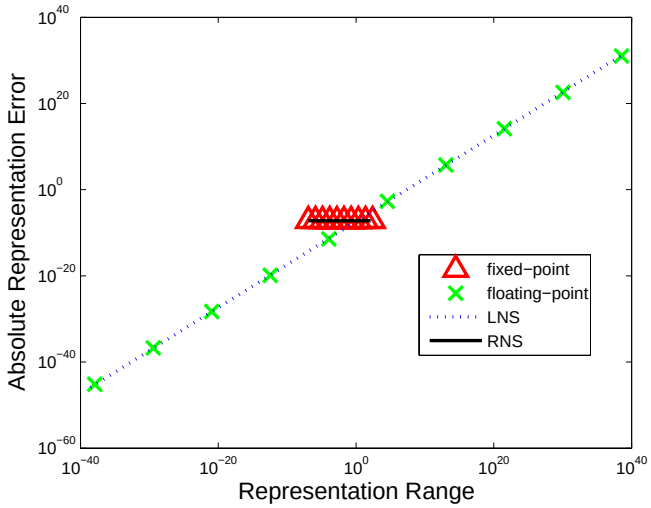
Taking a well known moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ [WSAS02] as an example, the RNS number format is shown as above.

Figure 2.1 shows the representation ranges and errors of all the four different number systems. To perform an unbiased comparison, we specify the same bit widths for different number systems when possible (the bit width of RNS is constrained by the selected moduli values), detailed as follows:

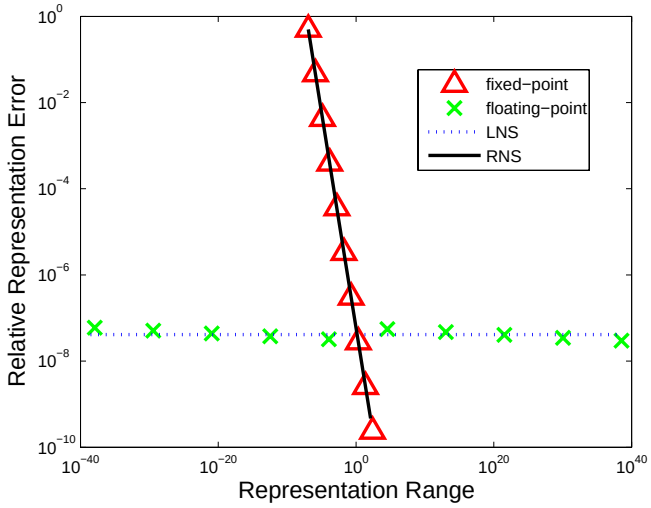
- 32-bit fixed-point: 9-bit integer part, and 23-bit fractional part.
- 32-bit floating-point: IEEE-754 single precision, 1 sign bit, 8-bit exponent, and 23-bit mantissa.
- 32-bit LNS: 1 sign bit, 8-bit integer part, and 23-bit fractional part.
- 31-bit RNS: use the moduli set $\{2^{10} - 1, 2^{10}, 2^{10} + 1\}$, the first and the second residue values are 10-bit, while the third residue value is 11-bit.

As shown in both Figure 2.1(a) and Figure 2.1(b), fixed-point and RNS numbers provide very similar representation ranges and errors to each other, while floating-point and LNS numbers are another similar pair. Compared among the two pairs, floating-point and LNS numbers have a dynamic range which is much larger than the representation range of fixed-point and RNS numbers. The absolute errors of floating-point and LNS numbers increase linearly with the actual value, thus have a wide range and very large maximum values. In contrast, fixed-point and RNS numbers have a small constant absolute error. However, floating-point and LNS numbers provide fairly small relative errors, while fixed-point and RNS numbers have a varying relative error which can be as large as 0.5 (excluding the case that the original value is zero). Compared between floating-point and LNS numbers, LNS numbers have a constant relative error while the relative error of floating-point numbers vary in a small range.

Besides representation formats and errors, the arithmetic operations of the four number systems are also different. Table 2.1 provides a rough qualitative description about the resource costs



(a) Variations of absolute errors over the representation range.



(b) Variations of relative errors over the representation range.

Figure 2.1: Representation ranges and errors of fixed-point, floating-point, LNS and RNS numbers.

and latencies of arithmetic operations of fixed-point, floating-point, LNS, and RNS numbers. In addition, Table 2.2 shows a similar evaluation of the conversion units between different number system. As most existing applications use fixed-point or floating-point numbers, the conversion units we discuss here are between LNS and floating-point, or RNS and fixed-point. A more detailed discussion about LNS and RNS arithmetic and their conversions to and from fixed-point and floating-point numbers is given in Section 2.4.1 and Section 2.4.2.

Table 2.1: Rough qualitative estimation of the area cost and latency for arithmetic operations of different number systems.

Number System	ADD		SUB		MUL		DIV	
	cost	latency	cost	latency	cost	latency	cost	latency
fixed-point	low	low	low	low	medium	medium	high	high
floating-point	low	low	low	low	medium	medium	high	high
LNS	high	high	very high	high	low	low	low	low
RNS	low	low	low	low	low	low	very high	very high

Table 2.2: Rough qualitative estimation of the area cost and latency for conversion units between different number systems. FIX and FLP refer to fixed-point and floating-point numbers.

Conversion Units	FLP to LNS	LNS to FLP	FIX to RNS	RNS to FIX
cost	medium	medium	medium	high
latency	medium	medium	medium	high

2.3 Automation of Design Space Exploration

The flexibility in the FPGA's architecture provides a lot of design freedom for the user. Meanwhile, the design options also make it difficult to determine the optimal choice for a given application. As the capacity of FPGA devices keeps growing, the complexity of the designs that can reside on FPGAs makes manual optimisation impractical, and requires automated design space exploration.

The DEFACTO compilation system [SHD02, SDH03] is one of the examples of automated design space exploration systems. The compilation system takes high-level algorithm descriptions written in a subset of C, and automatically converts them into application-specific designs for FPGAs. DEFACTO explores design options that relate to the parallelism of the computation, such as the unroll factors for the loops in a loop nest. Applying different unroll factors, DEFACTO uses parallelising compiler techniques, such as code transformations, to convert the description into different parallel designs. After that, DEFACTO estimates the performance and area cost of the FPGA designs from behavioural synthesis, and picks the best design based on the estimates.

C. Brandolese et al. [BFP+06] provide a detailed discussion about design exploration for heterogeneous multiprocessor system-on-chip (MP-SoC). A MP-SoC generally consists of a number of different components, such as Digital Signal Processors (DSPs), General Purpose

Processors (GPPs), and specific hardware components. For a given application, there can be many potential system architectures. This work takes a system description in SystemC [IEE05], and investigates the different candidates with quantitative measures correlating the type of executor, the functionality, and a timing estimation.

As shown in the above work, a design space exploration tool generally takes a high-level description of the design as input, performs transformations to map the design description onto different computation platforms, and provides measures or estimations of different metrics, so as to evaluate different design options and select the most suitable solution.

The automated design space exploration platform in my work takes a similar approach to tackle the problem. However, while the above design space explorations focus on the design options about execution sequences (how to unroll a loop) or system architectures (hardware/software partitioning), my work explores different number representations of the variables in the design, thus requiring arithmetic unit generation for different number systems and automated bit-width optimisation that determines the bit widths of the variables. The following Sections 2.4 and 2.5 discuss these two topics in more detail.

2.4 Arithmetic Unit Generation

Automated exploration of number representations requires arithmetic unit generation. The generated arithmetic units should be fairly optimised to provide state-of-the-art performance and efficiency. Otherwise, exploration based on the generated units may not reflect the design solution with up-to-date technologies.

The exploration platform in my work uses ASC as the hardware compilation tool. ASC supports automated generation of fixed-point and floating-point arithmetic units, which provide a competitive performance to commercial arithmetic cores. On top of ASC, we develop optimised library generators for LNS and RNS arithmetic units. The following Sections 2.4.1 and 2.4.2 provide background discussion about LNS and RNS arithmetic unit designs.

2.4.1 LNS Arithmetic Design

Brief Introduction

Suppose a and b are the logarithmic forms of positive numbers A and B ($A > B$), thus $A = 2^a$ and $B = 2^b$. The basic LNS arithmetic operations of these two numbers include:

MUL: $A \times B = 2^a \times 2^b = 2^{a+b}$.

DIV: $A \div B = 2^a \div 2^b = 2^{a-b}$.

SQRT: $\sqrt{A} = \sqrt{2^a} = 2^{a/2}$.

ADD: $A + B = 2^a + 2^b = 2^{b+\log_2(1+2^{a-b})}$.

SUB: $A - B = 2^a - 2^b = 2^{b+\log_2(2^{a-b}-1)}$.

Thus, multiplication and division operations, which are complicated for fixed-point and floating-point numbers, become as simple as fixed-point addition and subtractions. Exponential operations, such as square and square-root, also become as easy as shifts. On the other hand, the addition and subtraction, which are easy for fixed-point and floating-point numbers, become quite difficult.

The central problem for implementing the addition/subtraction of logarithmic numbers is to evaluate the two functions: $f_1(x) = \log_2(1 + 2^x)$ and $f_2(x) = \log_2(2^x - 1)$. By determining which operand is larger before the addition/subtraction, we can assure that $x \geq 0$ in the above two functions. As shown in Figure 2.2, the values of the two functions are easy to approximate when x is quite large. But when the value of x gets close to 0, the functions become highly nonlinear. Moreover, the function $f_2(x) = \log_2(2^x - 1)$ and its derivatives have singularities at zero, which make the evaluation even more difficult.

Previous Work on LNS Arithmetic

There have been numerous research efforts that try to improve the performance of LNS addition/subtraction hardware implementation with various approaches.

(1) Direct Look-up Table: When the bit width of the LNS number is small, implementation with direct look-up tables is a reasonable choice. The FFT LNS implementation in [SCNS83]

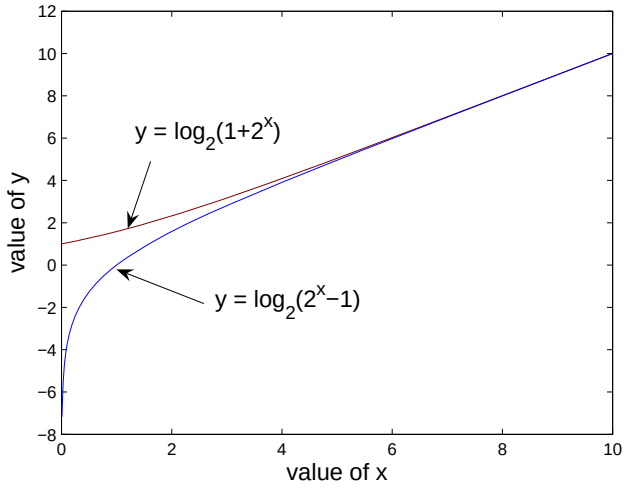


Figure 2.2: Values of functions $f_1(x) = \log_2(1 + 2^x)$ and $f_2(x) = \log_2(2^x - 1)$.

uses two read-only memories (ROMs) to store the values of the two key functions separately. For a small bit width of 17, it gives a smaller error than fixed-point and floating-point numbers.

(2) Polynomial Approximation: For large bit width situations, direct look-up tables become impractical. Considering a LNS number of 32 bits, a direct look-up table needs $2^{32} \times 32 = 128$ Gbits. Thus, polynomial approximation methods are proposed to handle large bit width cases. In [Lew93], Lewis uses the Lagrange interpolation to compute the values of functions f_1 and f_2. Different from other polynomial approximation methods, the coefficients are calculated on the fly, based on an interleaved memory, which can greatly reduce the storage cost. The work in [OPS95] uses the minimax algorithm to calculate the coefficients and store them in the memory for the later polynomial approximation. The work also proposes a mechanism to perform multi-operand addition/subtraction operations. M. Arnold proposes a single-multiplier quadratic interpolator in [AW01b]. ROMs are used to store the log and anti-log results for some special values, thus the interpolation can be performed with one multiplication and some log and anti-log operations. In his another work in [Arn01], M. Arnold proposes a faithful interpolator with a multiple-of-four partitioning. The memory cost is reduced to one-third with similar accuracy. In [LB03], the authors calculate the functions with a degree-two polynomial interpolation using Chebyshev approximation, and provide a Better-Than-Floating-Point (BTFP) accuracy.

(3) Iterative Methods: Other than polynomial approximation methods, there are also algorithms that handle the logarithmic arithmetic functions with iterative methods, e.g. calculating the result digit by digit. M. Arnold proposes a Dual Redundant Logarithmic Number System in [ABCC90], which calculates the addition/subtraction with an on-line algorithm and it has a much smaller size requirement for memories than the polynomial approximation methods. The LNS chip design in [HC94] evaluates f_1 and f_2 with log and anti-log operations. The log operation is performed with the Digit-Partition technique, which divides the bits into two halves to handle. An Interactive Difference by Linear Approximation (IDLA) method is also proposed to implement the anti-log function. In [CCY00], a pipelined iterative method is used to calculate very large world-length (64-bit) LNS addition and subtraction. The calculation is also based on log and anti-log operations. The digit-parallel additive-normalisation method is proposed to compute the anti-log function. The digit online multiplicative-normalisation method is proposed to compute the log function. Because it uses iterative algorithms, the look-up tables are extremely small but the propagation delay is quite long. The work is improved in [CC03] with a base-e exponential function implementation. Using this transformation, half of the pipeline stages can be replaced by one stage of multiply-and-accumulate (MAC) operation.

(4) Function Decomposition/Transformation: Due to the singularity near zero, the function f_2 for LNS subtraction is even more difficult to evaluate than the function f_1 for addition. There are also approaches specially proposed for the subtraction. In the subtraction algorithm in [PS96], the function $f_2 = \log_2(2^x - 1)$ is transformed into $f_2 = \log_2(x) + \log_2(\frac{2^x-1}{x})$. Calculation of both $\log_2(x)$ and $\log_2(\frac{2^x-1}{x})$ are easier than the original function. In [Arn03], subtraction is performed through additions by either reverse lookup of the addition table (as $\log_2(1 + 2^x)$ and $\log_2(2^x - 1)$ are inverse functions to each other), or using an iterative method based on the equation: $\log_2(1-2^x) = -\sum_{m=0}^{\infty} \log_2(1+2^{x \cdot 2^m})$. Co-transformation methods are also proposed in [Arn02] to avoid the difficulties of logarithmic subtractions near the singularity at zero.

(5) Error Correction: For practical implementations, an LNS microprocessor is developed with the arithmetic implementations described in [CCSK00]. In order to achieve the same latency as its floating-point counterpart, the LNS processor implements the addition/subtraction function with a linear Taylor interpolation, so that only one multiplication and one addition is needed

(in most of the other polynomial approximation approaches, quadratic interpolation is used for a better accuracy). Meanwhile, in order to achieve the same accuracy as the floating-point numbers, a table-based error correction approach is used. The correction scheme is based on two observations: firstly, the error distributions in different segments are roughly the same, thus, only one error value at a specific point is stored for each different segment and all the different segments use the same distribution table, which records the ratios between the error values at other points and the stored error value. Secondly, a large part of the correction values for addition and subtraction functions are the same, thus can be shared between each other. Based on these two empirical facts, the correction scheme can use a small-size table.

In summary, existing approaches either approximate the function values with polynomial interpolations or calculate the values in an iterative method, digit by digit. For both kinds of approaches, other techniques, such as error correction [CCSK00], coefficients generation on the fly [Lew93], and transformations of the original functions [PS96], [Arn02], can be combined to improve the accuracy or reduce the memory requirement.

There are also research studies about hybrid or variations of LNS. M. Arnold et al. [ABCC90] propose a redundant logarithmic number system, in which a number is expressed as two separate LNS numbers. Arnold also presents a hybrid residue and logarithmic number system [Arn05]. A multidimensional logarithmic number system is introduced in [MDJM05]. Most of these new number systems combine LNS and other number formats to mitigate the difficulty in implementing LNS ADD and SUB functions. However, more complexity is brought in at the same time. Paliouras [Pal02] investigates the optimal choice for the logarithmic base of LNS rather than two. This work brings interesting theoretical insights into the logarithmic number format, but a base different from two is practically difficult to implement on hardware because of the need for compatibility with base-two floating point.

Comparisons between LNS and Floating-point

Based on the work of the LNS microprocessor in [CCSK00], a comparison between LNS and floating-point numbers is presented in [MTP+02].

In [DD07], J. Detrey and F. de Dinechin provide a tool for unbiased comparisons between LNS and floating-point arithmetic. Based on their VHDL library for LNS operators in the previous work [DdD03], they provide two libraries of parameterised arithmetic operations, which are convenient tools to set up an application in two different number systems concurrently so as to compare their performance and accuracy. The authors have also given some comparison examples such as a 3D transformation pipeline. However, as mentioned by the authors, the example is not a practical application on FPGA and there lacks a thorough analysis about different number systems' strengths and drawbacks.

The most recent work is the comparison between floating-point and LNS for FPGAs in [HBUH05]. It compares all the basic operators in both 32-bit and 64-bit bit widths, and gives an analysis about on what kind of condition (the range of the ratio between multiplications/divisions and additions/subtractions) the LNS would provide a smaller area or a shorter latency than floating-point numbers. However, there is no comparison or analysis about a practical FPGA application.

In general, most of the above comparison works either only study bit-width values below 32, or focus on the counterparts of IEEE 754 single and double precision FLP formats; they lack a systematic analysis and reconfigurable support for a wide range of LNS bit widths.

2.4.2 RNS Arithmetic Design

Notations and A Brief Introduction

In this thesis, we use the following notations for RNS numbers:

$MS = \{M_1, M_2, \cdots M_n\}$ is the moduli set which contains n different moduli that are pairwise co-prime to each other. The representation range of the moduli set is $M = M_1 \cdot M_2 \cdots M_n$.

A number X in the range of $[0, M-1]$ can be represented by $\{x_1, x_2, \cdots x_n\}$, where x_i is the residue value of X mod M_i, denoted as $|X|_{M_i}$. For the cases that we need to represent negative values, the range becomes $[-M/2, M/2-1]$.

For the convenience of discussions, we also define S_i as $S_i = \prod_{j \neq i}^{n} M_j = M/M_i$, and the value $|x^{-1}|_M$ as the multiplicative inverse of $|x|_M$ that satisfies $\big||x^{-1}|_M \cdot x\big|_M = 1$.

The advantages of RNS in addition and multiplication come from the following property:

$$|a + b|_M = \big| |a|_M + |b|_M \big|_M \tag{2.1}$$

$$|a * b|_M = \big| |a|_M * |b|_M \big|_M \tag{2.2}$$

Thus, to add or multiply two RNS numbers $\{x_1, x_2, x_3\}$ and $\{y_1, y_2, y_3\}$, we only need to add or multiply the corresponding value pairs x_i and y_i. This significantly reduce the length of the carry chain and the latency of the arithmetic operation.

Forward converter

As most existing devices and applications use binary representations, such as fixed-point or floating-point numbers, the first part of an RNS design is usually a converter that converts binary numbers into the residue form, usually denoted as a forward converter or a residue generator.

The early efforts [AM84, CG88] on forward converters decompose the binary value into an array of power-of-two values, compute the residue of each power-of-two value, and sum them up with modular adders. The basic idea is as follows: assume $r_{ji} = |2^j|_{M_i}$, where $0 \leq j \leq n-1$, and $X = \sum_{i=0}^{n-1} b_i \cdot 2^i$, the residue x_i can then be processed as $x_i = |\sum_{j:b_j=1} r_{ji}|_{M_i}$.

Based on the above approach, S. Piestrak [Pie94] proposes the concept of periodic properties of modular operations in his converter designs. Based on the periodic property, the input bits can be divided into a number of groups and handled similarly, which greatly reduces the complexity of the design.

In the most recent work, A. Premkumar [Pre02] formulates the forward conversion with modular exponentiation and addition operations, and provides a formal framework for memory-free

$$\begin{aligned} X &= \{x_1, x_2, \cdots, x_n\} = |\{x_1, 0, 0, \cdots, 0\} + \{0, x_2, 0, \cdots, 0\} + \cdots + \{0, 0, \cdots, x_n\}|_M \\ &= \left|x_1 \cdot |S_1^{-1}|_{M_1} \cdot S_1 + x_2 \cdot |S_2^{-1}|_{M_2} \cdot S_2 + \cdots + x_n \cdot |S_n^{-1}|_{M_n} \cdot S_n\right|_M = \left|\sum_{i=1}^{n} (|S_i^{-1}|_{M_i} \cdot S_i \cdot x_i)\right|_M \end{aligned} \quad (2.3)$$

$$\begin{aligned} X &= |x_1 + k_1 M_1 (x_2 - x_1) + k_2 M_1 M_2 (x_3 - x_2) + \cdots + k_{n-1} M_1 M_2 \cdots M_{n-1} (x_n - x_{n-1})|_{M_1 M_2 \cdots M_{n-1} M_n} \\ \text{where } & |k_1 M_1|_{M_2 \cdots M_n} = 1, |k_2 M_1 M_2|_{M_3 \cdots M_n} = 1, \cdots, |k_{n-1} M_1 \cdots M_{n-1}|_{M_n} = 1 \end{aligned} \quad (2.4)$$

conversions. A. Premkumar et al. extend this work in [PAL06], using the periodicity property of residues to further reduce the cost and improve the speed.

Reverse converter

Compared to the forward converter, the reverse converter (converting RNS numbers back into binary form, also referred to as RNS decoder) is more complicated and costs more resources to implement.

The earliest algorithm of reverse conversion dates back to the Chinese Remainder Theorem (CRT), which is proposed by an ancient Chinese mathematician Tzu Sun [SunAD, LA04]. The basic idea of CRT can be illustrated as equation (2.3). Multiplicative inverse values are used to reconstruct the binary value from residue values.

There are recent designs [AM84, CG88] that apply the same idea of CRT to perform the reverse conversion with efforts to improve the performance with different hardware architectures.

Y. Wang proposes another reverse conversion algorithm based on his New Chinese Remainder Theorems [Wan98]. This algorithm calculates the binary value as shown in equation (2.4). Similar ideas are proposed for conversion from RNS numbers to Mixed-Radix (MR) representations [Hua83]. Applying this algorithm to the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, Y. Wang provides an adder-based reverse converter [WSAS02], which is one of the most efficient reverse converter designs for this moduli set.

Scaling

In DSP applications, the RNS scaling operation is important for keeping the magnitude of the

signals within the representation range and preventing overflows. Assume X is the original value, K is the scaling factor, and Y is the scaled value, i.e. $X = K \cdot Y + |X|_K$. The scaling operation is usually performed as $Y = \frac{X - |X|_K}{K}$.

Most of the early scaling algorithms are based on the CRT, and use lookup tables to perform complicated operations such as multiplications, division and modular[GST89, SK89]. Recent approaches [DMST08] perform the computations in the residue representation rather than doing a CRT reverse conversion first. Assume the residue form of the scaling result Y is $\{y_1, y_2, \cdots, y_n\}$, then y_i can calculated as $y_i = \left| \left| x_i - ||X|_K|_{M_i} \right|_{M_i} \cdot |K^{-1}|_{M_i} \right|_{M_i}$.

Power-of-two values are one kind of commonly-used scaling factors [MBS03]. The power-of-two scaling is usually done in an iterative manner, i.e. scale the value by two each time, and iterate it n times for 2^n scaling. In each scaling-by-two step, depending on whether X is even or odd, y_i is calculated as $\left| |x_i|_{M_i} \cdot |K^{-1}|_{M_i} \right|_{M_i}$ (for even values) or $\left| |x_i + 1|_{M_i} \cdot |K^{-1}|_{M_i} \right|_{M_i}$ (for odd values).

Magnitude Comparison

The most straightforward way of magnitude comparison is to reverse-convert the residue representation into binary representation or Mixed-Radix (MR) representations, and perform the comparison.

There are also other methods to derive a monotonic growing function from the residue values. Based on the observation of a "diagonal function" in the sequence of RNS numbers, G. Dimauro et al. [DIP93] propose a function that uses residue values and an extra modulus to compute a value that grows monotonically with the magnitude of RNS numbers. However, the computation of the "diagonal" value still involves costly multiplication and modular operations.

2.5 Bit-width Optimisation

Automation of the design space exploration requires arithmetic generation. When we generate the arithmetic units in an application, we need to specify how many bits to use for each variable.

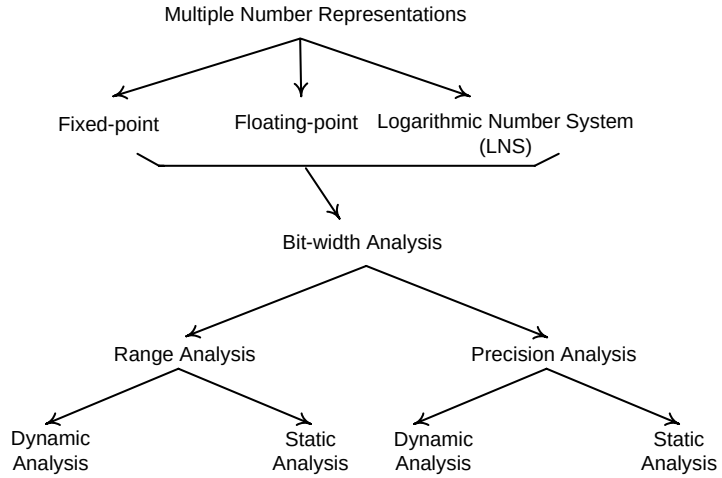


Figure 2.3: Bit-width analysis (precision and range) for multiple number representations.

Bit-width optimisation is a technique which automatically deduces the optimal bit widths for the variables in a design to minimise a cost metric, while satisfying the range and precision requirements. We use the classification in Figure 2.3 for bit-width optimisation. First, there are multiple choices for the number representation of the circuit design, and the bit-width optimisation tool in this work handles three major formats used on reconfigurable devices: fixed-point, floating-point and LNS. Second, for each number format, we adopt a set of different analysis techniques.

In this classification, *range analysis* considers the problem of ensuring that a given variable inside a design has a sufficient number of bits to represent the range of its values. In *precision analysis* the objective is to find the minimum number of precision bits for the variables in the design such that the output precision requirements of the design are met.

Dynamic analysis involves the simulation of the design with actual data to perform the analysis. *Static analysis* operates on the computational flow of the design and uses techniques based on range and error propagation to perform the analysis. In general, dynamic analysis provides more optimistic results than static analysis, however these results are dependent on the data sets used for the simulation.

There exist a number of research projects that focus on bit-width optimisation. This section considers key approaches in the field, with the aim of comparing between them in terms of their

strengths and weaknesses. When considering previous work we follow the classification scheme shown in Figure 2.3.

In [BP00] a static analysis technique based on interval arithmetic [Moo66] is proposed. However, when applied to range analysis, this method suffers from the loss of correlation between variables, which leads to overestimation in the ranges.

Another static analysis scheme [NHCB01] is proposed as part of the MATCH [HNS+01] compilation system. The MATCH compilation system targets high-level Matlab [Mat05] descriptions and converts them into optimised FPGA descriptions. The aim of this compilation system is to "dramatically reduce" the compilation time from high-level algorithm to low-level hardware. In this scheme, the authors divide the problem of bit-width optimisation into two major parts: finding the integer bit width which corresponds to the range, and finding the fractional bit width which corresponds to the precision. The algorithm for finding the integer bit-width relies on a data-range propagation method [BP00].

The FRIDGE project [WBGM97] provides a set of tools for hardware software co-design, where high-level floating-point designs are converted to fixed-point hardware descriptions. In [WBK+97] a bit-width optimisation scheme is proposed. This method requires the designer to specify the bit widths for some of the fixed-point variables. The bit widths of the remaining fixed-point variables are then found through simulation.

In [WP98], the authors consider the sensitivity of the output to errors in the intermediate variables for fractional bit-width optimisation. They use the Taylor series to propagate the worst case truncation error at each node in the dataflow graph to the output.

A bit-width minimisation scheme targeting Linear Time Invariant (LTI) is proposed in [CCL04]. LTI systems are commonly used in digital signal processing systems, such as digital signal filters. The proposed scheme is analytical and independent of the input data set. As part of the Synoptix [CCL04] hardware synthesis system, the scheme converts high-level algorithm descriptions to register-transfer level (RTL) descriptions. Key features of this work include the support for multiple word-lengths and a novel resource binding technique. To overcome the

problem of NP-hardness [CW02] associated with multiple-word-length optimisation, a heuristic based bit-width allocation approach is proposed. This work is subsequently extended to include sensitivity analysis for differentiable non-linear systems in fixed-point calculations [Con06].

Shi and Brodersen [SB04] propose a statistical modelling based method for bit-width optimisation. This method is targeted at VLSI-based communication systems and integrated with the Xilinx System Generator [Xil04] design system.

Fang et al. [FRC03] propose a method which tries to overcome the correlation problem [BP00] of interval arithmetic [Moo66]. The solution makes use of a refinement of interval arithmetic known as affine arithmetic [ACS94]. In recent works, other techniques based on probabilistic models [SFMR06] are applied to the original affine arithmetic to achieve probabilistic intervals to achieve accurate and tight interval bounds.

2.6 Summary

This chapter provided background knowledge about achieving application-specific number representations on FPGAs. Section 2.2 introduced the general architecture of FPGA, design tools, accelerations of applications on FPGA, and four different number representations, fixed-point, floating-point, LNS, and RNS. Section 2.3 discussed about automation of design space exploration. The automated design exploration requires arithmetic unit generation and bit-width optimisation, which were discussed in Section 2.4 and Section 2.5.

Compared with previous work on arithmetic generation, the LNS arithmetic unit generation in our platform (detailed in Chapter 4) supports a larger set of bit-width settings, and provides systematic optimisation of various parameters. Compared to previous LNS units in [HBUH05], our generated LNS units provide in most cases 6% to 37% reduction in area and 20% to 50% reduction in latency. Our RNS arithmetic unit generation (detailed in Chapter 5) provides a novel design of the reverse converter using only n-bit adders, and reduces the area cost by 14.3% when compared to the previous design in [WSAS02].

Compared with previous work on bit-width optimisation, R-Tool in our platform (detailed in Chapter 6) improves on it by supporting both dynamic and static methods to provide optimised bit widths for different guarantees of accuracy, and supporting multiple number representations, such as fixed-point, floating-point, and LNS numbers.

Part I

An Automated Exploration Platform

Chapter 3

Automated Design Space Exploration for Number Representations

3.1 Introduction

Different number representations have significant cost and performance differences on certain arithmetic operations. As mentioned in Chapter 2, LNS multiplication and division consume much less resources than their counterparts in the other number formats, while LNS addition and subtraction consume much more than the other number formats. RNS addition and multiplication provide a smaller latency than other number formats, while comparison and division operators become extremely difficult.

However, given all the known characteristics of different number representation systems, there are still no straightforward methods to figure out the most appropriate number format for a given application, especially for large scale applications involving numerous different computations. To solve the above problem, we develop an automated exploration platform that can explore different number representations for a given application. In this chapter, we will provide an overview of our exploration platform.

The platform currently supports four different number systems: fixed point, floating point,

LNS, and RNS. For the four different number systems, optimised arithmetic units can be automatically generated for a wide range of bit widths. For the fixed point, floating point, and LNS, the platform also provide a bit-width optimisation tool, which can minimise the bit widths of the variables under the constraint of a given error requirement.

While supporting a wide range of different number formats, there are also some limitations about the designs that the platform can handle. Firstly, the platform targets feed-forward streaming designs, and does not work quite well with circuits containing complicated controls or feedbacks. Secondly, using multiple number systems in one design is not considered in our current work. We assume that all the variables in the design are using the same number system, although they can have different bit widths.

Figure 3.1 shows the general workflow of our automated exploration platform for different number representations:

1. The user feeds a high-level description of the target application into the exploration platform. The description is in a C++ syntax with hardware data-types defined in ASC.

2. With the arithmetic support for fixed-point, floating-point, LNS and RNS numbers, the platform automatically converts the description into various designs using different number systems (Note that our platform targets feed-forward designs without complicated controls, so the major task is to map arithmetic operations into hardware design).

3. For each of the generated designs, the platform performs an automated bit-width optimisation based on a specified accuracy constraint. The bit-width optimisation determines the bit widths of the variables in the design.

4. After the number representations for all the variables are determined, the platform evaluates all the generated designs for metrics, such as area, latency, throughput, and accuracy.

5. With all the results regarding different design options, we can then study the different characteristics of different number systems, and figure out the tradeoffs between different design parameters.

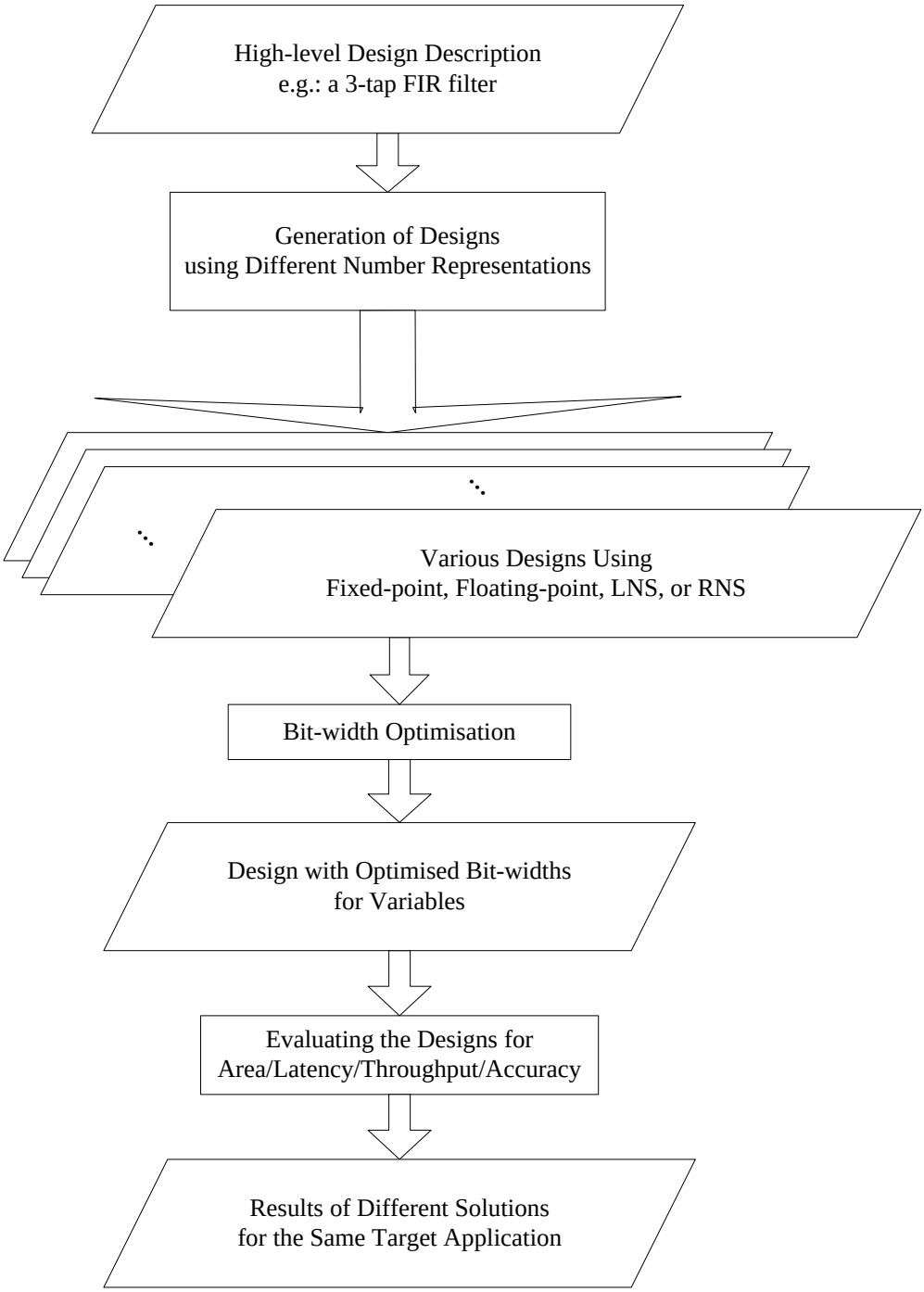


Figure 3.1: General workflow of our automated exploration platform for different number representations.

Corresponding to the above five steps, our automated exploration platform contains the following four key elements:

- firstly, arithmetic unit generation for different number systems, so that the high-level description of the design can be automatically converted into implementations using different number representations;
- secondly, bit-width optimisation, which minimises the bit widths of all the variables under a given accuracy constraint;
- thirdly, evaluation methods that estimate or measure different metrics of the hardware design, such as area, latency, throughput, and accuracy;
- lastly, an automatic design exploration mechanism, so that a large number of different designs can be tested, and the results can be collected automatically.

The following sections discuss the four key elements respectively.

3.2 Arithmetic Support

In the second step of the general workflow, arithmetic unit generation is required to map the arithmetic operations (+, -, *, /, comparator, etc.) into arithmetic units of different number systems. The generated arithmetic units of different number systems should be fairly optimised to provide state-of-the-art performance and efficiency. Otherwise, the exploration results based on the generated units may not reflect design solutions with the most recent technologies.

As noted in Chapter 1, our hardware design tool, ASC, already provides reconfigurable arithmetic units for fixed-point and floating-point numbers. Figure 3.2 shows a simple example of ASC code. In the example, the user defines a hardware fixed-point number with a construction function, `HWfix(arch_type, tbw, fbw)`. In the function, 'arch_type' indicates the architecture type of the variables, which can be input (`IN`), output (`OUT`), temporary variables (`TMP`) or registers (`REGISTER`); '`tbw`' indicates the total bit width of the variable, while '`fbw`' indicates the fractional bit width of the variable. Using these parameters, the user can easily specify the types and bit widths of the variables. Moreover, after declaration of these variables, the

```
#include "asc.h"

int main(int argc, char **argv){

    STREAM_START;

    HWfix a(IN, 32, 23);
    HWfix b(TMP, 32, 23);
    HWfix c(OUT, 33, 24);

    STREAM_LOOP(10);

    b = a+1.5;
    c = b*488;

    STREAM_END;
}
```

Figure 3.2: Sample code of ASC.

user can then describe the operations just as programming software code. This provides an easy-to-use interface to fast prototype a design on FPGA platforms.

To provide a similar reconfigurable support for LNS and RNS arithmetic, we develop optimised library generators for LNS and RNS arithmetic units.

For LNS arithmetic unit generation, we first introduce a general polynomial approximation approach for LNS arithmetic function evaluation. Second, we develop a library generator that produces LNS arithmetic libraries containing $+, -, *, /$ operators as well as converters between floating-point and LNS numbers. The generated libraries cover a wide range of bit widths and provide a systematic optimisation of LNS arithmetic. When compared with existing LNS designs [HBUH05], our LNS units typically achieve 6% to 37% reduction in area (number of slices) and 20% to 50% reduction in latency, with a reduced or comparable usage of block RAM (BRAM) and 18-by-18 hardware multipliers (HMUL). The LNS library generator enables the users to fast prototype LNS FPGA applications, and compare between LNS and FLP implementations on area, performance and accuracy. Thus, the users can efficiently study the potential benefits of using LNS number representation on various applications.

My work also provides improved designs for RNS arithmetic unit generation. For the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, we propose a novel design for reverse converters, which uses only n-bit additions and consumes 14.3% less area than previous work [WSAS02]. We also develop an RNS arithmetic library generator for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. The generator supports a customisable parameter n from 4 to 20, and enables us to perform an extensive comparison between RNS and other number representations at both the arithmetic unit level and the application level.

A more detailed description about LNS and RNS arithmetic library generators is given in Chapter 4 and Chapter 5.

Our current arithmetic unit generation for different number systems is based on ASC. However, the proposed techniques and architectures can also be applied to other hardware description languages or compilers. One of the future plans is to extend the current arithmetic library generators so that they produce arithmetic units described in VHDL as well as in ASC.

3.3 Bit-width Optimisation

In the third step of the exploration workflow, bit-width optimisation is performed to automatically minimise the bit widths of all the variables in the design, under a given accuracy constraint. To accomplish this task, we develop R-Tool, which can optimise variable bit widths for different number representation systems. When doing the bit-width optimisation, the constraint is an error requirement specified by the user, and the object function is one of the metrics that we want to minimise or maximise. In most cases, the object function is the area cost of the resulting design.

R-Tool extends a previous work "BitSize" [GMLC04]. The tool is extended to support LNS designs as well as fixed-point and floating-point designs. Meanwhile, the error and area-cost models for different number representations are refined to provide more accurate error and area analysis for different bit-width settings, thus helping to achieve a better optimisation result.

R-Tool combines a dynamic automatic differentiation approach and a static affine arithmetic method to provide optimised bit widths with user-desired guarantees of accuracy. The static method tackles the problem conservatively to assure the required accuracy over all possible inputs, while the dynamic approach reduces resource costs by only guaranteeing accuracy for tested input values.

Chapter 6 provides a more detailed discussion about R-Tool.

3.4 Evaluation of the Metrics

In the fourth step of the design exploration, the exploration platform evaluates the generated designs for different metrics, such as area, latency, throughput, and accuracy.

A straightforward approach to measure these metrics is to synthesise the generated designs, map them onto an FPGA, use the vendor's tool to place and route the circuits, and collect the area and timing results from the reports. This approach provides precise experimental results for these metrics. However, as a price of being precise, the time taken to map, place, and route a circuit design is generally very long. Depending on the complexity of the circuit, the entire process takes hours to days. Therefore, exploration of a design with numerous options becomes extremely time-consuming.

To reduce the time for evaluating the area cost of hardware designs, the exploration platform provides a two-level area cost estimation tool. The first level is an area model based on table lookup or interpolation of recorded area results, and produces results in less than a second, with an average error around 5%. The area model serves as an extension to the ASC compiler. The tool derives a data flow graph of the design from the ASC description, and calculates the area cost of the arithmetic units according to their bit widths. The calculation of the area cost is a table lookup or interpolation of recorded area results. The second level maps, places and routes the actual design onto FPGAs to acquire experiment results which are more accurate.

To evaluate accuracy, we develop a bit-accurate simulator based on truncation or rounding

of floating-point calculations. To achieve high efficiency of the value simulation, we use the long double data type in C++. The long double data type is implemented as 80-bit floating-point format (63-bit mantissa, 16-bit exponent, and 1-bit sign) on x86 machines. By doing bit manipulation of the mantissa and exponent values, we are able to provide bit-accurate simulation of variables up to 64 bits.

Some more detailed discussions about the area estimation tool and bit-accurate simulator are presented in Chapters 4 and 6.

3.5 Automatic Design Exploration

ASC provides a mechanism to perform design explorations automatically. As shown in Figure 3.3, in the source file 'stream.cxx' which describes the design of the circuit, we declare the hardware data-types and the bit-width values with C++ macros `DATATYPE`, `TWIDTH` and `FWIDTH`. If we also want to explore the different optimisation modes of ASC, we can declare the optimisation mode with another macro `OPT`. Then, in the makefile of ASC, we use flag '`--ALT`', which means getting results for area, latency and throughput, and use `-D` to assign the different values we want to experiment to the macros. With the design description file and the configuration file, we can simply type '`make run0`' to explore all the possible combinations of different design options, and collect the corresponding area, latency and throughput results.

With the results of all the different designs, the user can then decide which data-type, bit-width values and optimisation mode fit the application the best.

3.6 Summary

This chapter gave an overview of our automated design space exploration tool for different number representations, and described the related key elements. To accomplish the entire workflow of design exploration, our exploration platform provides arithmetic support for different number systems, which was discussed in Section 3.2. Our exploration platform also includes

```
    design file: stream.cxx
```

```
#include "asc.h"

int main(int argc, char **argv)
{
    STREAM_START;
    STREAM_OPTIMIZE=OPT;

    DATATYPE a(IN, TWIDTH, FWIDTH);
    DATATYPE b(TMP, TWIDTH, FWIDTH);
    DATATYPE c(OUT, TWIDTH, FWIDTH);

    STREAM_LOOP(10);

    b = a+1.5;
    c = b*488;

    STREAM_END;
}
```

```
    configuration file: makefile
```

```
RUN0 = --ALT -DDATATYPE="HWfix;HWfloat;HWlns"
             -DOPT="AREA;LATENCY;THROUGHPUT"
             -DTWIDTH="30;31;32"
             -DFWIDTH="17;18;19"
```

Figure 3.3: Sample code of automatic design exploration.

the bit-width optimisation process to automatically determine the bit widths of the variables, which was presented in Section 3.3. Section 3.4 presented the basic techniques to evaluate the different metrics in our exploration platform. Section 3.5 showed the basic mechanism of automated design exploration in ASC.

The following Chapters 4, 5, and 6 provide some further discussions about arithmetic support and bit-width optimisation.

Chapter 4

Optimised Logarithmic Arithmetic Unit Generation

4.1 Introduction

The Logarithmic Number System (LNS) was first introduced into computer systems for processing of low-precision FFT in the 1970s [SCNS83]. Unlike the floating-point (FLP) numbers defined by the IEEE 754 standard, there is no commonly accepted standard for LNS numbers. As mentioned in Section 2.2.4, this work uses the signed-logarithmic representation format similar to [ABCC90], which consists of a sign bit and a fixed-point number to record the logarithmic value, shown as follows:

Sign Bit	Fixed-point Logarithmic Value	
	Integer: m bits	Fractional: f bits
S	M	F

Its value is given by $(-1)^S \times 2^{M.F}$, which provides a similar representation range to FLP numbers with m-bit exponent, f-bit mantissa and one sign bit. Similar to the encoding of infinity and 'Not a Number' (NaN) cases in FLP, we use special encoding of the integer bits to indicate zero and exceptional cases.

As shown in Section 2.4.1, we can implement LNS MUL and DIV with fixed-point addition or subtraction, and acquire the SQRT result by a one-bit shifting. However, for LNS ADD and SUB operations, we need to evaluate two transcendental functions, $f_1(x) = \log_2(2^x + 1)$ and $f_2(x) = \log_2(2^x - 1)$. The LNS SUB function f_2 and its derivatives have singularities at zero, which make the evaluation very difficult.

For applications that compute over a wide dynamic range, LNS provides an alternative solution to FLP, and the possibility to implement the design with a smaller area or higher throughput. Based on our previous work on optimising LNS arithmetic on FPGAs [FML07], we develop tools to provide automated generation of optimised LNS arithmetic units, enable convenient design and implementation of LNS application, and facilitate evaluation of area, precision and throughput metrics of the designs. The major contributions of this chapter are:

- A systematic optimisation of LNS arithmetic on FPGAs, using a general polynomial approximation approach. The approach provides an adaptive divide-in-halves segmentation method, supports polynomial degrees from one to five and reconfigurable ratio between approximation and quantisation errors.

- A library generator that produces LNS arithmetic libraries containing $+, -, *, /$ operators as well as converters between floating-point and LNS numbers. The generated libraries provide a simple-to-use address encoding mechanism for coefficient tables, on-chip block memory sharing to save coefficient storage, and bit-width minimisation based on affine arithmetic.

- Tools to facilitate comparison between LNS and floating-point designs. We develop a two-level area cost estimation tool: the first level uses area models to acquire results in less than a second, with an average error around 5%; the second level performs an automated mapping of different designs onto FPGAs to acquire the exact experimental results. To evaluate accuracy, we also develop a bit-accurate simulator based on truncation or rounding of floating-point calculations.

The basic arithmetic units are tested on practical FPGA board as well as software simulation. When compared with existing LNS designs [HBUH05], our LNS units typically achieve 6% to 37% reduction in area (number of slices) and 20% to 50% reduction in latency, with a reduced or comparable usage of block RAM (BRAM) and 18-by-18 hardware multipliers (HMUL). The tool infrastructure enables the users to fast prototype LNS FPGA applications, and compare between LNS and FLP implementations on area, performance and accuracy. Thus, the users can efficiently study the potential benefits of using LNS number representation on various applications.

The rest of the chapter is organised as follows. Section 4.2 presents our evaluation method for the LNS arithmetic functions. Section 4.3 shows the details of our LNS arithmetic library. Section 4.4 compares between LNS and FLP arithmetic units, presents our two-level area cost estimation tool and the bit-accurate simulator. Section 4.5 summarises this chapter.

4.2 Evaluation of LNS Arithmetic Functions

Since FLP numbers dominate most of the existing hardware/software applications, a complete LNS arithmetic library also needs conversion functions between LNS and FLP numbers. Thus, we include the logarithmic function $f_3(x) = \log_2(x)$ and the exponential function $f_4(x) = 2^x$ in our target LNS library. For these four functions (f_1 to f_4), our library generator produces LNS arithmetic units for integer bit width m from 4 to 11, fractional bit width f from 13 to 52, and all the possible combinations between them, i.e. over 300 different bit-width settings. The largest bit-width setting ($m = 11, f = 52$) corresponds to double precision floating-point numbers.

4.2.1 Piecewise Polynomial Approximation

This work adopts the piecewise polynomial approximation method as a general approach to evaluating all four functions. The approximation approach divides the entire evaluation range

into a number of segments, and evaluates each segment with a different minimax polynomial, which is generated through the Remez algorithm and provides the minimum maximum approximation error over the range [Mul97].

Many different kinds of polynomials can be used for the approximation of elementary functions. The most famous ones include the Taylor polynomial derived from Taylor Series, the interpolating polynomial in the Lagrange form, and Chebyshev approximation [Mul97].

The Taylor polynomial calculates the coefficients based on the derivatives of the function to be approximated. The equation for approximating function $f(x)$ on the range of $[a, b]$ using a degree-n Taylor polynomial is shown as follows:

$$f(x) \approx P_{Taylor}(x) = f(a) + f'(a) \cdot (x - a) + \cdots + f^{(n)}(a) \cdot (x - a)^n \tag{4.1}$$

The Lagrange polynomial is the interpolating polynomial based on a given set of data points in the range to be approximated. Assume that (x_0, y_0), (x_1, y_1), $\cdots$, (x_n, y_n) are the $n+1$ data points we choose from the range of $[a, b]$ (a and b are usually selected to be x_0 and x_n), where $y_i = f(x_i)$. The degree-n Lagrange polynomial to approximate $f(x)$ over the range of $[a, b]$ is shown as follows:

$$f(x) \approx P_{Lagrange}(x) = \sum_{k=0}^{n} y_k \cdot \frac{(x - x_0) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)} \tag{4.2}$$

The Chebyshev approximation is based on Chebyshev polynomials, which are a sequence of orthogonal polynomials. The Chebyshev polynomials can be defined in a recursive way. E.g., Chebyshev polynomials of the first kind (there are two different kinds of Chebyshev polynomials) can be defined as follows:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \end{aligned}$$

Based on this sequence of polynomials, function $f(x)$ over the range of $[a, b]$ can be approximated as follows:

$$f(x) \approx P_{Chebyshev}(x) = \sum_{k=0}^{n} c_k \cdot T_k(\frac{2x - (a+b)}{b-a}) \tag{4.3}$$

The coefficient c_k in Equation 4.3 is defined as follows:

$$c_k = \begin{cases} \frac{1}{n+1} \sum_{i=0}^{n} f(x_i) \cdot T_0(\frac{2x_i - (a+b)}{b-a}) & \text{if } k = 0 \\ \\ \frac{2}{n+1} \sum_{i=0}^{n} f(x_i) \cdot T_k(\frac{2x_i - (a+b)}{b-a}) & \text{if } 1 \leq k \leq n \end{cases} \tag{4.4}$$

$$\text{where } x_i = \frac{b-a}{2} \cdot cos(\frac{i-n}{n+1} \cdot \pi) + \frac{a+b}{2},\ 0 \leq i \leq n \tag{4.5}$$

As shown above, the Chebyshev approximation takes a more complicated form than Taylor or Lagrange polynomials, while the approximation polynomial is also closer to the optimal one.

In our work, we use the minimax polynomial, which is the optimal polynomial for the given range and polynomial degree in terms of generating the smallest maximum error. We generate the minimax polynomial using the Remez algorithm. The Remez algorithm that produces a degree-n minimax polynomial for the range of $[a, b]$ is described as follows:

1. Take the $n+1$ Chebyshev points $(x_0, x_1, \cdots, x_n)$ described in Equation 4.5 and points a, b as the initial $n+3$ points, and use the Chebyshev approximation described in Equation

4.3 as the initial approximation polynomial. With the $n+3$ initial points, we have $n+2$ intervals $([a, x_0], [x_0, x_1], \cdots, [x_n, b])$. Find out $n+2$ extrema points $(e_0, e_1, \cdots, e_{n+1})$ that either produce the minimum or the maximum approximation errors in the $n+2$ intervals (one point in each interval).

2. Use the $n+2$ extrema points to set up a system of equations, where each equation is in the form of $a_0 + a_1 \cdot e_i + a_2 \cdot e_i^2 + \cdots + a_n \cdot e_i^n + (-1)^i \cdot E = f(e_i),\ 0 \leq i \leq n+1$. Solve the system to get the values of $(a_0, a_1, \cdots, a_n, E)$, and use $(a_0, a_1, \cdots, a_n)$ as the coefficient for the new approximation polynomial.

3. For the $n+1$ intervals in between of the $n+2$ extrema points, find out $n+1$ root points $(r_0, r_1, \cdots, r_n)$ that satisfies $a_0 + a_1 \cdot r_i + a_2 \cdot r_i^2 + \cdots + a_n \cdot r_i^n - f(r_i) = 0$, i.e. having an approximation error of zero.

4. Take the $n+1$ root points and points a, b as the updated $n+3$ points, and perform the same procedures as in step 1 to find out the updated $n+2$ extrema points $(e_0, e_1, \cdots, e_{n+1})$.

5. If the updated extrema points are the same as the extrema points in the previous iteration, we can terminate the process, and output $(a_0, a_1, \cdots, a_n)$ as the minimax polynomial coefficients and E as the maximum approximation error over the range $[a, b]$. If they are the same, jump back to step 2 and continue the iteration.

As shown above, the Remez algorithm does not only provide the minimax polynomial coefficients, but also give the maximum approximation error, which can be quite useful for hardware designs that requests certain level of accuracy. Compared with other approximation algorithms, the major advantage of minimax polynomial is that the algorithm is assured to produce the minimum maximum error over the range, i.e. given the same polynomial degree, it is providing a better accuracy than the other polynomial approximations.

4.2.2 Accuracy Requirement

For the discussion about accuracy in this thesis, we generally use ulp (unit in the last place) to describe the errors. For a fixed-point number with f fractional bits, 1 ulp equals to 2^{-f}.

For MUL/DIV operations, LNS wins by introducing no rounding errors, compared to a 2^{-f-1} relative rounding error for FLP MUL/DIV. However, LNS becomes the worse one for the four LNS arithmetic functions (f_1 to f_4). These transcendental functions can only be approximated, which already brings a half ulp error for the rounding in the last step.

Existing works [Lew93], [LB03], [CCSK00] report that two or three extra bits need to be calculated in order to achieve a Better-Than-Floating-Point (BTFP) error behaviour for LNS ADD/SUB. On the other hand, Arnold et al. [AW01a, Arn01] propose that faithful rounding (1 ulp error requirement) is good enough for some LNS applications, and can greatly save hardware resources.

When generating the LNS units, my library generator only requires faithful rounding for the evaluation of the four functions. The maximum error of the hardware designs for the four functions should be less than 1 ulp, i.e. 2^{-f}. If an application requires BTFP units, as the generator provides LNS arithmetic units for various bit widths, we can calculate extra bits and then round to the required bit width, so as to achieve BTFP accuracy.

We define Δ as the number of extra bits we need to calculate to achieve BTFP accuracy. Then the BTFP unit for x fractional bits can be achieved by rounding the result of a faithful unit for $x+\Delta$ bits to x bits. In logarithmic value domain, the error bound for the parts excluding the last rounding is $2^{-x-1-\Delta}$, and the error bound for the last rounding is 2^{-x-1}. Thus, the total error in logarithmic value domain is $2^{-x-1}+2^{-x-1-\Delta}$. To achieve BTFP accuracy for x bits, we have to ensure $2^{2^{-x-1}+2^{-x-1-\Delta}}-1<2^{-x-1}$, which approximates as $ln2\times(2^{-x-1}+2^{-x-1-\Delta})<2^{-x-1}$, and gives $\Delta>1.18$. Thus, to achieve a BTFP LNS unit for x bits, we generally need to calculate two extra bits and modify the faithful rounding unit for $x+2$ bits to round to x bits.

Note that the above calculation is based on a very conservative model, which assumes maximum error in each single rounding step. By doing an exhaustive test, our precision experiments

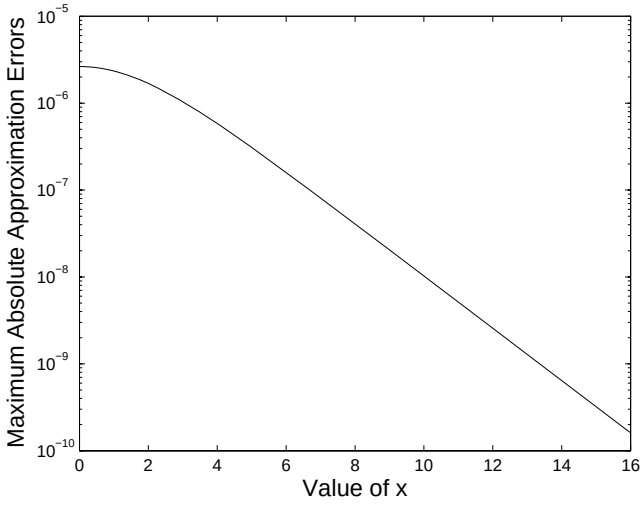


Figure 4.1: Approximation difficulty of function f_1 using degree-one minimax polynomial. The difficulty is described by the maximum absolute approximation errors in different segments. The interval of [0, 16] is divided into 256 small segments.

(detailed in Section 4.3.5) show that one extra bit is already enough to achieve BTFP accuracy for 32-bit LNS adder and subtractors.

4.2.3 Segmentation Method

To segment the evaluation range properly, we first check the approximation difficulty of the four functions by investigating their maximum approximation errors over the range.

Figure 4.1 shows the results of f_1. We divide the range of [0, 16] into uniform segments with a segment length of 2^{-4}, and record the maximum absolute approximation error of each segment using degree-one minimax polynomials. The error values provide a rough indication of the approximation difficulty within this segment. When x goes from 16 to 0, the error values increase by around 10^5 times. The same change can be observed for different degrees from one to five.

Function f_2 shows an even faster increase. When x goes from 16 to 0, the absolute approximation error increases by more than 10^{15}.

To deal with the fast variation of approximation difficulty in different ranges, we use a non-

```
##########################################################################
Segmentation of function f(x) in [a, b] with an error requirement of E
##########################################################################

begin = a; end = b; K = 1;

while ( end != begin ){

    //handle the right half.
    mid = (begin+end)/2; seg_len = (b-a)/2^K;
    compute the max_error for approximation of segment [mid, mid+seg_len];

    if ( max_error ≥ E ) {
        //test fails, need to compute with more segments
        K = K+1;
        //jump to the next iteration of while
        continue;
    }
    else {
        //test passes, divide with the current K
        divide [mid, end] into 2^K uniform segments;
        end = mid;
    }

    //handle the left half.
    compute the max_error for approximation of segment [begin, begin+seg_len];
    if ( max_error ≥ E ) {
        //test fails, jump to the next iteration;
        continue;
    }
    else {
        //test passes, segmentation is finished
        divide [begin, mid] into 2^K uniform segments;
        end = begin;
    }
}
```

Figure 4.2: Right-to-left adaptive divide-in-halves segmentation approach.

uniform adaptive divide-in-halves segmentation approach for f_1 and f_2. Figure 4.2 gives a detailed description of the approach in a right-to-left manner, which suits functions that become more difficult to approximate from right to left. Generally, we divide the range into two halves. For the right half, we try to find a proper K value (the right half is divided into 2^K uniform segments) that meets the error requirement. We then try to handle the left half with the same K. If it also meets the error requirement for the left half, the segmentation is finished; otherwise, we divide the left part into halves and continue the process recursively.

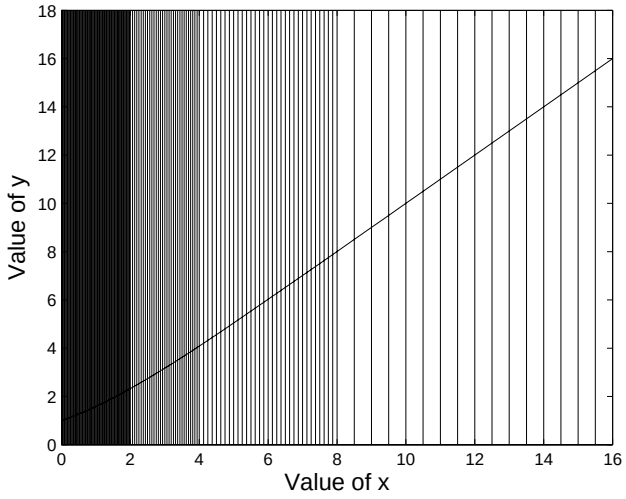


Figure 4.3: An example of segmenting the range of $[0, 16]$ for the evaluation of LNS ADD function (f_1) using degree-one minimax polynomial. The evaluation is for LNS numbers with 13 fractional bits. The error requirement is 0.3 ulp, i.e. $0.3 \cdot 2^{-13}$.

Figure 4.3 shows an example of our segmentation approach. In this example, we use degree-one minimax polynomials to evaluate the LNS ADD function (f_1) over the range of $[0, 16]$. The error requirement for the polynomial approximation is 03 ulp $= 0.3 \times 2^{-13}$. As shown in the figure, the lines demonstrate the resulting segmentation of the range. The basic principles are: firstly, as the function becomes more and more difficult to approximate from right to left (as shown in Figure 4.1), the segmentation becomes more and more dense when x approaches zero; secondly, the adjustment of the segmentation density only happens at the middle point of the interval needed to be segmented. The first principle makes the algorithm adaptable to the change of approximation difficulty over the range, while the second one ensures that the address encoding of the table can be easily performed via detection of the leading-one position (more discussion can be found in Section 4.3.2).

Table 4.1 shows the number of segments needed to evaluate a 32-bit LNS ADD function (f_1) in our design, compared to other existing LNS adder implementations. Using degree-two minimax polynomial approximation, our design with one extra bit achieves BTFP accuracy with only 416 segments, compared to 768 segments in [Lew93] and 1536 segments in [CCSK00]. Our faithful design uses 264 segments, slightly more than the 234 segments in [Arn01]. However, the design in [Arn01] applies a multiple-of-three segmentation, and the address encoding scheme consumes

Table 4.1: Number of segments needed to evaluate a 32-bit (m=8, f=23) LNS ADD function. A detailed introduction about the Taylor, Lagrange, and minimax polynomial approximations is given in Section 4.2.1.

	BTFP designs			faithful designs	
	[Lew93]	[CCSK00]	33-bit[a]	[Arn01]	32-bit
# Segments	768	1536	416	234	264
Algorithm	Lagrange	Taylor	minimax	Lagrange	minimax
Degree	two	one[b]	two	two	two

[a]The analysis based on our conservative error model requires two extra bits to achieve BTFP accuracy. However, the exhaustive precision test (detailed in Section 4.3.5) shows that one extra bit already provides BTFP accuracy for the 32-bit case.

[b]The design applies an error correction step that needs another multiplier. The complexity is similar to a degree-two polynomial.

more resource to implement than our design. In our design, as mentioned above, the adaptive adjustments only take place at half points, which provide a relatively easy address encoding mechanism for the coefficient tables when we map the approximation method into hardware design.

Conversion functions f_3 and f_4 have a small evaluation range. Meanwhile, the approximation errors change by only seven times for f_3 and two times for f_4 over the range, compared to 10^5 for f_1 and 10^{15} for f_2. Thus, we use uniform power-of-two segmentation for them.

4.2.4 Selection of Polynomial Degree

In piecewise polynomial approximation, there is a tradeoff between the polynomial degree and the density of segmentation. In a practical FPGA implementation, the tradeoff maps into usage of different hardware resources. A higher polynomial degree reduces the number of segments needed, while consumes a larger number of multiplications and additions, i.e. more hardware multipliers (HMULs) and logic slices. On the other hand, a larger number of segments require a larger number of block RAMs (BRAMs) to store the polynomial coefficients. As the cost of logic slices (used to implement additions) is relatively small, we only consider the tradeoff between HMULs and BRAMs in this problem.

The BRAM on Xilinx FPGAs stores 18K bits, which can be mapped into different organisations

from 512 36-bit values to 16384 1-bit values [Xil07b]. To improve the utilisation rate of the BRAM storage, one strategy is to keep the number of segments close to the power of two values from 512 to 16384.

The other issue, as discussed above, is to keep a balance point between the BRAMs and HMULs consumed for one arithmetic unit. Minimising one type of resource may lead to too much consumption of the other type, and increase the total cost consumed for one LNS arithmetic unit.

To determine the proper balance point, for a given bit-width value, we run the segmentation method with different polynomial degrees, and find out the number of segments for each different degree. Using the number of segments and bit-width values, we apply the equations (4.6) and (4.7) to compute a rough estimations of BRAM and HMUL numbers. In equations (4.6) and (4.7), FBW denotes the fractional bit width of the LNS number. P denotes the polynomial degree. N_{HMUL}, N_{BRAM} and N_{SEG} represent the numbers of HMULs, BRAMs and segments.

$$N_{HMUL} = \left\lceil \frac{FBW}{18} \right\rceil^2 \times P \tag{4.6}$$

$$N_{BRAM} = \begin{cases} \lceil \frac{FBW}{36} \rceil \times (P+1) & : \quad N_{SEG} \in (0, 512] \\ \lceil \frac{FBW}{18} \rceil \times (P+1) & : \quad N_{SEG} \in (512, 1024] \\ \cdots & : \quad \cdots \\ \lceil \frac{FBW}{1} \rceil \times (P+1) & : \quad N_{SEG} \in (8192, 16384] \end{cases} \tag{4.7}$$

To estimate the number of HMULs, we assume the bit widths of all the multiplication operands are FBW (in practical designs, the bit widths of the variables are optimised to different values in a range close to FBW). Thus, a multiplication consumes $\lceil \frac{FBW}{18} \rceil^2$ HMULs (HMULs on Xilinx FPGAs support 18-by-18 multiplications [Xil07b, Xil07a]), and we need P multiplications for a degree-P polynomial. For estimation of BRAMs' cost, based on the number of segments, we can determine whether to organise the BRAM as 512-by-36-bit or 16384-by-1-bit or other

Table 4.2: Examples of determining the optimal polynomial degree for LNS ADD function $f_1(x)$. The bit-width setting is described in the form of 'sign:integer:fraction'. P stands for the polynomial degree value. n_{seg}, N_{HMUL} and N_{BRAM} stand for the number of segments, HMULs and BRAMs respectively. 'esti.' and 'exp.' refer to estimated and experimental results, which are very close to each other as shown in this table. In this example, we define the total cost as the sum of estimated HMUL and BRAM numbers.

Bit Width	P	n_{seg}	N_{HMUL}		N_{BRAM}		Total Cost
			esti.	exp.	esti.	exp.	
1:7:13	**1**	**144**	**1**	**1**	**2**	**2**	**3**
	2	24	2	2	3	3	5
1:8:23	1	6272	4	4	24	33	28
	2	**264**	**8**	8	**3**	3	**11**
	3	60	12	10	4	4	16
1:11:52	4	1856	36	33	30	27	66
	5	**544**	**45**	**42**	**18**	**17**	**63**

structures, and compute the number of BRAMs accordingly.

When we investigate LNS arithmetic at an unit level, we can simply define the balance point as the polynomial degree with the least total number of BRAMs and HMULs. Table 4.2 shows an example of LNS addition function f_1 with a number of typical bit-width settings. The estimation numbers given by the equations match the experimental results quite well. We select the polynomial degree with the least sum of estimated HMUL and BRAM numbers, shown in bold in the table.

However, when we employ these arithmetic units in a large application design, the primary goal becomes to fit the whole design into the limited area of one FPGA device. Suppose we have a design that requires four 64-bit LNS ADD units, and we want to implement the design with Xilinx Virtex-4 FX100 FPGA, which has 376 BRAMs and 160 HMULs. The optimal 64-bit unit in Table 4.2 (shown in the last row) requires $45 \times 4 = 180$ HMULs plus $18 \times 4 = 72$ BRAMs, and does not fit into the device. However, as the number of BRAMs is still much less than the constraint, we can solve the problem by using a smaller polynomial degree and trading-off more BRAMs to reduce the number of HMULs.

Based on this consideration, our LNS library generator (detailed in Section 4.3) reserves the interface for users to configure the polynomial degree values for each arithmetic unit. Meanwhile, the numbers of segments for polynomial degrees one to five are all recorded for all supported

bit-widths. Applying the above estimation equations, we can acquire the rough cost of HMULs and BRAMs for arithmetic units using different polynomial degrees. Based on the rough cost estimation, the users can use enumeration or other more intelligent optimisation algorithms to find out the optimal polynomial degree that fits the design into the targeted FPGA device.

4.2.5 Error Ratio Adjustment

The error of a hardware function evaluation unit consists of two parts [LGC+06]: (1) approximation error: the error due to the mathematical approximation method, provided that we compute with infinite precision; (2) quantisation error: the rounding and truncation errors due to finite precision of hardware number representation. For convenience, we define G as the ratio between the approximation error requirement and the total error requirement. Based on experimental results, setting G to 0.3 minimises the total cost in general cases. However, for cases with an 'edge' segment number (the segment number is slightly larger than the supported power-of-two dimension sizes from 512 to 16384), we perform an extra adjustment of the error ratio G to identify the optimal setting. For instance, with G=0.3, 544 segments are needed to calculate 64-bit f_1 with a degree-five polynomial. Since 544 is slightly larger than the bound of 512, the BRAM has to organise the data in an address space of 1024 elements, which wastes almost 50% of the storage. To reduce this big waste, we try with different G values from 0.05 to 0.45 with a step size of 0.05.

Table 4.3 shows the area and latency of LNS ADD designs with different G values. When the value of G decreases from 0.3 to 0.05, the number of segments increases from 544 to 832. However, the number of BRAMs does not change as they are both using the 1024-by-16-bit configuration of the BRAM, and the number of slices only decreases by 24. On the other hand, when the value of G increases from 0.3 to 0.45, the number of segments drops below 512, and the number of BRAMs falls from 17 to 10. Although the number of slices increases by 31, it is negligible (2.8%) when compared with the BRAM reduction (41%).

In our LNS arithmetic library generator (described in Section 4.3), we keep the error ratio G as a configurable parameter. In normal cases, G takes the default value of 0.3. In 'edge' cases

Table 4.3: Area and latency of 64-bit ($m = 11$, $f = 52$) LNS ADD units with different G values. G indicates the ratio between the approximation error requirement and the total error requirement.

G	segments	slices	BRAMs	HMUL	latency
0.05	832	1079	17	42	72.9 ns
0.3	544	1103	17	42	73.8 ns
0.45	480	1134	10	42	75.7 ns

similar to the above one, the users can specify different G values to achieve a more efficient usage of the hardware resources.

4.2.6 Evaluation of f_1 and f_2

As shown in Figure 4.1, the error in calculating f_1 and f_2 decreases very quickly for all polynomial degrees when x becomes large. This is because the linear function $y = x$ gives a quite accurate approximation for f_1 for large x. Thus, we can find a point $x = 2^r$, which assures that for all the x values on the right side of this point, the difference between $y = x$ and f_1 is within the error requirement. The approximation of f_1 is then divided into three ranges:

- If $x \geq 2^r$, f_1 is approximated with $y = x$.
- If $2 \leq x < 2^r$, we use the adaptive divide-in-halves segmentation approach for f_1.
- If $0 \leq x < 2$, for the degree-one, three, four and five cases, we continue the adaptive divide-in-halves approach until the end. For the degree-two case, as shown in Figure 4.4, the error keeps on decreasing after the point of $x = 2$. Therefore, instead of continuing the divide-in-halves approach, we first divide the range into sections [0, 0.5] and [0.5, 2], and further divide each section into uniform segmentations.

To circumvent the zero singularity of f_2's evaluation, we use the function decomposition approach [LB03], [PS96] to transform $f_2 = log_2(2^x - 1)$ into $f_2 = g + f_3 = log_2(\frac{2^x-1}{x}) + log_2(x)$, thus f_2 can be evaluated through two other functions which are relatively easier to approximate.

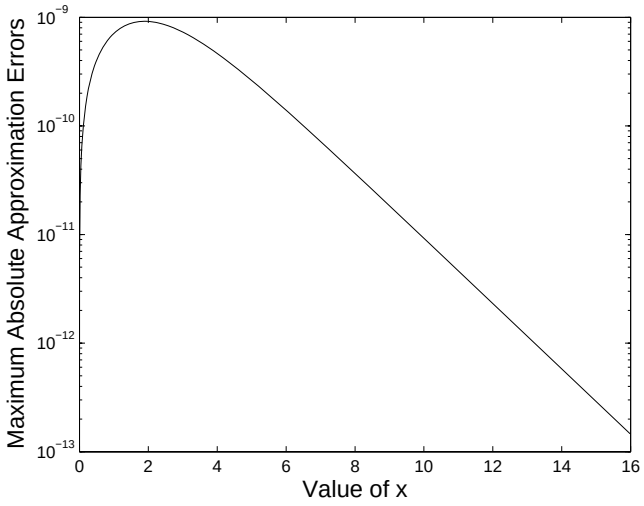


Figure 4.4: Approximation difficulty of function f_1 using degree-two minimax polynomial. The difficulty is described by the maximum absolute approximation errors in different segments. The interval of [0, 16] is divided into 256 small segments.

To find out under what kind of condition we shall replace f_2 with $g + f_3$, we perform a similar approximation difficulty investigation as Section 4.2.3 to f_2 and g. Function f_3 is not considered, as it is much easier to approximate. As shown in Figure 4.5, from the point around $x = 6.703125$, g becomes easier to approximate than f_2. However, as the error of g and f_3 are combined to make the total error, we require the error of g to be below half of the error of f_2. Based on these considerations, we use the decomposition for the range of [0, 4] (at point $x = 4$, error of g is about 16.5% of f_2), rather than the range of [0, 2] in [LB03].

Similar to f_1, when x is very large, $y = x$ provides a good approximation of f_2. The approximation of f_2 is then divided into three ranges:

- If $x \geq 2^r$, f_2 is approximated with $y = x$.
- If $4 \leq x < 2^r$, we segment the range with the adaptive divide-in-halves approach and approximate each segment with a minimax polynomial.
- If $0 \leq x < 4$, approximation of f_2 is decomposed into g and f_3. As the approximation difficulty of both g and f_3 does not vary too much, we use uniform segmentation for functions g and f_3.

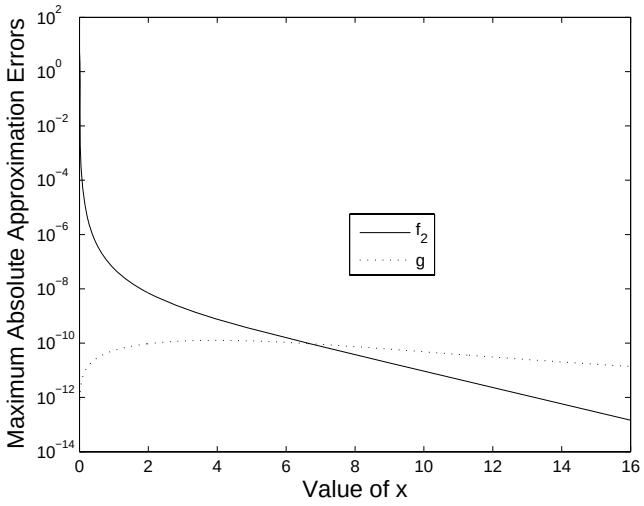


Figure 4.5: Approximation difficulty of functions f_2 and g using degree-two minimax polynomial. The difficulty is described by the maximum absolute approximation errors in different segments. The interval of [0, 16] is divided into 256 small segments.

4.3 LNS Arithmetic Library Generator

4.3.1 General Structure

As described in Section 4.2, the optimal choice for design options, such as polynomial degree and the ratio between approximation and quantisation errors, can be different for different cases. Therefore, instead of providing fixed arithmetic libraries, we develop a library generator that keeps these design options and the bit-width settings as reconfigurable parameters for the users.

The general structure of our LNS library generator is shown in Figure 4.6. We use Maple [Wat] as the mathematical approximation design tool, as it provides convenient support for polynomial approximation methods, such as Chebyshev and minimax algorithms (discussed in Section 4.2.1). Our Maple programs segment the function evaluation range using the approach discussed in Section 4.2.3 and compute the coefficients of each segment. The result files, which describe all the segments' ranges, errors and coefficients, are stored into the coefficient file repository.

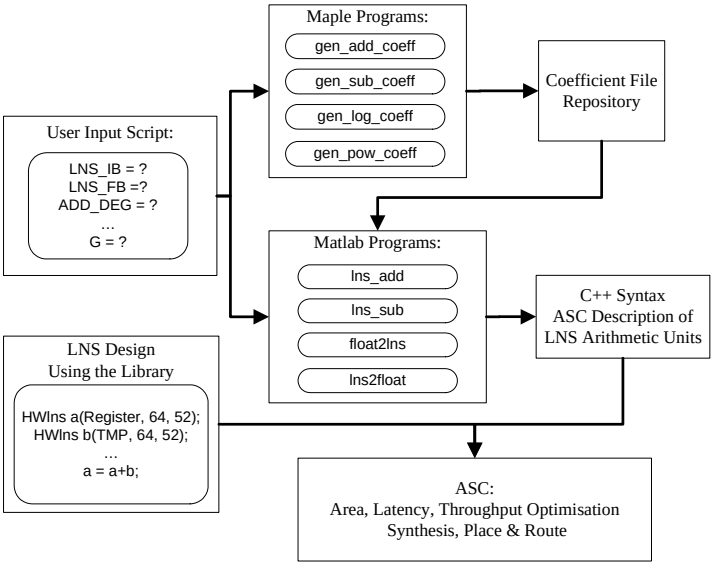


Figure 4.6: General structure of the LNS library.

We use Matlab [Mat05] as the interface tool between the mathematical and the hardware design levels. The Matlab programs read coefficient files from the repository, and generate the arithmetic designs described in the C++ syntax format of A Stream Compiler (ASC) [Men06].

As a high-level FPGA programming tool, ASC provides hardware data-types, such as integer (HWint), fixed-point (HWfix) and floating-point number (HWfloat), with configurable bit-widths. It also provides configurable optimisation modes, such as area, latency and throughput. By specifying the throughput mode, ASC automatically generates a fully-pipelined circuit design for the application. These features make ASC an ideal hardware compilation tool for our arithmetic library.

To generate LNS arithmetic units with specific settings, the user only needs to write a simple script file which specifies the bit-widths of the LNS number, the approximation error ratio G and the polynomial degrees to be used for the four functions. With these parameters, the script file calls the corresponding Maple and Matlab programs, which automatically generate the LNS library file that contains a full set of LNS arithmetic units described in ASC syntax. With the LNS library file, users can then design their target applications using LNS as a normal data-type that supports basic operators such as $+, -, *, /$.

4.3.2 Address Encoder of Coefficient Table

The circuit design of the LNS arithmetic units consists of three major parts: the coefficient tables, which store the coefficients for different segments; the address encoder, which calculates the coefficient table address for an input value; and the polynomial evaluation unit, which reads the coefficients with the corresponding address, and calculates the approximation result. This section discusses the design of the address encoder.

For uniform power-of-two segmentations such as f_3 and f_4, after transforming the original range into [0,1], we can simply use the first K bits of the number as the address of the coefficient table. The address is more complicated to calculate for divide-in-halves adaptive segmentation. Suppose we segment the range $[0, 2^m]$ with right-to-left adaptive divide-in-halves segmentation, the range is divided into a number of sections s_0, s_1, s_2, $\cdots$, s_n, and each section s_i is divided into 2^{K_i} uniform segments. Because the sections are actually formed in a divide-in-halves manner, the distribution shall be $\{s_0 = [0, 2^p],\ s_1 = [2^p, 2^{p+1}],\ \cdots,\ s_n = [2^{p+n-1}, 2^{p+n}]\}$, $p \in \{0, \pm1, \pm2, \cdots\}$. Thus, given an input value x, based on its leading one position, we can determine the section the input value falls in, and then use the next K_i bits after the leading one to find out its segment number in the section.

Figure 4.7 shows the hardware architecture of the address encoder for a 32-bit LNS adder. Two tables are used to calculate the coefficient table address based on the input value's leading one position, which identifies its corresponding section. The offset table records the address offset of each section, i.e. the address of the first segment in that section. The table of address bits maps the corresponding K_i bits of the input value into a number. The sum of the section offset and the K_i address bits gives the table address of the input value.

4.3.3 BRAM Cost Minimisation

When using LNS arithmetic units to implement applications, we try to minimise BRAM costs by sharing BRAMs among different units or packing the coefficients of different degrees into the same BRAM.

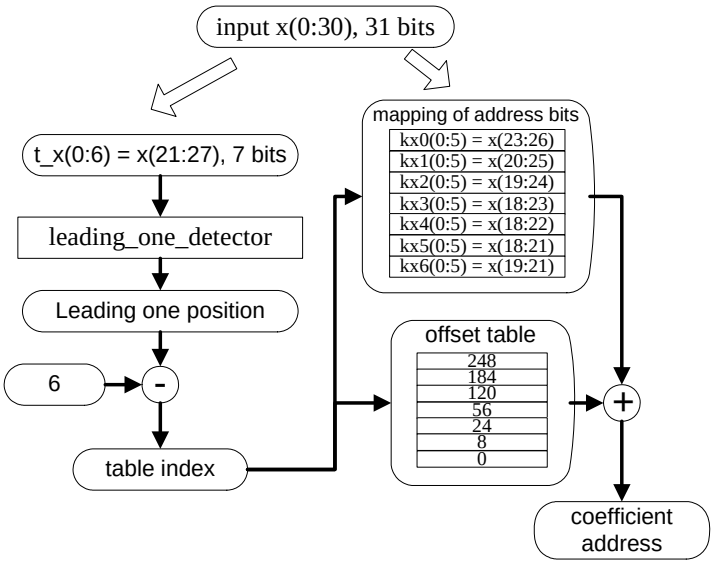


Figure 4.7: Address encoder for 32-bit LNS ADD. 'a(x:y)' denotes bits x to y of variable a.

As a BRAM supports two concurrent reading ports, a pair of identical arithmetic units can read the coefficients from the same BRAMs. On the other hand, if the number of segments or the bit-width of the coefficient is small, we can organise the coefficients of two different degrees into the same BRAM and set up their reading addresses with a predefined offset.

Generally, we share BRAMs between identical LNS ADD and SUB units, as applications normally consume a large number of ADD and SUB units. For conversion units (f_3 and f_4), we compact coefficients of different degrees into the same BRAM, as conversion functions have a smaller number of segments and smaller bit-widths compared to LNS ADD and SUB functions.

In our LNS library generator, a module called coefficient manager automatically performs the task of minimising BRAM costs. When we generate the LNS ADD/SUB units, instead of instantiating BRAMs for coefficients in each unit, we send a request to the coefficient manager with information of the function type and the bit-width. The coefficient manager analyses the requests for BRAMs, and checks whether there are sharing possibilities. If the coefficient manager locates a matched set of BRAMs with vacant reading ports to share, it connects the requesting unit to the located reading ports, and updates the read port usage of this set of BRAMs. If there is no BRAMs both matched and vacant, the coefficient manager instantiates a new set of BRAMs. For LNS conversion units, the coefficient manager runs an analysis of

the number of segments and the bit-width of coefficients to figure out the proper way to pack different coefficients into minimum number of BRAMs.

4.3.4 Bit-width Optimisation

In the approximation of LNS arithmetic functions, we evaluate an n-degree polynomial using Horner's rule. Horner's Rule is a rule for polynomial computation which both reduces the number of necessary multiplications and results in less numerical instability due to potential subtraction of one large number from another. The rule can be illustrated as the following equation:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = (\cdots (a_n x + a_{n-1})x + \cdots)x + a_0 \tag{4.8}$$

Based on Horner's rule, the general form of an n-degree polynomial in our LNS library is, $y = (\cdots (c_n x + c_{n-1})x + \cdots)x + c_0$, where $c_0 \cdots c_n$ are the polynomial coefficients.

The hardware unit contains $n+1$ coefficients ($c_0 \cdots c_n$), and $2n$ intermediate results ($b_n = c_n x$, $d_n = b_n + c_{n-1}$, $b_{n-1} = d_n x$, $\cdots$, $b_1 = d_1 \cdot x$, $d_1 = b_1 + c_0$). To perform bit-width optimisation on all the variables, we adopt the approach proposed by D. Lee, et al. [LGC+06, LGML05]. The approach uses affine arithmetic [ACS94] to induce a conservative error model that gives a strict bound to the error of the hardware design, and build up a cost function that estimates the hardware cost. With the error function and the cost function, we then use the adaptive simulated annealing (ASA) method [Ing04] to find the heuristic best solution in the multi-dimensional parameter space.

4.3.5 Precision Test

The minimax approximation algorithm provides an upper bound for the maximum approximation error, while the error model based on affine arithmetic provides the upper bound for the

Table 4.4: Exhaustive precision test results for 32-bit LNS arithmetic units. When using one extra bit, we apply a 33-bit 'faithful' design and round the result to 32-bit in the final rounding step. '# BTFP' and '# NBTFP' refer to the number of BTFP (better-than-floating-point) cases and non-BTFP (not-better-than-floating-point) cases respectively. '$error_{MAX}$' is the recorded maximum relative error.

LNS ADD	# BTFP	# NBTFP	$error_{MAX}$
32-bit 'faithful'	268415966	19490	6.5e-8
one extra bit	268435456	0	4.9e-8
LNS SUB	# BTFP	# NBTFP	$error_{MAX}$
32-bit 'faithful'	268430674	4783	5.96e-8
one extra bit	268435456	0	5.03e-8

maximum quantisation error. Combining these two error bounds together, we ensure that the total absolute errors of the generated LNS arithmetic units are less than 1 ulp.

To perform a further verification of the precision, we include a precision test in the Matlab functions that generate the arithmetic units. The precision test generates 100000 uniform random values within the input range, and performs a bit-accurate circuit simulation to check whether the errors are within 1 ulp. The ASC description of the unit is only generated when the precision test passes.

For the most typical 32-bit LNS units, we run exhaustive precision tests. For value x larger than 32, we approximate functions $f_1(x) = log_2(2^x + 1)$ and $f_2(x) = log_2(2^x - 1)$ using $y = x$, thus the relative errors of $f_1(x)$ and $f_2(x)$ are assured to be less than $|2^{log_2(2^{32}\pm1)} - 2^{32}|/2^{log_2(2^{32}\pm1)} \approx 2.33e-10$. The maximum relative error of 32-bit (single precision) FLP ADD/SUB operations is $2^{-24} \approx 5.96e-8$. Thus, for input values larger than 32, our units are assured to provide BTFP accuracy.

For values below 32, we test every possible input. For 23 fractional bits, the total number of values to test is $2^{28} = 268435456$. As shown in Table 4.4, the precision of our 32-bit 'faithful' LNS units are already very close to BTFP accuracy. For LNS ADD and SUB units, only 19490 and 4783 cases out of 268435456 produce relative errors larger than the maximum FLP relative error. With one extra bit, both LNS ADD and SUB design achieves BTFP accuracy in all possible cases.

4.3.6 Experimental Results for LNS Arithmetic Units

Using ASC, we map all the arithmetic units onto the Sepia card [MSH99] and Xilinx Virtex-II XC2V6000 FPGA to test their performance and hardware cost. Table 4.5 shows the area and latency of the LNS arithmetic units automatically produced by our library, compared with the designs in [HBUH05]. There is no clear description about whether a faithful rounding or a BTFP accuracy is achieved in [HBUH05], thus we compare with both our faithful rounding and BTFP units. For LNS ADD units, our designs consume 5.9% to 37.2% fewer slices, 41.7% to 75% fewer HMULs, but consume more BRAMs for 64-bit ADD. For LNS SUB units, we use 25% fewer BRAM and 62.5% fewer HMULs in 32-bit case. In the 64-bit case, our SUB units consume more resources than [HBUH05]. For converters, our designs consume much less resources, and support a larger number of units on one FPGA board. Meanwhile, for most units, our designs also provide 20% to 50% reduction in latency.

In practical arithmetic computation, we normally do not know the signs of the two operands in advance. Depending on whether they have the same sign or not, the operation of $+/-$ can be mapped into either LNS ADD or SUB function. Thus, we also generate mixed units that can perform both addition and subtraction (not concurrently). Based on the results summarised in [LB03], Table 4.6 gives a comparison of some existing designs with our BTFP mixed units that can perform both ADD and SUB operations. Note that the LNS design by Lewis [Lew93] is not implemented on an FPGA, but M. Arnold [AW01a] gives an estimation for the hardware cost of its data-path in slices. For the estimation of BRAM cost, the design by Lewis [Lew93] requires 91K bits of ROM for the 'weak' error mode (relax the approximation error requirement for f_2 when x gets near zero), which is equivalent to about five 18Kbit BRAM. Compared with these designs, our automatically generated units cost more HMULs and a similar number of BRAMs. However, our unit consumes the least amount of slices, and provides the shortest latency.

The other advantage of our LNS library is the automatic generation of different bit-width arithmetic units according to the user requirement. Figure 4.8 shows the latency and area cost of our ADD/SUB units at typical bit-widths. The basic arithmetic unit designs are tested on Xilinx Virtex-II XC2V6000 FPGA as well as by software simulation. When the bit-width

Table 4.5: Area and latency of our auto-generated LNS arithmetic units, compared with the designs in [HBUH05]. F stands for our faithful rounding units (1 ulp error bound in logarithmic domain) while B stands for our BTFP units (less than 0.5 ulp error bound in floating-point domain). Number of units per FPGA is calculated based on Xilinx Virtex-II XC2V6000, which has 33792 slices, 144 BRAMs and HMULs.

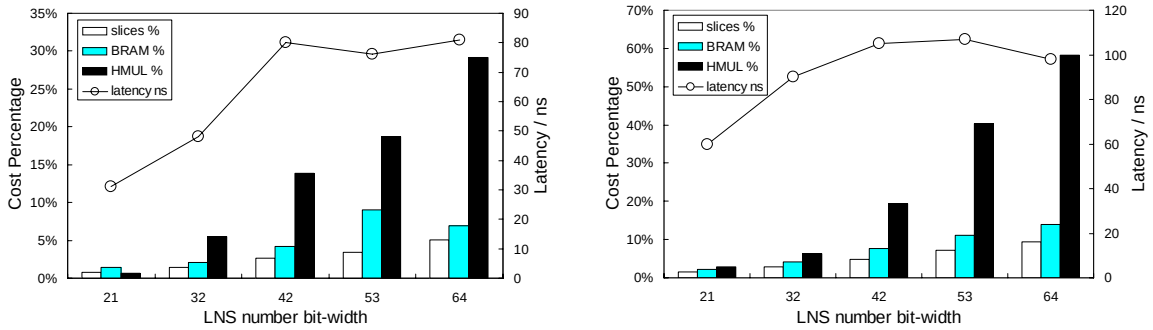
		LNS ADD			LNS SUB		
		[HBUH05]	F	B	[HBUH05]	F	B
32-bit LNS $m = 8, f = 23$	slices #	**750**	**471**	**490**	838	989	1044
	BRAM #	**4**	**3**	**3**	**8**	**6**	**6**
	HMUL #	**24**	**8**	**6**	**24**	**9**	**9**
	latency (ns)	**91**	**48**	**58**	95	90	91
	units per FPGA	**6**	**18**	**24**	**6**	**16**	**16**
64-bit LNS $m = 11, f = 52$	slices #	**1944**	**1727**	**1830**	2172	3140	3410
	BRAM #	8	10	18	16	20	30
	HMUL #	**72**	**42**	**42**	72	84	101
	latency (ns)	**107**	**81**	**84**	110	98	122
	units per FPGA	2	3	3	1	1	1

		FLP to LNS			LNS to FLP		
		[HBUH05]	F	B	[HBUH05]	F	B
32-bit LNS $m = 8, f = 23$	slices #	163	286	327	236	376	408
	BRAM #	**24**	**2**	**2**	**4**	**2**	**2**
	HMUL #	0	3	3	**20**	**3**	**3**
	latency (ns)	**76**	**51**	**52**	**72**	**47**	**46**
	units per FPGA	**6**	**48**	**48**	**7**	**48**	**48**
64-bit LNS $m = 11, f = 52$	slices #	**7631**	**1029**	**1063**	**3152**	**996**	**1026**
	BRAM #	0	5	5	**14**	**5**	**5**
	HMUL #	0	36	33	**226**	**30**	**30**
	latency (ns)	**380**	**77**	**73**	84	111	111
	units per FPGA	4	4	4	**0**	**4**	**4**

Table 4.6: Area and latency of our 32-bit mixed BTFP LNS arithmetic units, compared with other existing designs.

design	Accuracy	BRAM	HMUL	slices	latency
MIX	BTFP	9	12	1066	83
[Lew93]	BTFP	5	0	2300	x
[MTP+02]	BTFP	96	0	1300	160
[LB03]	BTFP	6	4	1210	125
[DD07]	faithful	0	0	3904	97

increases from 21 to 64, the consumed slices increase from 0.8% to 5.1% of Virtex-II XC2V6000 for ADD, and from 1.6% to 9.3% for SUB. BRAMs increase from 1.4% to 9.0% for ADD (64-bit ADD use fewer BRAMs than 53-bit, because we perform the G ratio adjustment for 64-bit case), and from 2.1% to 14.9% for SUB. These costs are acceptable for applications with



(a) LNS ADD units for different bit-width.

(b) LNS SUB units for different bit-width.

Figure 4.8: Hardware resource cost and latency of faithful rounding LNS arithmetic units for different bit-widths. The percentage of slices, BRAM and HMUL are calculated based on Virtex-II XC2V6000, which has 33792 slices, 144 BRAMs and HMULs.

various accuracy requirements. However, the usage of HMULs increases from 0.7% to 29.2% for ADD and from 2.8% to 58.3% for SUB, which makes it difficult to map high precision LNS arithmetic units onto an FPGA board.

Compared with previous work [HBUH05], our LNS designs achieve significant reduction on both area cost and latency. One major reason for the area and latency reduction of our LNS units might be a systematic optimisation for all the related design options, which include: (1) the approximation polynomial, (2) the segmentation of the range, (3) selection of the polynomial degree, (4) the error ratio between the approximation error and the quantisation error, (5) the bit widths of the intermediate variables.

For some of the design options, we can determine the optimal or close-to-optimal choices based on theoretical knowledge or analytical methods. E.g., for approximation polynomials, we choose to use the minimax polynomials because they provide the minimum approximation error in theory. For the bit widths of the intermediate variables in the design, we apply the ASA (Adaptive Simulated Annealing) method to produce a result that is very close to the optimal solution [LGC+06].

For other parameters, such as the polynomial degree and the error ratio between different error parts, we solve the problem by applying the automation feature of the tool. As the entire process of our LNS library generation is automated, we can apply an automated exploration

of the possible values, and determine the most appropriate choice from an empirical point of view. Therefore, another lesson we learnt here is that an automated design generation process can be quite useful for finding out a good solution from a large design space.

Note that, the optimisation techniques and the automated design generation process used here for evaluating LNS arithmetic functions can also be applied to evaluation of other elementary functions.

4.4 Comparing LNS and FLP Designs on FPGAs

4.4.1 Comparison of Arithmetic Units

Combining our LNS library introduced in Section 4.3 and ASC's support for reconfigurable FLP arithmetic units, we run a profiling of LNS and FLP arithmetic units with 231 different bit-width settings (7 different integer bit widths and 33 different fractional bit widths). The integer bit widths of LNS numbers (corresponding to the exponent bit widths of FLP numbers) vary from 4 to 10; while the fractional bit widths of LNS numbers (corresponding to the significand bit widths of FLP numbers) vary from 13 to 45.

Figure 4.9 shows a comparison between LNS and FLP on basic arithmetic units. We apply the model based on relative silicon area proposed by M. Haselman, et al. [HBUH05], to describe the cost of BRAMs and HMULs in equivalent number of slices. In this model, a BRAM equals 27.9 slices and a HMUL equals 17.9 slices. With the fractional bit width changes from 13 to 45, the area cost of FLP ADD increases from 151 to 548 slices, while the area cost of LNS ADD/SUB unit increases from 729 to 3766 slices. On the other hand, with the same change of bit widths, the LNS MUL/DIV only requires 11 to 27 slices, while the area cost of FLP MUL increases from 87 to 445 slices, and the area cost of FLP DIV increases from 451 to 4365 slices. In general, LNS arithmetic provides very efficient implementation for multiplication and division, at the price of very expensive addition and subtraction implementations.

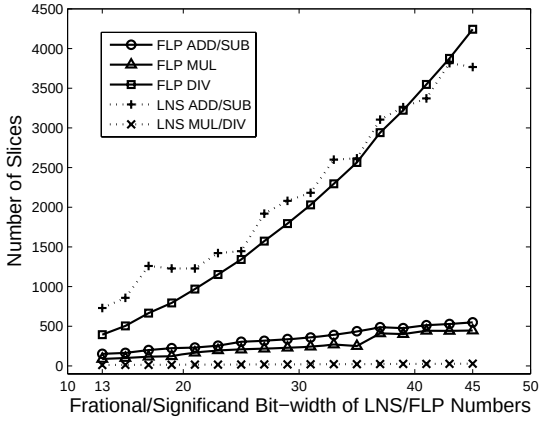


Figure 4.9: Area cost comparison of LNS and FLP arithmetic units.

4.4.2 A Two-level Area Cost Estimation Tool

To facilitate comparison of area cost between LNS and FLP designs, we develop a two-level area cost estimation tool: the first level is an area model based on table lookup or interpolation of recorded area results, and produces area cost estimation in less than a second, with an average error around 5%; the second level maps, places and routes the actual design onto FPGAs to acquire experiment results which are more accurate. Depending on the circuit size, the mapping, placing and routing of a design takes hours to days.

The area model serves as an extension of the ASC compiler. The tool derives a data flow graph of the design from the ASC description, and calculates the area cost of the arithmetic units based on the exploration result of 231 different bit-width settings (7 different integer bit width and 33 different fractional bit width). If the bit-width values of the arithmetic unit are already included in the 231 cases, we directly look up the numbers of slices, HMULs and BRAMs. Otherwise, we make an interpolation of the nearest values to acquire an estimation of the cost. When we need to describe the cost of BRAMs and HMULs in equivalent number of slices, we use the model proposed by M. Haselman, et al. [HBUH05], which counts a BRAM as 27.9 slices and a HMUL as 17.9 slices.

To illustrate the accuracy of our area modelling tool, we investigate a typical design, a degree-four polynomial (poly4). Horner's rule is used to evaluate the polynomial as follows: $y =$

$(\cdots((c_n x + c_{n-1})x + c_{n-2})x \cdots)x + c_0$.

Figure 4.10 shows the comparison between the area cost results given by area models and experiments. LNS and FLP designs optimised for both latency and throughput are compared with the corresponding area modelling results. Our area modelling tool provides accurate estimation for FLP designs optimised for either latency or throughput. The solid lines (area modelling results) match the markers (experimental results) quite well, giving a maximum error of 15.7% and an average error within 4.7%. However, for the LNS cases, the lines vary from the markers by 21 to 30%. There are two possible reasons for the difference between the estimation results and experimental results. When we combine different arithmetic units into an application, optimisation among the units may squeeze the area into a more compact mapping of the logic slices. On the other hand, when we pipeline different units together, inserting extra registers in between brings additional cost. These two factors reduce the accuracy of the area modelling when the circuit is large and complicated.

Although the area estimation of LNS designs vary from the experimental results, the lines still preserve the same trend as the markers. Based on this observation, we apply a scaling procedure to mitigate the errors. Using a typical bit-width setting (e.g. $m = 8, f = 23$), we perform the mapping of a practical circuit, collect the area cost result, and calculate a scale factor between the experimental result and the modelling result. We can then use this factor to scale the area modelling results for all the other bit-width combinations. As shown in Figure 4.10, the scaled area modelling results (shown as dotted lines) for LNS designs fit the experimental results much better, providing a maximum error of 14.3% and an average error of 4.4%.

For cases that require more accurate area cost results, our tool performs an automated mapping of designs using different number representations onto FPGAs, and parse the resource usage data from Xilinx place-and-route result files.

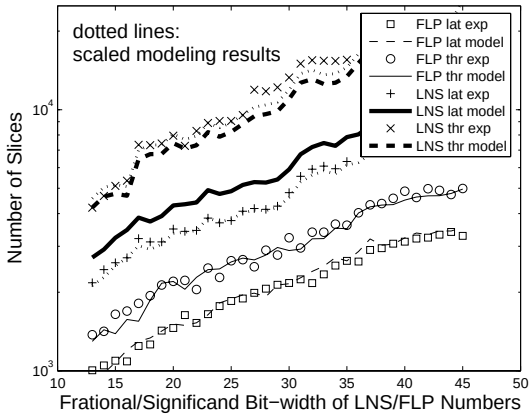


Figure 4.10: Comparison between the estimated area cost given by area modelling tool and the actual experimental results. 'lat' and 'thr' refer to designs that are optimised for latency and throughput respectively, while 'model' and 'exp' refer to area modelling and experimental results respectively.

4.4.3 Bit-Accurate Hardware Simulator

To provide a complete tool chain for comparison between LNS and FLP numbers, we develop a value simulator to investigate the precision behaviour of different number representations. The simulator performs a bit-accurate value simulation of the basic hardware arithmetics, such as addition, subtraction, multiplication, and rounding.

Similar to the area modelling tool, the value simulator is also an extension of the original ASC compiler. To achieve high efficiency of the value simulation, we use the long double data type in C++. The long double data type is implemented as 80-bit floating-point format (63-bit mantissa, 16-bit exponent, and 1-bit sign) on x86 machines. By doing bit manipulation of the mantissa and exponent values, we are able to provide bit-accurate simulation of variables up to 64 bits.

The simulator currently supports all the basic arithmetic operations of fixed-point (HWfix) and floating-point (HWfloat) hardware variables with configurable bit-widths up to 64-bit. Since the LNS arithmetic functions are evaluated using fixed-point variables, the simulator can also handle all the basic arithmetic operations of LNS hardware data-types.

4.5 Summary

This chapter demonstrated our optimised LNS arithmetic targeting reconfigurable hardware designs. In particular, we introduced a general polynomial approximation approach for LNS arithmetic function evaluations. The approach provides an adaptive divide-in-halves segmentation method, supports polynomial degrees from one to five and reconfigurable ratio between approximation and quantisation errors. We developed a library generator that produces LNS arithmetic libraries containing $+, -, *, /$ operators as well as converters between floating-point and LNS numbers. The generated libraries provide a simple-to-use address encoding mechanism for coefficient tables, on-chip block memory sharing to save coefficient storage, and bit-width minimisation based on affine arithmetic. The generated arithmetic units are tested on advanced FPGAs and in software simulation. Our evaluation showed that the generated LNS arithmetic units have significant improvements over existing LNS designs.

We think that the area and latency reduction of our generated units mainly comes from a systematic optimisation of all the related design options, such as the polynomial degree, the bit widths of the intermediate variables, etc. By optimising the design options using both analytical and empirical approaches, we manage to provide more efficient LNS arithmetic units than previous work.

To facilitate comparison between LNS and floating-point designs, we developed a two-level area cost estimation tool: the first level uses area models to give approximate results in less than a second, with a maximum error of 15.7% and an average error around 5%; the second level performs an automated mapping of different designs onto FPGAs to acquire more accurate experiment results. We also provided a bit-accurate simulator to investigate the accuracy of LNS and FLP designs.

Chapter 5

Optimised Residue Arithmetic Unit Generation

5.1 Introduction

Dating back to the Chinese Remainder Theorem (CRT) in the ancient Chinese Mathematics book "Sun Zi Suan Jing" [SunAD] (with a recent English translation in [LA04]), the theory behind Residue Number System (RNS) has existed for over a thousand years. By decomposing one large number into a number of small residue values, RNS greatly reduces the carry chain length of adders and the size of multipliers. Compared to conventional binary representations, RNS provides low latency in addition and multiplication, plus possible reduction in area and power consumption [CRNR04]. Due to these special features, RNS has long been regarded as a promising number format in digital signal processing [MM99, CS03].

However, RNS also has its inherent disadvantages when compared to binary representations. As the residue values do not contain any magnitude information, comparison, scaling and division are very difficult. Conversions between binary and residue numbers are also costly. These difficulties constrain the utilisation of RNS to only a small fraction of applications.

As a reconfigurable hardware device, an FPGA is an ideal platform to evaluate the RNS repre-

sentation. There is existing work on implementing residue arithmetic on FPGAs [Tom06], and also research efforts that use RNS to implement applications such as Intellectual Property Protection (IPP) procedures [PCGL04] and the RSA algorithm [CNPQ03] on FPGAs. However, general support for RNS arithmetic is still lacking.

To facilitate further investigations on the potential of RNS in different applications, we work on improving the arithmetic designs of RNS numbers, and developing an optimised RNS arithmetic library generator that targets FPGAs. The optimised arithmetic library generator enables fast prototyping of efficient RNS designs. By studying the tradeoffs between RNS and other number representations, we can apply RNS to suitable applications to achieve improvements in performance and precision or a reduction in resource consumption.

The major contributions of this chapter are:

- improvement of RNS arithmetic on FPGAs. For reverse converters from RNS to binary numbers, we propose a novel design which consumes up to 14.3% less area and provides lower latency than previous work [WSAS02]. The forward converter is implemented with simple modular additions. We also provide simplified solutions for specific cases of scaling and magnitude comparison operations.

- an RNS arithmetic library generator for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. We keep n as a reconfigurable parameter, so that the generator supports a wide range of RNS numbers.

- significant resource reduction by using RNS for multiplications. We can therefore fit more multipliers into one FPGA.

Using the library generator, we perform an extensive comparison between RNS and other number representations on both arithmetic units and simple benchmarks. The comparison shows that, the major advantage of RNS lies in the reduction of resources in multiplication operations.

5.2 Design of RNS Arithmetic Units

Note that this Chapter uses the same RNS notations defined in Section 2.4.2.

5.2.1 Selection of Moduli Set

As the only requirement for the moduli set is that all the modulus values shall be pairwise co-prime, there are many different possible combinations of moduli set. The choice of the moduli set is very important, and can greatly affect the complexity of the design for arithmetic units.

There are two major categories of moduli sets. One is prime-number moduli sets, which contain only prime numbers in the set. For RNS numbers that use a prime-number moduli set, multiplication can be implemented efficiently using the approach called index calculus [GL98], which is based on Galois field theory. However, as the prime modulus values are irregular, the conversions between binary and residue numbers become very difficult. The magnitude-related operations, such as comparison and scaling, are also costly to implement. The other major category of moduli set is power-of-two moduli set, the most popular one of which is $\{2^n - 1, 2^n, 2^n + 1\}$. As the modulus values are more similar to the binary representations, the conversions between binary and residue numbers are easy to achieve. For magnitude comparison and scaling, there are also methods with acceptable complexity.

In our work, we choose to use the power-of-two moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ for the following reasons: firstly, it provides simpler designs for converters and magnitude-related operations, thus is more possible to provide efficient designs for common applications; secondly, although the power-of-two moduli set cannot use the index calculus approach to implement efficient multipliers, we can utilise the dedicated hardware multipliers on FPGA platforms to implement multiplications with acceptable cost; thirdly, as it is the most commonly used one in previous work, using this moduli set makes our work comparable to most existing designs.

5.2.2 Forward Converter

The previous designs of forward converters (detailed in Section 2.4.2) break the binary number into bits, and calculate the residue values of each bit separately. In our design, as we use the specific moduli set $\{2^n-1, 2^n, 2^n+1\}$, the generation of the residue values is greatly simplified.

The representation range of the moduli set $\{2^n-1, 2^n, 2^n+1\}$ is $2^{3n}-2^n$, which approximately corresponds to binary numbers with $3n$ bits. If we divide the $3n$ bits into three groups of n bits and denote each group with A, B, and C respectively, then the binary number X can be expressed as $X = A \cdot 2^{2n} + B \cdot 2^n + C$. The residue values of the three moduli can then calculated as follows:

$$
\begin{aligned}
x_1 &= |A \cdot 2^{2n} + B \cdot 2^n + C|_{2^n-1} \\
&= |A \cdot (2^n-1)(2^n+1) + A + B \cdot (2^n-1) + B + C|_{2^n-1} \\
&= |A + B + C|_{2^n-1} \qquad (5.1) \\
x_2 &= |A \cdot 2^{2n} + B \cdot 2^n + C|_{2^n} = C \qquad (5.2) \\
x_3 &= |A \cdot 2^{2n} + B \cdot 2^n + C|_{2^n+1} \\
&= |A \cdot (2^n-1)(2^n+1) + A + B \cdot (2^n+1) - B + C|_{2^n+1} \\
&= |A - B + C|_{2^n+1} \qquad (5.3)
\end{aligned}
$$

Therefore, the forward converter of this moduli set can be easily implemented through modular adders. For all the FPGA designs in this chapter and our RNS arithmetic library, we use ASC [Men06] as the hardware compiler that maps the design into FPGA implementations. In our experiments, all the designs are targeted on the Xilinx Virtex IV FX100 FPGA board. Table 5.1 shows the area cost and latency of RNS forward converters with typical bit-width settings.

Table 5.1: RNS Forward Converters: area cost and latency for different n values (the moduli set we use is $\{2^n - 1, 2^n, 2^n + 1\}$).

value of n	4	8	12	16	20
area cost / # slices	113	186	236	318	392
latency / ns	14.2	17.2	18.7	19.7	22.4

5.2.3 Reverse Converter

As mentioned in Section 2.4.2, the previous reverse converter designs are based on the CRT or the new CRT by Y. Wang [Wan98], and usually involve multiplications or at least 2n-bit modular additions. In our design, we try to acquire the binary value with only n-bit adders.

We describe our algorithm using the same A, B, C denotations as in Section 5.2.2. As A, B, C are all values in the range of $[0, 2^n - 1]$, the equations for forward conversion ((5.1), (5.2), (5.3)) can be converted as follows (as noted in Section 2.4.2, we denote three moduli values as M_1, M_2, and M_3):

$$A + B + C = x1 + c_1 \cdot M_1, \quad \text{where} \quad c_1 \in \{0, 1, 2\} \tag{5.4}$$

$$C = x2 \tag{5.5}$$

$$A - B + C = x3 + c_2 \cdot M_3, \quad \text{where} \quad c_2 \in \{-1, 0, 1\} \tag{5.6}$$

Altogether, c_1 and c_2 have nine different combinations. However, given one set of residue values $\{x_1, x_2, x_3\}$, there is always only one valid combination of c_1 and c_2, as the above equations include hidden constraints among x_1, x_2, x_3 and c_1, c_2 values. The first kind of constraints is about the parity of the numbers. As $A + B + C$ and $A - B + C$ have the same parity, if x_1 and x_3 also have the same parity, then c_1 and c_2 shall have the same parity. Otherwise, if x_1 and x_3 have different parities, then c_1 and c_2 shall have different parities. The second kind of constraints relate to the range of the values. We can derive the expressions of $2A$ and $2B$ from the above equations (5.4), (5.5), and (5.6). As $2A$ and $2B$ shall both be in the range of

Table 5.2: Conditions for selecting valid c_1 and c_2 values. For each different c_1 or c_2 value, the multiple rows on the right describes the corresponding cases that the value is valid, i.e. if the x_i values satisfy any of the rows on the right, c_1 or c_2 are determined to be the value on the left. 'SAME' and 'DIFF' denote that x_1 and x_3 have the same or different parity.

$c_1 = 0$	$x_1 \geq x_3$ & $x_1 + x_3 \geq 2 \cdot x_2$ & SAME
	$x_1 + x_3 \geq 2 \cdot x_2 + M_3$ & DIFF
$c_1 = 2$	$x_1 < x_3$ & $x_1 + x_3 \leq 2 \cdot x_2$ & SAME
	$x1 + x3 \leq 2 \cdot x_2 - M_3$ & DIFF
$c_1 = 1$	all other cases
$c_2 = -1$	$x_1 < x_3$ & $x_1 + x_3 > 2 \cdot x_2$ & SAME
	$x_1 + x_3 \geq 2 \cdot x_2 + M_3$ & DIFF
$c_2 = 1$	$x_1 \geq x_3$ & $x1 + x3 > 2 \cdot x_2$ & SAME
	$x1 + x3 \leq 2 \cdot x_2 - M_3$ & DIFF
$c_2 = 0$	all other cases

$[0, 2^{n+1} - 2]$, we can derive another set of equations regarding the relationship between x_1, x_2, x_3 and c_1, c_2.

Applying all the above hidden constraints in the equations (5.4), (5.5), and (5.6), we can acquire a full set of complete conditions that correspond to different combinations of c_1 and c_2 values, as shown in Table 5.2.

Applying all these conditions, we can determine the values of c_1 and c_2 according to residue values $\{x_1, x_2, x_3\}$. As C equals to the value of x_2, we can then calculate the values A and B from the values of $A + B + C$ and $A - B + C$ easily.

As shown in Figure 5.1, we implement the above reverse conversion algorithm as a circuit structure with two major parts. The first part is the condition decoder that takes the residue values $\{x_1, x_2, x_3\}$ as input, compare all the different value combinations as shown in Table 5.2, and figure out the correct coefficients c_1 and c_2. The second part is the binary generator that uses the coefficients c_1 and c_2 to construct the A and B parts. As part C comes directly from the value of x_2, we can combine them to produce the binary result.

To evaluate the cost and performance, we implement the reverse converter design on FPGAs with different n values (n being the parameter of the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$), and compare the experimental results with the design by Wang et al. [WSAS02], which is one of

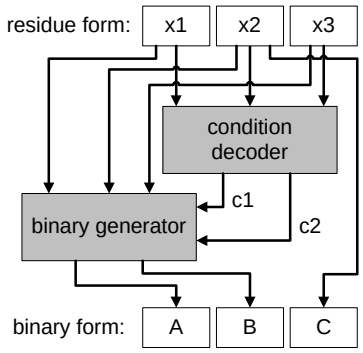


Figure 5.1: General structure of our reverse converter.

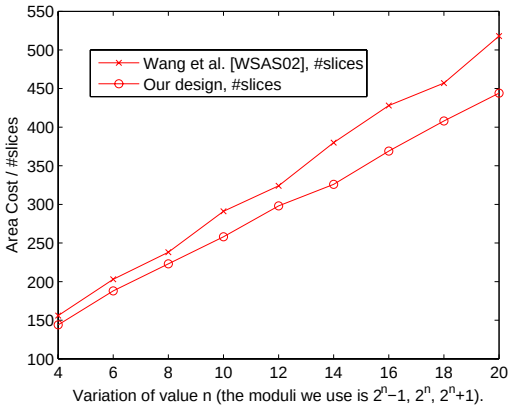
the most efficient existing designs for the $\{2^n - 1, 2^n, 2^n + 1\}$ moduli set. As shown in Fig. 5.2, compared to Wang's design, our reverse converters consume 7.7% to 14.3% less area on FPGAs. Our design consumes 7.7% less area for the case of $n = 4$, and consumes 14.3% less area for the case of $n = 20$. Thus, our reduction in area cost increases with the size of the moduli set. On the latency side, our design also takes less time to produce the output from the input values in most cases.

5.2.4 Scaling

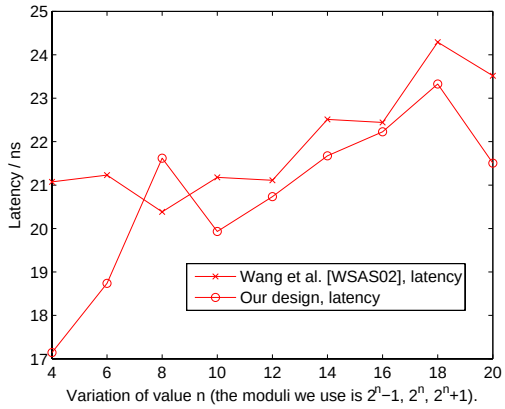
As mentioned in Section 2.4.2, scaling is an important operation to keep accumulated values or multiplication results within the representation range of RNS. Besides, if we want to use RNS numbers to represent fractional values, scaling is needed to keep the decimal point at the correct position.

In our library, we target at scaling by power of two values, i.e., shifting to the right in the binary representation. The algorithms based on multiplying modular inverse of scaling factors [GL98, DMST08] are not applicable, as the modular inverse of two does not exist for the modular 2^n. Instead, we try to achieve the scaling through forward and reverse converters.

If we want to scale by any power of two values, we convert the residue form into binary form first, shift by the corresponding number of bits, and convert the binary form back into the residue form. In this way, the cost of a scaling unit amounts to the sum of a reverse converter, a shifter and a forward converter.



(a) Comparison of area cost.



(b) Comparison of latency.

Figure 5.2: Our reverse converter versus the design by Y. Wang et al. [WSAS02]: comparison of area cost and latency.

On the other hand, if the scaling factor is a specific value, such as 2^n, the scaling operation can be greatly simplified. Assume the binary form of the original residue value x_1, x_2, x_3 is $X = A \cdot 2^{2n} + B \cdot 2^n + C$, then after scaling by a factor of 2^n, the result is $X' = A \cdot 2^n + B$. Thus, we can derive the residue representation of the scaling result as follows:

$$\begin{aligned} x'_1 &= |A \cdot 2^n + B|_{2^n-1} = |A + B|_{2^n-1} \\ &= \big||A + B + C|_{2^n-1} - C\big|_{2^n-1} = |x_1 - x_2|_{2^n-1} \qquad (5.7) \\ x'_2 &= |A \cdot 2^n + B|_{2^n} = B \qquad (5.8) \\ x'_3 &= |A \cdot 2^n + B|_{2^n+1} = |-A + B|_{2^n+1} \\ &= \big|C - |A - B + C|_{2^n+1}\big|_{2^n+1} = |x_2 - x_3|_{2^n+1} \qquad (5.9) \end{aligned}$$

To implement this scaling operation, we only need two modular subtractions and a part of the reverse converter to produce the value of B.

Table 5.3 shows the area cost and latency of the above two different scaling units. The 'normal scaling' unit can scale by any power-of-two value, while the 'specific scaling' unit only scales by 2^n, which is one of the moduli values. In most cases, the 'specific scaling' units only consume

Table 5.3: Two different RNS scaling units: the area cost and latency for different n values (the moduli set we use is $\{2^n - 1, 2^n, 2^n + 1\}$). 'normal scaling' rows scale by any power of two values. 'specific scaling' rows scale by one of the moduli values, 2^n.

value of n		4	8	12	16	20
normal scaling	area / # slices	238	391	553	705	854
	latency / ns	31.5	32.7	35.5	40.7	40.2
specific scaling	area / # slices	146	220	292	354	422
	latency / ns	15.3	17.125	17.9	16.3	19.5

around half of the area cost of the 'normal scaling' units. The latency of 'specific scaling' units is also reduced to about half of 'normal scaling' units.

5.2.5 Magnitude Comparison

Similar to the scaling operation, one straightforward method for magnitude comparison is to convert the residue form into binary form and perform the comparison afterwards. In this way, the area cost of a comparison unit equals two reverse converters and a comparator.

However, there are also special cases where we can simplify the comparison unit. A simple example is, if we know in advance that the binary forms of the two values only have difference in the last n bits, i.e. the parts A and B are the same and only part C is different, then we only need to compare the residue value x_2. In this special case, the comparison cost is similar to an n-bit adder.

5.2.6 Addition and Multiplication

RNS's advantage in addition and multiplication comes from the following property:

$$|a + b|_M = \big|\, |a|_M + |b|_M \,\big|_M \tag{5.10}$$

$$|a * b|_M = \big|\, |a|_M * |b|_M \,\big|_M \tag{5.11}$$

Thus, to add or multiply two RNS numbers $\{x_1, x_2, x_3\}$ and $\{y_1, y_2, y_3\}$, we only need to add or multiply the corresponding value pairs x_i and y_i. When implementing the adders and multipliers, we can apply the same techniques for binary adders and multipliers, such as carry-save adders and Booth's multiplication algorithms [FO01]. However, as our arithmetic library targets FPGA platforms, we use the adder and multiplier cores provided by ASC [Men06] directly.

Table 5.4 compares the area cost and latency of our RNS addition and multiplication units with the design by J. Beuchat [Beu03]. J. Beuchat's work provides modular adders and multipliers for the moduli of 2^n-1 and 2^n+1, and does not directly provide RNS arithmetic units. However, as mentioned above, an RNS adder or multiplier is three modular adders or multipliers putting together. Therefore, to get an area estimation of an RNS adder, we add the area of the modular adders for the moduli of $2^n - 1$, $2^n + 1$, and 2^n (J. Beuchat [Beu03] provides the results for modulo $2^n - 1$ and modulo $2^n + 1$ adders, and we estimate the area for a modulo 2^n adder to be n slices). For the area of an RNS multiplier, we multiply the area cost of the modulo $2^n + 1$ multiplier in [Beu03] by three times to get a rough estimation. For the latency estimation, we take the result of modulo $2^n + 1$ adder and multiplier as it has the longest carry chain of the three. For our units, different from the results shown in Section 5.3, the area results here do not include the interface circuits and streaming controller logics.

As shown in Table 5.4, our RNS units consume more area than the optimised designs in [Beu03], while providing a much smaller latency. The area disadvantage of our RNS units is mainly because they are generated using the default addition and comparison units of ASC, and there is no optimisation for the circuit architecture of the design. One of the our future plans is to further optimise the circuit architecture of our RNS addition and multiplication units, so as to achieve a better optimised RNS arithmetic library generator.

Table 5.4: Our RNS addition and multiplication units versus the design by J. Beuchat [Beu03]: comparison of area cost and latency. The RNS units are using the moduli set of $\{2^n-1, 2^n, 2^n+1\}$.

value of n	RNS Addition Units				RNS Multiplication Units			
	area/slices		latency/ns		area/slices		latency/ns	
	ours	[Beu03]	ours	[Beu03]	ours	[Beu03]	ours	[Beu03]
4	40	15	5.6	8.4	81	57	10.7	16.1
8	69	27	6.3	11.6	205	159	15.2	21.8
12	94	39	7.1	11.6	376	333	17.0	27.6
16	108	51	7.2	13.4	614	546	20.8	30.0
20	151	63	7.6	14.8	909	810	22.7	34.5

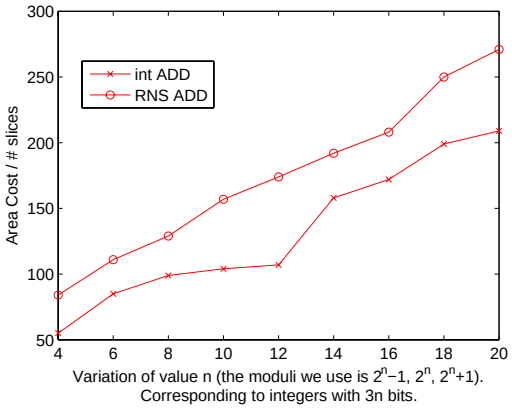
5.3 Comparing RNS and Integer Arithmetic Units

The previous section describes the algorithm design of the major units in our RNS arithmetic library. By implementing the designs using ASC syntax descriptions, we develop a generator for RNS arithmetic libraries. The generator provides arithmetic units for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, and takes n as an input parameter that users can configure.

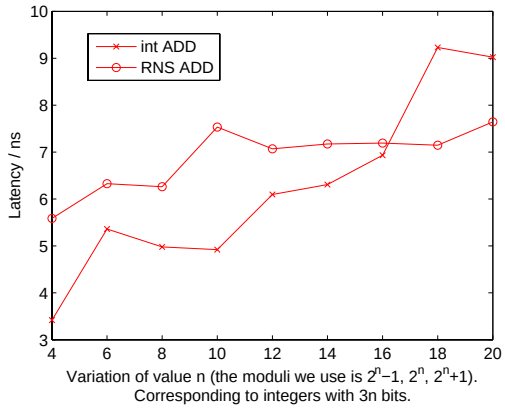
Combining this library generator and ASC, we have a tool that can automatically generate RNS arithmetic units for different bit widths, and evaluate the area costs and latencies of the units. To compare RNS and common binary representations, we perform a comparison between RNS and integer arithmetic units with different bit-width settings.

Figure 5.3 shows the comparison between RNS and integer adders. We compare RNS operations for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ to 3n-bit integer operations, as they have a similar representation range.

The area costs of both RNS and integer adders increase almost linearly to the bit widths of operands. The RNS adders consume around 50 slices more than the integer adders. The integer adder performs a 3n-bit addition, while the RNS adder performs three n-bit additions separately. However, the RNS adder also needs to perform a modular operation after the addition, which incurs more area consumption. On the latency side, the RNS adders also do not show the expected advantage. This is mainly because the adders are implemented on the FPGA with fast-carry chains, which greatly reduce the latency difference between the carry



(a) Comparison of area cost.



(b) Comparison of latency.

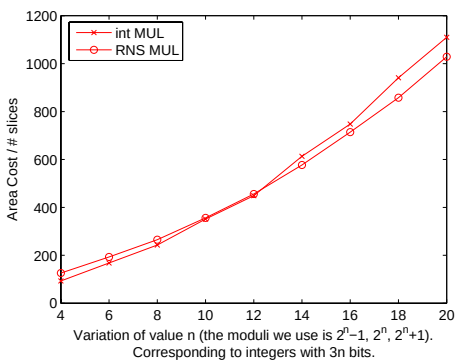
Figure 5.3: RNS adders versus integer adders: comparison of area cost and latency.

chain of a 3n-bit adder and the carry chain of a n-bit adder. Meanwhile, the extra modular step of RNS adders introduces more latency. Thus, in most cases, the RNS adders show a higher latency than integer adders. Only when the value n increases to 18, which corresponds to integers over 54 bits, the short carry chain of RNS starts to outweigh the extra latency of the modular step, and the RNS adders show a lower latency than the integer adders.

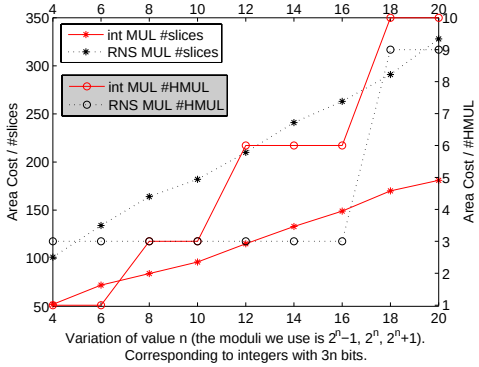
As RNS multiplier is quite different from integer multiplier, it is not immediately clear how to perform an unbiased comparison of the multipliers of the two different number systems. RNS multiplications are bounded by the range of the moduli, i.e. the multiplication of two $3n$-bit values only produce a $3n$-bit result. For integers, the multiplication of two $3n$-bit values produce a $6n$-bit result. In our comparison, we try to make the computation complexity in the two number systems as equivalent as possible. Thus, we reduce the the multiplication of two $3n$-bit integers into a partial multiplication that only produces the lower $3n$ bits as the result.

Figure 5.4 shows the comparison between RNS and integer multipliers. The multipliers can be implemented using only logic slices, or using both logic slices and HMULs.

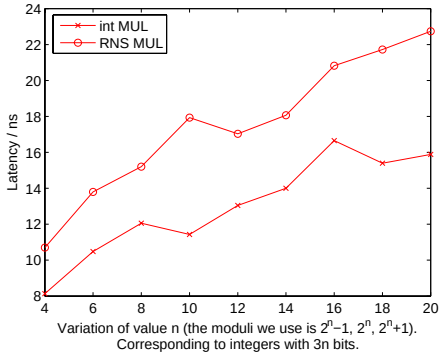
When using only logic slices, the area costs for integer and RNS multipliers are similar. For bit widths smaller than 10, RNS multiplier consumes more area due to the extra modular operations to the results. For bit widths larger than 10, RNS multipliers start to bring area reductions. However, the latency of RNS multipliers are higher than integer multipliers, which



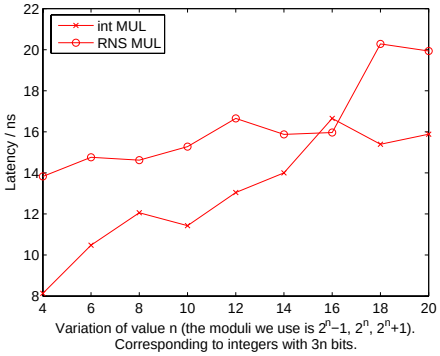
(a) Comparison of area cost, using slices only.



(b) Comparison of area cost, using slices and HMULs.



(c) Comparison of latency, using slices only.



(d) Comparison of latency, using slices and HMULs.

Figure 5.4: RNS multipliers versus integer multipliers: comparison of area cost and latency. The RNS multipliers can save up to 100 slices or 50% HMULs for large bit-width settings.

is also because of the modular operation needed after the multiplication.

When using both logic slices and HMULs, the RNS multipliers consume 50 to 200 more slices than integer multipliers, but the number of HMULs is reduced greatly. For large bit widths ($n = 12, 14, 16$), the RNS multipliers only consume 1/2 of the number of HMULs consumed by integer multipliers. The latency of RNS multipliers is still higher than integer multipliers.

The comparison results show that the major advantage of RNS on current FPGAs lies in the reduction of resource consumptions for multipliers. Because of the extra modular operations after addition, RNS adders consume slightly more area than integer adders. For similar reasons, RNS adders and multipliers also show a higher latency than integer units. For large bit widths, RNS adders start to produce a similar or even lower latency.

5.4 Summary

Our work optimises RNS arithmetic designs on the platform of reconfigurable devices, such as FPGAs. In particular, we proposed a novel design for reverse converters from RNS to binary numbers, which consumes up to 14.3% less area and provides lower latency. Based on the optimised RNS arithmetic units, we developed an RNS arithmetic library generator for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. The generator takes the value n as a configurable parameter and supports a wide range of RNS numbers. This library generator enables us to perform an extensive comparison between RNS and other number representations at both the arithmetic unit level and the application level. The comparison showed that, the major advantage of RNS on current FPGAs lies in reduced resources in multiplication operations.

Chapter 6

R-Tool: Bit-width Optimisation across Multiple Number Representations

6.1 Introduction

This chapter presents an optimisation tool to deduce variable bit widths automatically from a given high-level description of an algorithm. We call this tool R-Tool, as it works across multiple number representation systems, including fixed point, floating point and LNS. R-Tool does not support RNS. As the inherent mechanism of RNS requires the operands of arithmetic operations to have the same moduli, i.e. the same bit widths, it is very inefficient to have non-uniform bit widths in an RNS design.

The major contributions of R-Tool include:

- support of both static and dynamic analysis techniques to provide different optimised bit widths for different guarantees of accuracy;
- bit-width analysis for multiple number representations, such as fixed-point, floating-point and the logarithmic number system;

- practical FPGA (Field Programmable Gate Array) implementations to demonstrate the proposed techniques and to compare between dynamic and static approaches.

The current tool focuses on minimising the area usage of the designs on reconfigurable hardware devices, such as FPGAs. However, by changing the cost functions we use in the optimisation process, the same tool can be adapted to minimise or maximise other metrics. For instance, we can build cost functions that estimate the power consumption for designs with different bit-width attributes, and apply the tool to figure out the optimised design with minimum power consumption [GCC06].

The rest of the chapter is organised as follows. Section 6.2 presents an overview of our optimisation tool, R-Tool. Section 6.3 describes the basic steps to perform bit-width optimisation for a design. Section 6.4 discusses precision analysis based on automatic differentiation. Section 6.5 introduces our support for different number representations. Section 6.6 provides concluding remarks.

6.2 Overview

Figure 6.1 shows the conceptual algorithm flow of R-Tool. The front-end of R-Tool takes a high-level C++ design description in ASC format as input. The R-Tool library redefines ASC hardware data types and overloads their arithmetic operators; it provides a straight-forward interface to analyse C/C++ programs with little manual change of the code.

In the first step, R-Tool maps the high-level algorithm description into an annotated dataflow graph, which forms the basis of the range and precision analysis afterwards.

The second step is to perform range analysis for all the declared and temporal variables through the dataflow graph. Using affine arithmetic [ACS94] or dynamic simulation of input values, we calculate the maximum ranges for the intermediate and output variables. With the calculated ranges, we can determine the range bit width, i.e. bit width that relates to the range of the

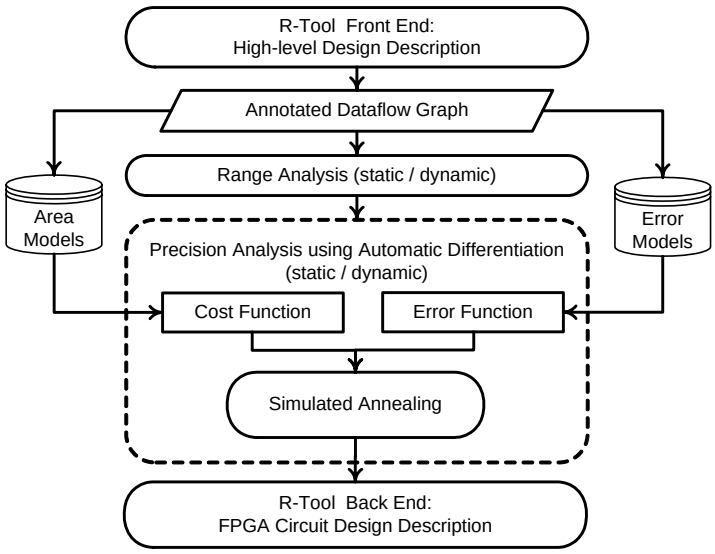


Figure 6.1: Conceptual algorithm flow of the bit-width optimisation framework, R-Tool. At both range and precision analysis stages, users have the option to employ static or dynamic approaches.

representation, such as the exponent bits of floating-point and integer bits of fixed-point and logarithmic numbers.

The third step is precision analysis, the major problem to tackle in R-Tool. We use the static affine arithmetic error model [LGC+06] or the dynamic automatic differentiation approach to analyse the dataflow graph and produce the error and cost functions for the design. The error and cost functions calculate the total error of the output and the hardware cost of the entire design according to the number representation format and bit widths of the variables. These two functions are then fed into the adaptive simulated annealing (ASA) engine [Ing04] to minimise the cost function while keeping the error function within the required values. In this step, we determine the precision bit widths, i.e. bit widths that are precision-related, such as the mantissa bits of floating-point and fractional bits of fixed-point and logarithmic numbers.

With the optimised bit widths from the ASA engine, R-Tool outputs the optimised design in ASC format. The description is then forwarded to ASC for either testing with a software simulator or mapping into hardware on FPGAs.

6.3 Bit-width Optimisation Using R-Tool

This section discusses our approach for bit-width optimisation. The optimisation process consists of three major steps. In the first step, the high-level algorithm description is mapped into a dataflow graph; in the second and third step, the range and precision analyses are performed respectively.

6.3.1 From C++ Code to Dataflow Graph

To handle the input ASC format C++ algorithm description, the R-Tool framework contains a library that redefines ASC-style hardware data types (e.g. HWint, HWfix, HWfloat) and overloads their arithmetic operators. Thus, to put a piece of C++ code under the bit-width analysis of R-Tool, we only need to modify the data types to the corresponding ASC-style hardware data types.

As shown in Figure 6.2, we compile the degree-two polynomial design description with hardware data type HWfloat (for floating-point representations), and map the description into a data flow graph. In the dataflow graph, the arithmetic operations are denoted as nodes, while the input and output variables of each operation are denoted as edges. The mapping process from C++ code to dataflow graph consists of two steps. The first step happens at the declaration of the variables, where the R-Tool library creates corresponding edges for each variable. The second step relates to the computational part, where R-Tool library calls overloaded operators to create the nodes for each arithmetic operation and to set up the connection between nodes and input and output edges. Note that there are two edges annotated with '`x`' in Figure 6.2, while in the actual data structure, it is one single edge that connects to two different arithmetic nodes. Generally, in our dataflow graph, each edge can be inputs to multiple nodes, but can only be output to one single node. To facilitate the later analysis process, each edge keeps record of its source node and destination node, while each node also keeps record of its input and output edges.

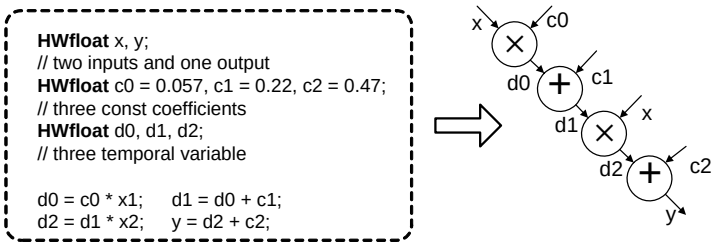


Figure 6.2: An example that maps C++ description with ASC-style hardware data types into a dataflow graph.

6.3.2 Range Analysis

Range analysis is concerned with determining of the minimum number of range bits to prevent data overflow or underflow. In recent bit-width analysis work, both static and dynamic methods are employed to estimate the ranges of variables [BGWS00, MRS+01, Raz94, SBA00, KS01, KKS00, FRC03].

Static analysis techniques [BGWS00, MRS+01, Raz94, SBA00, FRC03, LGC+06] operate on a flow graph description of the designs, by propagating range information from the inputs to the outputs. A common technique for propagating range information involves interval arithmetic [Moo66]. However as mentioned in Section 2.5, interval arithmetic suffers from the "correlation problem". Consider the function $f(x) = x - x$ with x in the range of [0,1], the rules of interval arithmetic give the range of $f(x)$ as $[-1, 1]$ and not 0 as expected. One proposed solution to this problem is to use affine arithmetic[ACS94], which not only captures the existing features of interval arithmetic but also provides the ability to keep track of the correlations between variables.

However, even though affine arithmetic keeps track of correlations between variables, static arithmetic still usually leads to a conservative estimation of ranges. Dynamic analysis [KS01, KKS00, GMLC04] overcomes this problem by simulation with actual data values. This gives the same ability as affine arithmetic to capture the correlation between the variables, while providing a realistic estimation of the required ranges. The main drawback with dynamic range analysis is that the results of the analysis are dependent upon the input data used in the

simulation, which may or may not be fully representative.

In the R-Tool framework, we provide both the static and dynamic analysis approaches in the range analysis stage. We use affine arithmetic as the static approach, to propagate the range information through the entire dataflow graph.

In affine arithmetic, the range of a variable x is expressed in affine form as:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n \tag{6.1}$$

where x_0 is the mid-point value of the variable, ε_i is a random value distributed in the range $[-1, 1]$, and $\sum_{i=1}^{n} x_i\epsilon_i$ denotes the deviations from the mid-point value. An input range of $[x_{min}, x_{max}]$ can be written as:

$$\hat{x} = \frac{x_{min} + x_{max}}{2} + \varepsilon\frac{x_{max} - x_{min}}{2} \tag{6.2}$$

Given the affine values of the input variables, we calculate the affine values of all the internal variables through the dataflow graph, using affine arithmetic rules. Addition and subtraction are straightforward:

$$\hat{x} \pm \hat{y} = x_0 \pm y_0 + \sum_{i=1}^{n}(x_i \pm y_i)\varepsilon_i \tag{6.3}$$

The following equation shows the case for multiplication.

$$\begin{aligned}
\hat{x}\hat{y} &= x_0y_0 + \sum_{i=1}^{n}(x_0y_i + x_iy_0)\varepsilon_i + (\sum_{i=1}^{n} x_i\varepsilon_i)(\sum_{i=1}^{n} y_i\varepsilon_i) \\
&\approx x_0y_0 + \sum_{i=1}^{n}(x_0y_i + x_iy_0)\varepsilon_i + (\sum_{i=1}^{n} |x_i|)(\sum_{i=1}^{n} |y_i|)\varepsilon_{n+1} \\
&= x_0y_0 + \sum_{i=1}^{n+1} u_i\varepsilon_i
\end{aligned} \tag{6.4}$$

$$\text{where} \quad u_i = \begin{cases} x_0 y_i + x_i y_0 & , \quad \text{if} \quad i \leq n \\ (\sum_{i=1}^{n} |x_i|)(\sum_{i=1}^{n} |y_i|), & \text{if} \quad i = n+1 \end{cases}$$

Some operations, such as division and square root, are difficult to study in the form of affine arithmetic. These operations are processed using min-range affine approximations [SdF97]. In this way, much of the correlation between different results can be kept, but there is still loss of information during approximation. After a complete pass from input to output, we can acquire the ranges for all the variables.

R-Tool also provides a simulation-based dynamic approach for range analysis. We read input values from a test file, perform the value calculations along the dataflow graph, and record the maximum/minimum values for each variable to calculate the range from the experiment data.

After computing the range information for each variable, we can then calculate the minimum range bit widths according to the characteristics of different number representations. A discussion about how to calculate range bit widths from ranges is given in Section 6.5.1.

6.3.3 Precision Analysis

In precision analysis, the objective is to find a set of bit widths that not only meet a pre-defined error requirement but also minimise a cost function which reflects one or more of the design metrics such as area, latency or power consumption.

The effect of the limited precision of the internal variables manifests in the output as an error or as a deviation from the ideal value:

$$E_{out} = f_{error}(W_1 \cdots W_n) \tag{6.5}$$

where E_{out} is the error at the output of the circuit and $W_1 \cdots W_n$ are the precision bit widths of the variables inside the circuit. The precision requirement is to ensure that $E_{out} \leq E_{req}$, where E_{req} is the required error at the output.

In our R-Tool framework, this error equation is based on the dataflow graph of the design, using either static or dynamic analysis method. For the static method, we use the affine arithmetic error model in a previous work [LGC+06] to express the error introduced into each variable due to quantisation and arithmetic calculation, and propagate the errors into the output. For the dynamic method, we use an automatic differentiation approach [GMLC04] to analyse the dataflow graph to find the sensitivity of the output to the precision bit widths of the input and intermediate variables. The approach is discussed further in Section 6.4.

Supporting both static and dynamic approaches, R-Tool offers the users the option to generate optimistic error functions based on dynamic simulations, or conservative error functions based on affine arithmetic error models.

The cost C that we aim to minimise can be expressed as a function of the precision bit widths $W_1 \cdots W_n$ of the variables:

$$C = f_{cost}(W_1 \cdots W_n) \tag{6.6}$$

In R-Tool, the optimisations are generally performed to minimise area. The cost function is formulated with investigation into the characteristics of different number representation systems, using either analytical or empirical method. A more detailed discussion about area modelling is presented in Section 6.5.3.

While keeping the E_{out} in equation (6.5) below the required value, the minimisation of C in equation (6.6) becomes the objective of the optimisation process. The solution space grows exponentially with the number of variables being optimised. In [CW02] this is shown as an NP-hard problem.

One feasible solution is to find the minimum value of $W_1 \cdots W_n$ with the condition that $W_1 = \cdots = W_n$. This solution is known as the uniform bit-width solution. However, the uniform bit-width solution is usually not optimal, as it fails to account for different costs for different parts in the design. For example, in a circuit consisting of adders and multipliers, reducing the

bit widths of the multipliers would provide greater area savings than reducing the bit widths of the adders.

In R-Tool, we feed the error and cost function into the ASA engine [Ing04], and use the simulated annealing algorithm to produce results that can optimise the target metric to some extent under the constraint of the problem. Based on the results in previous work [LGC+06], the difference between the results given by ASA engine and the optimum results computed through integer linear programming (ILP) is within 1%, while the ASA engine produces optimisation results much faster than ILP tools. Meanwhile, when applying the ASA engine, we use the uniform bit-width solution as an initial guess of the simulated annealing algorithm, so that the algorithm can converge to the result more quickly.

6.3.4 Constraints of the Precision Analysis in R-Tool

In R-Tool, we need to derive an error function from the dataflow graph, using either the static affine arithmetic method or the dynamic automatic differentiation approach. This mechanism works fine for streaming designs, as we only need to iterate once over the dataflow graph to generate the error function. For designs that include loops with feedback, we need to iterate numerous times over the data flow graph. The derived error function can easily become meaningless due to the excessive propagation of errors in each step.

Another constraint is that we need a specific requirement (E_{req}) for the maximum error of the design. For numerical applications, there is generally a clear precision requirement for the results. However, for applications that produce images or patterns, there is no clear definition for the accuracy of the results.

Therefore, R-Tool works best for streaming circuits with a determined error requirement of the results. For designs that involve numerous iterations and do not have a clear error requirement, other techniques are needed to perform the bit-width analysis. One example is our work on bit-width analysis for seismic imaging algorithms shown in Chapter 8. In this case, the imaging computation applies thousands of processing iterations on huge volumes of data, and

Table 6.1: Differentiation rules for basic arithmetic operations.

operation $z = f(x, y)$	$\frac{\partial z}{\partial x}$	$\frac{\partial z}{\partial y}$
$z = x + y$	1	1
$z = x - y$	1	-1
$z = xy$	y	x
$z = x/y$	$1/y$	$-x/y^2$

the generated result is an image that estimates the underlying earth model. To perform the precision analysis, we run a bit-accurate simulation to explore different bit widths, and figure out the minimum bit widths that still generate accurate enough seismic images.

6.4 Automatic Differentiation Algorithm

This section introduces the dynamic precision analysis technique based on automatic differentiation, the basic idea of which comes from A. Gaffar's work on "BitSize" [GMLC04]. We first describe the basic theory of automatic differentiation; then illustrate our error model based on automatic differentiation and the related workflow to derive the error in the output from the errors in internal variable.

6.4.1 Automatic Differentiation

Automatic differentiation [GF91, GMLC04] is a method developed by the applied mathematics community for the differentiation of algorithms. The main advantage of automatic differentiation is the ability to calculate the differentials as a side effect of the execution of the user algorithm, with few changes to the algorithm itself.

R-Tool performs automatic differentiation from input to output of the dataflow graph in a forward accumulation manner. At the beginning of the process, the gradients of the input variables are assigned to one. After that, we propagate the differentiation information forward through the operator nodes until the output is reached.

As shown in Table 6.1, for the basic arithmetic operation nodes ($+, -, \times, \div$), we employ simple

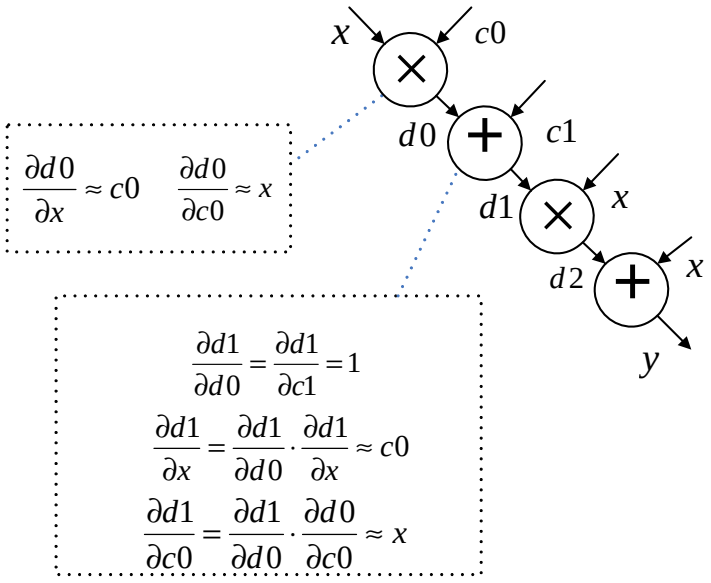


Figure 6.3: An example of the automatic differentiation workflow.

rules to calculate the gradients of output to input variables. Meanwhile, for a composition $f(x) = g(h(x))$, the chain rule gives:

$$\frac{df}{dx} = \frac{dg}{dh}\frac{dh}{dx} \tag{6.7}$$

Thus, for variables not directly related to the same arithmetic operation, the differentiation chain rule is utilised to build up the correlation.

At each node, we update the differentiation information of the output edge in two steps: first, calculate the gradient of the output edge with respect to its direct input edges; second, calculate the gradient for the indirect input edges by propagating the gradients of the direct input edges to indirect input edges.

Consider a degree-two polynomial in Figure 6.3. At the first multiplication node, we calculate the gradient of output edge $d0$ for input edges x and $c0$. Although $c0$ is a constant coefficient, we still process the gradient of $d0$ for $c0$ to account for the error introduced due to the finite precision of $c0$. At the second operation, the addition node, we calculate the gradient of output edge $d1$ for input edges $d0$ and $c1$. Because it is an addition, the gradient is simply one. Meanwhile, we propagate the gradients of $d0$ ($\frac{\partial d0}{\partial x}$, $\frac{\partial d0}{\partial c0}$) for $d1$ to find the correlation between error of $d1$ and errors of x and $c0$. Utilising differentiation rules from input to output step by

step, we can find out the gradients of the output for all the input variables and intermediate variables.

6.4.2 Error Model Based on Automatic Differentiation

As shown in A. Gaffar's work on "BitSize" [GMLC04], for a function $y = f(x)$ with single input x, the sensitivity relationship between the output and the input is defined as follows:

Definition 6.1 (Sensitivity) *Provided $y = f(x)$ is a differentiable function, the sensitivity s of an output y is a function of the input x, such that a change Δx in the input x causes a change Δy in the output y: $\Delta y = s(x)\Delta x \approx f'(x)\Delta x$*

In the above definition, we only consider the first order derivative item. As the equation is used to derive errors, and the variable Δx is generally very small, the second order and other higher order items become negligible.

A. Gaffar extends the definition of sensitivity, by considering a node with n inputs $x_0 \cdots x_n$, and output y. The inputs are related to the output by the differentiable function f:

$$y = f(x_0, x_1, \cdots, x_n) \tag{6.8}$$

Let Δx_i be the absolute error introduced when x_i is represented in finite precision. Its value is given by:

$$\Delta x_i = |\bar{x_i} - x_i| \tag{6.9}$$

where $\bar{x_i}$ is the value of x_i in finite precision.

Let Δy be the effect on y in response to the changes in $x_0, x_1, \cdots, x_n$. Then it can be expressed using the Taylor approximation:

$$\Delta y \approx \Delta x_1 \frac{\partial y}{\partial x_1} + \cdots + \Delta x_n \frac{\partial y}{\partial x_n} \tag{6.10}$$

where $\frac{\partial y}{\partial x_i}$ is the sensitivity or gradient of y, to changes in x_i. The higher order terms in the Taylor approximation are ignored, under the assumption that their contribution to the accuracy is negligible. This equation gives an accurate estimation of the correlation between the total error in the output and the errors of internal variables.

Based on the above error model in A. Gaffar's work, for a simple arithmetic operation node ($c = f(a, b)$), we derive the error in output c as follows:

$$E(c) = E(c)_r + E(c)_p = E(c)_r + E(a)\frac{\partial c}{\partial a} + E(b)\frac{\partial c}{\partial b} \tag{6.11}$$

where $E(c)_r$ stands for the error introduced in this arithmetic step and $E(c)_p$ stands for the error propagated from previous steps.

Assuming that a and b are accurate input values with no errors, there is still error in the output c due to its finite bit width and the hardware implementation of the arithmetic operation. This part of error mainly relates to the variable's representation and the corresponding arithmetic design, thus it is denoted as representation error $E(c)_r$. Meanwhile, in practical designs, a and b also have errors due to their finite precision or previous calculations; these errors are propagated into output c as the part $E(c)_p$, which is denoted as propagation error.

For a circuit design, such as the degree-two polynomial shown in Figure 6.3, if we keep on applying the above error calculation rule, the error of the final output y can be expressed as:

$$E(y) = E(y)_r + E(y)_p = E(y)_r + \sum_{i=1}^{n} E(x_i)_r \frac{\partial y}{\partial x_i} \tag{6.12}$$

where x_i ($i = 1..n$) stands for all the input and intermediate variables in the design. Given that the gradient of output y to itself can be regarded as 1, this equation shows that the total error is the sum of products of the representation error at each variable (including input, output and intermediate variables) and the gradient of the output to that variable.

Comparing Equation 6.12 and Equation 6.5, to achieve a complete error function, we still need to map the representation error parts ($E(x_i)_r$) into an equation of precision bit widths. This depends on different error models of different number systems, and is detailed in Section 6.5.2.

For the calculation of gradients in automatic differentiation, variables with non-deterministic values are involved. Consider the first multiplication node in Figure 6.3, where the gradient of $d0$ to $c0$ equals to x. In the later calculation of the error function, we need to assign certain values to the variable x. In our dynamic automatic differentiation approach, similar to the dynamic range analysis, we feed in a set of input values into the dataflow graph, and go through the entire graph repeatedly. During each run, the dynamic values are used to calculate the gradients. After the runs are finished, we select the maximum/minimum absolute values of gradients to form the error functions.

6.5 Support for multiple number representations

In this work, we assume that the entire design uses the same number system, i.e. no mixed use of different number formats.

The bit-width optimisation approaches discussed in the previous sections are independent of number representations. For example, we can employ the same automatic differentiation techniques to calculate the error propagation (the E_p part) for fixed-point, floating-point and LNS numbers. In this section, we discuss the parts that are dependent of number representations, such as the calculation of the representation error (E_r), the mapping from range, or area cost to bit widths, and vice versa. We demonstrate our solution for fixed-point, floating-point and LNS.

6.5.1 Calculation of Range Bit Widths

As mentioned in Section 6.3.2, using affine arithmetic or dynamic tests, we can find the ranges of all the variables in the design. In this part, we show our range bit width calculation function,

$RBW(\text{range}, \text{representation})$, which calculates the range bit width based on the range and the representation format.

A *fixed-point* number with m integer bits and f fractional bits has a representation range of $[-2^{m-1} + 2^{-f}, 2^{m-1} - 2^{-f}]$ (suppose it uses sign-magnitude sign mode). Thus, the range bit width m mainly relates to the maximum absolute value that it can represent. Let $[x_{min}, x_{max}]$ be the estimated range for variable x. The maximum logarithmic value lv is given by

$$lv = \log_2(\max(|x_{min}|, |x_{max}|))$$

The required integer bit width m for fixed-point representation is:

$$\begin{aligned} m &= RBW([x_{min}, x_{max}], \text{fixed-point}) \\ &= \begin{cases} \lfloor lv \rfloor + 2 & \text{if } lv \geq 0 \\ 1 & \text{if } lv < 0 \end{cases} \end{aligned}$$

For *floating-point* numbers with m exponent bits and f significand bits, the representation range is $\pm 2^{-2^{m-1}+2}$ to $\pm(2 - 2^{-f}) \times 2^{2^{m-1}-1}$ plus zero. Thus, m does not only relate to the maximum absolute value but also the minimum absolute value (excluding zero) that it can represent. If the range of the variable ($[x_{min}, x_{max}]$) does not include zero, the maximum absolute exponent value e_{max} is given by

$$e_{max} = \max(|\log_2(|x_{min}|)|, |\log_2(|x_{max}|)|)$$

The maximum logarithmic value of the exponent le is

$$le = \log_2(e_{max}) = \log_2(\max(|\log_2(|x_{min}|)|, |\log_2(|x_{max}|)|))$$

The required exponent bit width m is:

$$\begin{aligned} m &= RBW([x_{min}, x_{max}], \text{floating-point}) \\ &= \lfloor le \rfloor + 2 \end{aligned}$$

If the range of the variable covers zero, the minimum absolute value to represent becomes unknown. Using only maximum absolute value to calculate the exponent bit width brings possible risk of underflow. Although we still use the above equation to calculate an initial value of the exponent bit width, during the error analysis afterwards, we perform an adjustment procedure to ensure that the error caused by underflow does not exceed the error caused by the rounding/truncation of the last mantissa bit. In this way, the resulting exponent bit width may not be optimal, but we can be sure that the error requirement can still be met even in the case of underflow.

LNS representation produces a similar representation range to floating-point numbers. Especially for the exponent part of floating-point and the integer part of LNS numbers, they produce exactly the same range with the same bit width. Thus, the integer bit width m of LNS can be calculated using the same approach as floating-point.

$$\begin{aligned} m &= RBW([x_{min}, x_{max}], \text{LNS}) \\ &= RBW([x_{min}, x_{max}], \text{floating-point}) \end{aligned}$$

A similar adjustment procedure is needed if the range of the LNS variable includes zero.

6.5.2 Mapping from Precision Bit Widths to Error Functions

The automatic differentiation approach introduced in Section 6.4 finds the relationship between the total error of the output and the representation errors of all the variables. This section

discusses how we describe the representation errors of the variables based on their precision bit widths.

For *fixed-point* numbers, the mapping from precision bit width to error is quite straight-forward. Consider a variable with f fractional bits. The absolute error introduced by this variable can be expressed as follows:

$$Err_{fix}(f) = \begin{cases} 2^{-f-1} & \text{if round} \\ \\ 2^{-f} & \text{if truncate} \end{cases}$$

Depending on whether rounding-to-nearest or truncation is used, the fixed-point variable introduces 0.5 unit-in-last-place (ulp) or 1 ulp error into the design. A rounding mode, such as round-to-nearest, while giving better error bounds and a more balanced error distribution than truncation, would require additional hardware to implement. On the other hand, truncation would require one extra bit to provide the same error bound as round-to-nearest. In the current system, we use truncation for most of the intermediate steps to save the area cost. For the final result of an evaluation unit, we perform rounding to achieve a better accuracy.

For *floating-point* representation, the error does not only relate to the mantissa, but also the exponent. The absolute error for a variable with f mantissa bits is:

$$Err_{flt}(f) = \begin{cases} 2^{-f-1} \times 2^{M} & \text{if round} \\ \\ 2^{-f} \times 2^{M} & \text{if truncate} \end{cases}$$

where M is the exponent value of the variable. Similar to fixed-point representation, Err_{flt} also depends on the rounding mode used.

For *LNS* numbers, the representation error does not only relate to the bit width of the variable, but also relates to the regarding arithmetic operation. LNS multiplication and division are implemented in the form of fixed-point addition and subtraction. For these two operations, if

the result has the same fractional bit width as the operands, there is no error incurred by the arithmetic operation. Otherwise, depending on whether rounding-to-nearest or truncation is used, the LNS variable has 0.5 or 1 ulp representation error. LNS addition and subtraction are approximated rather than calculated directly, thus extra approximation errors are introduced into the arithmetic units. To keep the total error acceptable, in our LNS library [FML07], round-to-nearest is required at the last step to provide 1 ulp total error. In general, the absolute error for an LNS variable with f fractional bits is shown as follows:

$$Err_{LNS}(f, \times/\div) = \begin{cases} (2^{2^{-f-1}} - 1) \times 2^{M.F} & \text{if round} \\ \\ (2^{2^{-f}} - 1) \times 2^{M.F} & \text{if truncate} \end{cases}$$

$$Err_{LNS}(f, +/-) = (2^{2^{-f}} - 1) \times 2^{M.F}$$

where M.F is the value of the LNS number.

6.5.3 Area Modelling

We use area as the cost metric to optimise for the non-uniform bit-width optimisation. As the FPGA platform includes heterogeneous resources, such as logic slices, hardware multipliers (HMULs), and Block RAMs (BRAMs), we utilise the conversion model in [HBUH05] to convert different categories of resources into one single cost value. The model counts a BRAM as 27.9 slices and a HMUL as 17.9 slices.

The area cost functions used in the optimisation derives from an area model that is parameterised by the operator types and the variable bit widths. Generally, we express the area model for arithmetic units in an equation as follows:

$$\text{Area} = f(BW_{in1}, BW_{in2}, BW_{out}) \tag{6.13}$$

where BW_{in1} and BW_{in2} are the bit widths of the two input variables to the operator and BW_{out} is the bit width of the output variable of the operator.

This area model can be formulated using either analytical or empirical methods. In this work we employ the analytical technique for the fixed-point arithmetic area model, since the addition, subtraction and multiplication of fixed-point numbers are regular, and the resource cost can be captured as linear or quadratic functions of the bit widths.

However for arithmetic representations that have a more complex structure, the analytical method is not always feasible, especially when information about the structure of the arithmetic implementation library is unknown. Section 4.4.2 presents our area models for floating-point and LNS numbers, which are formulated using empirical methods.

Division area models for fixed-point and floating-point are not included in our current tool. There are many different algorithms and design options to implement division in hardware. However, if the implementation algorithm is fixed, we can similarly use the design exploration capability of ASC to generate division units of different bit widths, and capture the data points in an area model.

6.6 Summary

We presented a bit-width optimisation tool, R-Tool, which supports multiple number representations, such as fixed-point, floating-point and LNS numbers. Supporting both the dynamic approach using automatic differentiation and the static method using affine arithmetic, our tool provides various options to generate different optimised bit widths that meet requirements with different guarantees of accuracy. The static method tackles the problem conservatively and assures the accuracy over all possible inputs, while the dynamic approach can reduce resource costs by only guaranteeing accuracy for tested input values.

Part II

Effects of Application-Specific Number Representation

Chapter 7

Case Study:
Number Representation Exploration

7.1 Introduction

To demonstrate the advantages of application-specific number representations, this chapter shows case studies of exploring different number representations on practical applications or computation kernels. For the same application, the automatic design space exploration tool generates different designs using different number systems, and compares the results in terms of area cost, accuracy, and performance.

As mentioned in Section 2.2.4, using a given number of bits, floating-point and LNS numbers have a similar representation range and errors. Section 7.2 presents a comparison between floating-point and LNS numbers on practical designs. On the other hand, integer/fixed-point (integer and fixed-point arithmetic are equivalent on FPGA platforms) and RNS numbers have a similar representation range and accuracy. Section 7.3 compares RNS and integer/fixed-point numbers on practical applications.

The comparisons in Section 7.2 and Section 7.3 do not include bit-width optimisation, i.e. the variables are assumed to have uniform bit widths. Section 7.4 explores the target design with

fixed-point, floating-point, and LNS numbers, performs bit-width optimisation for the three number representations, and produces designs with optimised bit widths.

7.2 Comparing LNS and FLP Designs

In this section, we demonstrate our exploration tools on a number of applications or computation kernels. As practical FPGA applications are usually fully pipelined to achieve a high throughput, we only discuss the fully pipelined designs in this section.

7.2.1 Brief Description of Benchmarks

Digital Sine/Cosine waveform Generator (DSCG): a DSCG is a common tool in digital signal processing and communication applications [Mit05], which generates a sequence of discrete values that represent a sine or cosine wave. In our case study, we implement a simple generator as follows:

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} \cos\theta & \cos\theta + 1 \\ \cos\theta - 1 & \cos\theta \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix}. \tag{7.1}$$

Matrix Multiplication (MM): a fully-pipelined two-by-two matrix multiplication unit, which consists of eight multiplications and four additions.

QR Decomposition (QRD): a part of the QR decomposition based recursive least-squares (QRD-RLS) adaptive filtering algorithm [KS99]. Similar to the work by Matousek, et al. [MTP+02], we implement the diagonal arithmetic element of QRD algorithm, using both LNS and FLP number representations. This arithmetic element includes one addition, three multiplications, two squares, two divisions, and one square root.

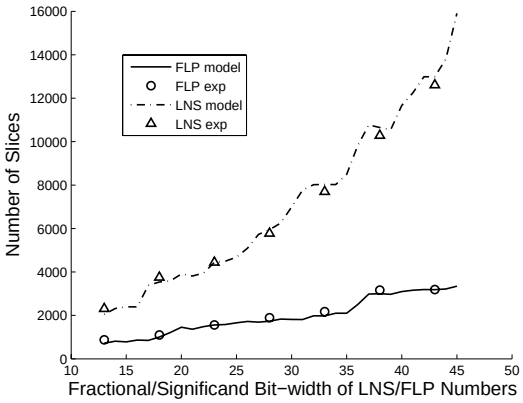
Radiative Monte Carlo Simulation (RMCS): M. Gokhale et al. [GFA+04] present an acceleration of radiative Monte Carlo simulation on FPGA, using floating-point numbers. The simulation traces photons emitted from the surfaces of a 2-D enclosure. The most inner loop, which is also the most computationally intensive part, is implemented on an FPGA. It performs 12 multiplications, 1 division, 3 additions and 7 subtractions.

7.2.2 Area Cost

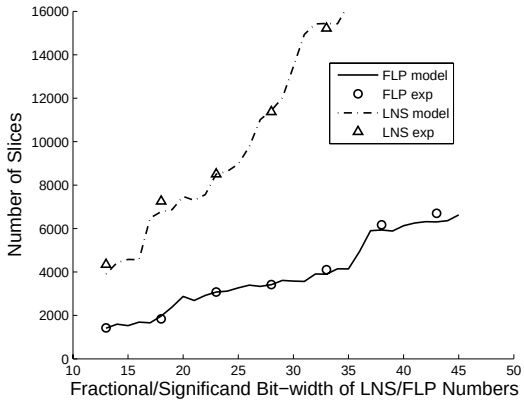
Figure 7.1 shows the comparison on area cost between LNS and FLP designs of the four case studies. Costs of different resources (logic slices, HMULs, and BRAMs) are converted into an equivalent total number of slices, using the same conversion rates as in Section 4.4.1. The lines in the figure show the scaled modelling results for the designs, provided by the area modelling tool introduced in Section 4.4.2, while the markers show the experimental results of mapping the designs into a Xilinx Virtex-4 FX100 FPGA. Similar to the results in Section 4.4.2, the scaled modelling results provide an accurate estimation of the area consumption, with maximum errors around 15% and average errors around 5%.

For the three designs DSCG, MM and RMCS, which mainly consist of addition and multiplications, the LNS designs consume two to three times more resources than the FLP designs with the same bit widths. Besides, when the bit width increases, the area cost of LNS designs increases faster than FLP designs. This is mainly because the LNS arithmetic functions (f_1 and f_2) become more and more difficult to evaluate for higher precision bit width, and cost many more HMULs and BRAMs. For the design QRD, which uses more multiplications, divisions and square roots, the LNS designs consume up to 73.5% less resources than the FLP designs.

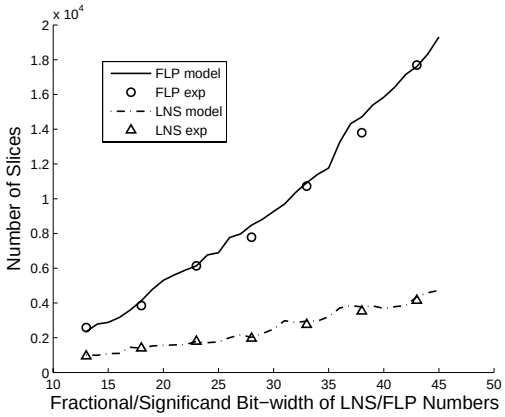
In general, LNS has a big potential to reduce the area cost for applications having a large fraction of MUL, DIV and exponential operations.



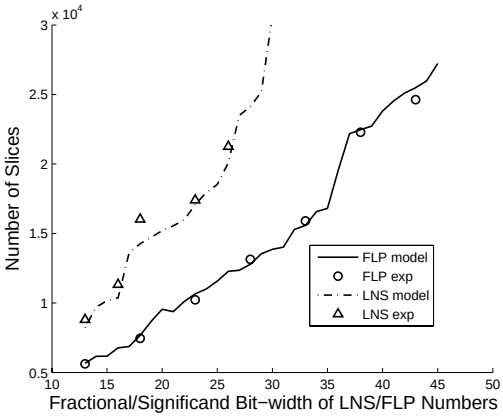
(a) Area Cost of 'DSCG', LNS versus FLP



(b) Area Cost of 'MM', LNS versus FLP



(c) Area Cost of 'QRD', LNS versus FLP



(d) Area Cost of 'RMCS', LNS versus FLP

Figure 7.1: Area cost comparison of LNS and FLP designs. 'DSCG' (Digital Sine Cosine waveform Generator), 'MM' (two-by-two Matrix Multiplication), 'QRD' (QR Decomposition), and 'RMCS' (Radiative Monte Carlo Simulation) are compared here. The line illustrates the scaled area modelling results (denoted as 'model') while the markers show the experimental results (denoted as 'exp') on a Virtex-4 FX100 FPGA.

7.2.3 Accuracy

To perform accuracy investigation, we implement software versions of the designs using C++ long double data-type (80-bit floating point on x86 implementation), and use their results as the true values for error analysis. We then use our bit-accurate simulator (introduced in Section 4.4.3) to acquire the results for the designs with different number representations and different bit-width settings.

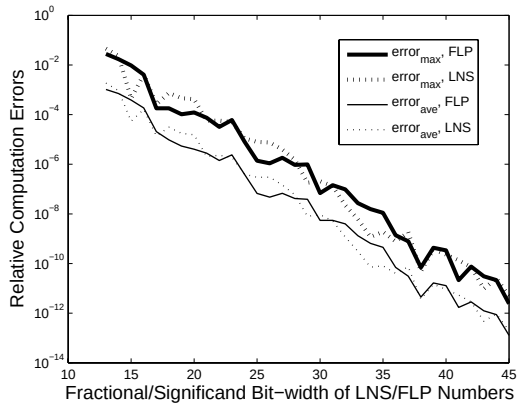
As 'DSCG', 'MM' and 'QRD' are small arithmetic modules that can be used in large designs,

we do not include the conversion units from/to FLP numbers when we investigate the accuracy. Instead, we use Maple [Wat] with 100 decimal-digit precision to convert the input and output of the LNS circuits into FLP format, and compare with the results of FLP designs. On the other hand, 'RMCS' is already a complete design and needs to communicate with a software part implemented in floating-point format. Thus, we include the conversion units in the LNS design.

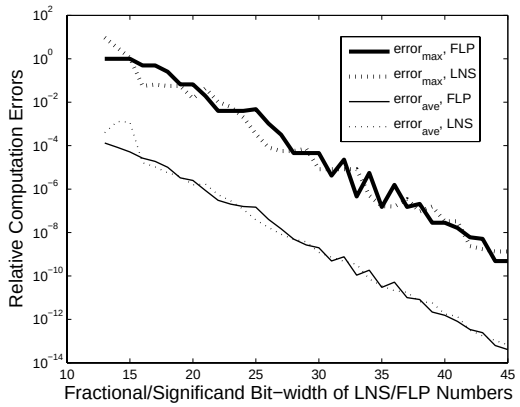
The other issue is what values we use to drive the computation. For 'DSCG' and 'RMCS', the computation only needs several initial values to start, and can continue by itself. For 'MM' and 'QRD', we generate random values that are uniformly distributed in the representation range of the number formats, and feed in these values to perform the accuracy experiments.

Figure 7.2 shows the comparison on accuracy between LNS and FLP designs of the four case studies. For the three simple ones, 'DSCG', 'MM' and 'QRD', LNS provides a similar maximum and average errors to the FLP designs with the same bit width. This shows that LNS arithmetic units with faithful roundings are able to provide similar accuracy to FLP units with rounding to nearest, which supports the arguments of [AW01a].

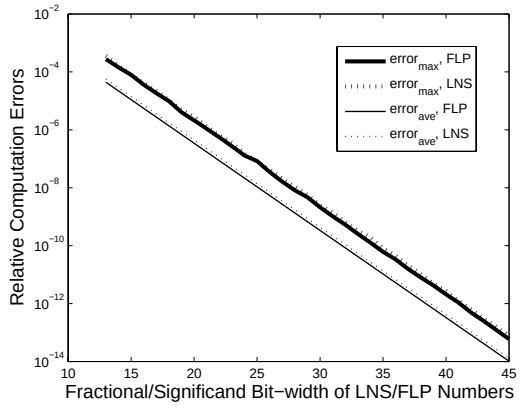
'RMCS' is much more complicated than the other three. Since it is performing a Monte Carlo simulation of photons' movements, the errors in each iteration get propagated to the next round and affect the accuracy of the final result. The core part that we implement on the FPGA calculates a value based on a photon's movement, compares the value to a pre-defined threshold, and determines whether the photon is absorbed or reflected. Thus, we take this value as the final result of each iteration, and analyse its error behaviours for different bit widths. As shown in Figure 7.2(d), for small bit widths, both FLP and LNS designs produce very large errors as the simulated photons make totally different movements from the accurate case. With the bit width increasing gradually, there emerges a turning point, where the simulation of photon movements moves from chaos to the accurate case and the error drops dramatically. The LNS design gets to the turning point with 16 fractional bits, while FLP design needs 17 mantissa bits. This is also shown in the final results of the simulation, which are the number of photons emitted and the number of photons absorbed on each surface. The LNS design with



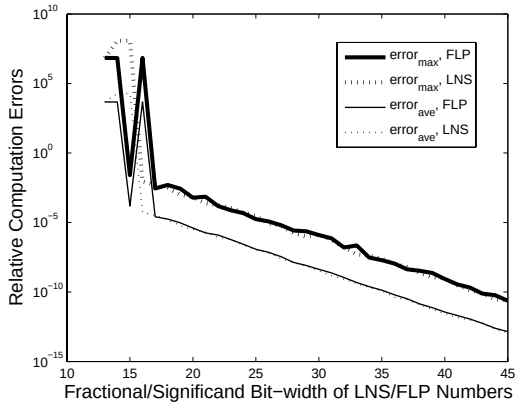
(a) Errors of 'DSCG', LNS versus FLP



(b) Errors of 'MM', LNS versus FLP



(c) Error of 'QRD', LNS versus FLP



(d) Error of 'RMCS', LNS versus FLP

Figure 7.2: Accuracy comparison of LNS and FLP designs. 'DSCG' (Digital Sine Cosine waveform Generator), 'MM' (two-by-two Matrix Multiplication), 'QRD' (QR Decomposition), and 'RMCS' (Radiative Monte Carlo Simulation) are compared here. We investigate both maximum and average errors.

29 bits (1 sign bit, 8 integer bits and 20 fractional bits) produces the same photon numbers as the long double software version, while the FLP design needs 31 bits (1 sign bit, 8 exponent bits and 22 significand bits) to produce the same values. Thus, even with two fewer bits, the LNS design manages to achieve the same or even better accuracy than FLP.

7.2.4 Performance

Although our current tool does not include estimation of performance results, the infrastructure provides support to fast prototype the applications and collect the performance results using

Table 7.1: Comparison of performance between LNS and FLP designs. The number format is described in the form of 'sign:integer:fraction' for LNS, and 'sign:exponent:significand' for FLP. 'thr' denotes the throughput of the design.

Number Format	BRAM	HMUL	slices	thr/MHz	Number Format	BRAM	HMUL	slices	thr/MHz
DSCG					**QRD**				
LNS (1:8:23)	9	22	3798	161.4	LNS (1:8:23)	3	8	1562	155.6
FLP (1:8:23)	0	16	1456	133.3	FLP (1:8:23)	0	24	5886	116.0
MM					**RMCS**				
					FLP (1:8:23) [GFA+04]	0	144	6758	33.4
LNS (1:8:23)	18	44	7218	161.3	LNS (1:8:23)	39	102	14485	137.5
FLP (1:8:23)	0	32	2501	141.2	FLP (1:8:23)	0	40	10012	117.6

Xilinx tools.

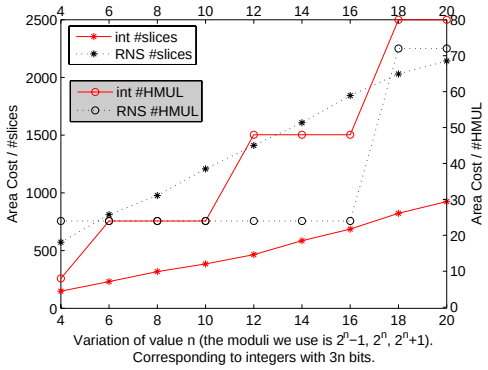
Table 7.1 shows the performance results of the four designs for a typical 32-bit setting. Resource utilisation details are also included in this table. For 'DSCG' and 'MM', LNS designs consume over twice as many resources as FLP designs, while providing 14.2% to 21.1% higher throughput. For 'QRD', the FLP design consumes three times more resources than the LNS design, while the LNS design also improves the throughput by 34.1%.

For 'RMCS', compared to FLP designs, LNS consumes 44.7% more slices, around 2.5 times HMULs and needing additional BRAMs. The throughput of the LNS design is 16.9% higher than FLP designs. Compared to the original FLP implementation of the RMCS computation core [GFA+04], the FLP circuits generated by our tool infrastructure consume 48% more slices, but use many fewer HMULs. Meanwhile, both our LNS and FLP circuits provide 3 to 4 times higher throughput than the original design.

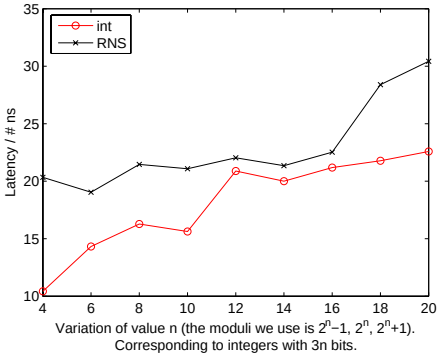
7.3 Comparing RNS and Integer/Fixed-point Designs

7.3.1 Small Matrix Multiplication: RNS versus Integer

We compare the RNS designs and integer designs for a 2-by-2 matrix multiplication processing core. As we focus on the area cost and latency of the processing core itself, the conversions between RNS and binary forms are not considered.



(a) Comparison of area cost, using slices and HMULs.



(b) Comparison of latency.

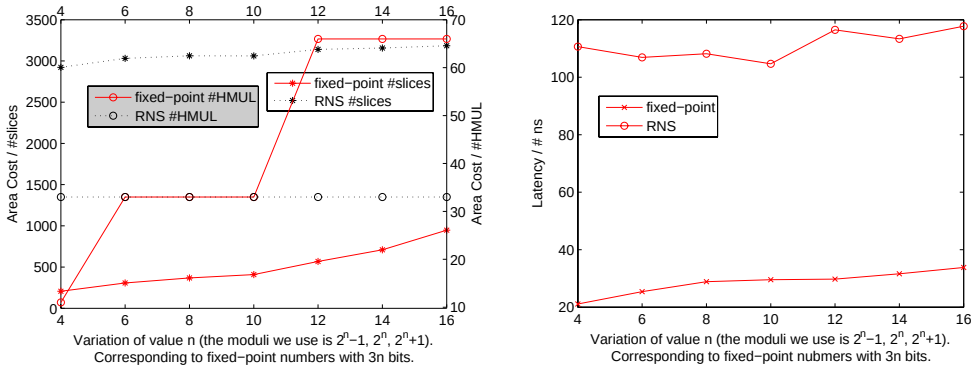
Figure 7.3: 2-by-2 matrix multiplication: area cost and latency comparison of RNS and integer designs.

The processing core consists of 8 multiplications and 4 additions. As shown in Figure 7.3, the consumption of HMULs is greatly reduced in RNS designs. For the cases where $n = 12, 14, 16$, the RNS designs consume only half the HMULs of the integer designs. Similarly, for the cases where $n = 12, 14, 16$, the RNS designs provide a similar latency to integer designs. As there are far fewer HMULs involved in the design, the critical path is shorter. In the other cases, the latency introduced in the additional modular operations outweighs the reduced portion in HMULs; the RNS designs show a higher latency. The RNS designs also consume around 500 to 1000 extra slices than integer designs due to the extra modular operations.

7.3.2 FIR Filter: RNS versus Fixed-point

In the second case study, we compare RNS and fixed-point representations in an 11-tap Finite Impulse Response (FIR) filter design. The computation involves 11 multiplications and 10 additions. We assume that the input and output values are in the form of fixed-point, thus forward and reverse converters are included in the RNS design. HMULs are used for multipliers.

As shown in Figure 7.4, the usage of HMULs is again greatly reduced for large bit width cases. In RNS designs, the number of consumed HMULs is reduced by half when $n = 12, 14, 16$. However, due to the converters added into the design and the additional modular operations,



(a) Comparison of area cost.

(b) Comparison of latency.

Figure 7.4: 11-tap FIR filter: area cost and latency comparison of RNS and fixed-point designs.

the RNS designs consume around 2000 more slices and show a much higher latency.

In this specific case, we implement the design on a Virtex IV FX100 FPGA, which consists of 42176 slices and 192 HMULs. By using RNS representation, we can cut the number of HMULs by half for large bit width cases ($n = 12, 14, 16$), and fit 5 copies of the same design into one FPGA, while the fixed-point version can only support 2 copies in one FPGA. On the other hand, we can also have more taps in the RNS design to produce a better FIR filter.

7.4 Exploration with Bit-width Optimisation

In this section, we explore a number of target designs with fixed-point, floating-point, and LNS numbers. The exploration performs a bit-width optimisation for all three different number representations, and produces designs with optimised bit widths.

7.4.1 Description of Application Kernels

Polynomial Approximation (poly4): we examine the degree-four minimax polynomial for the approximation to $y = \log(1 + x)$ where $x \in [0, 1)$. Horner's rule is used to evaluate the

polynomial:

$$y = (\cdots(c_n x + c_{n-1})x + \cdots)x + c_0 \tag{7.2}$$

where $c_0 \ldots c_n$ are the polynomial coefficients. The coefficients are obtained using minimax algorithm to minimise the maximum absolute error.

RGB to YCbCr (rgb2ycc): we consider the RGB to YCbCr colour space converter specified by the JPEG 2000 standard [SCE01]. The input signals R, G and B are assumed to be 8-bit unsigned integers. Shifts are used for the multiplications by 0.5.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.16875 & -0.33126 & 0.5 \\ 0.5 & -0.41869 & -0.08131 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{7.3}$$

Matrix Multiplication (matrix2): 2-by-2 matrix multiplication using Strassen's algorithm [Str69] is considered, which is commonly used as a basic processing element for large matrix multiplications. We assume the elements of the input matrices are in [0,1).

$$\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} \tag{7.4}$$

The four quadrants of the result matrix can be calculated as follows:

$$\begin{aligned} y_{00} &= p_0 + p_3 - p_4 + p_6 \\ y_{01} &= p_2 + p_4 \\ y_{10} &= p_1 + p_3 \\ y_{11} &= p_0 + p_2 - p_1 + p_5 \end{aligned} \qquad \begin{aligned} p_0 &= (a_{00} + a_{11})(b_{00} + b_{11}) \\ p_1 &= (a_{10} + a_{11})b_{00} \\ p_2 &= a_{00}(b_{01} - b_{11}) \\ p_3 &= a_{11}(b_{10} - b_{00}) \\ p_4 &= (a_{00} + a_{01})b_{11} \\ p_5 &= (a_{10} - a_{00})(b_{00} + b_{01}) \\ p_6 &= (a_{01} - a_{11})(b_{10} + b_{11}) \end{aligned} \tag{7.5}$$

B-Splines (bspline): we examine uniform cubic B-splines, commonly used for image warping applications [JLR03]. The B-spline basis functions, B_0, B_1, B_2 and B_3, are defined by

$$\begin{aligned} B_0(u) &= \tfrac{(1-u)^3}{6} & B_2(u) &= \tfrac{-3u^3+3u^2+3u+1}{6} \\ B_1(u) &= \tfrac{3u^3-6u^2+4}{6} & B_3(u) &= \tfrac{-u^3}{6} \end{aligned} \tag{7.6}$$

where $u \in [0, 1)$. In the implementation of this design, techniques such as shifts instead of multiplications, and sharing of common intermediate results are used.

Discrete Cosine Transform (dct8): we consider the 8-point discrete cosine transform (DCT) design [MJ95]. A vector of input data $x_{0...7}$ can be transformed to DCT values $y_{0...7}$ by

$$\begin{aligned} \begin{bmatrix} y_0 \\ y_2 \\ y_4 \\ y_6 \end{bmatrix} &= \begin{bmatrix} c_0 & c_0 & c_0 & c_0 \\ c_2 & c_5 & -c_5 & c_2 \\ c_0 & -c_0 & -c_0 & c_0 \\ c_5 & -c_2 & c_2 & -c_5 \end{bmatrix} \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \end{bmatrix} \\ \begin{bmatrix} y_1 \\ y_3 \\ y_5 \\ y_7 \end{bmatrix} &= \begin{bmatrix} c_1 & c_3 & c_4 & c_6 \\ c_3 & -c_6 & -c_1 & -c_4 \\ c_4 & -c_1 & c_6 & c_0 \\ c_6 & -c_4 & c_3 & -c_1 \end{bmatrix} \begin{bmatrix} x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix} \end{aligned} \tag{7.7}$$

where $c_{0...7}$ are trigonometric constants. We use 8-bit unsigned integers for the elements in the input vector $x_{0...7}$.

7.4.2 Area Reduction Results

To evaluate our proposed approaches, we apply our bit-width optimisation tool to analyse the different application kernels described in the previous section. Experiments are performed using

both dynamic and static optimisation approaches. Three different number representations, fixed-point, floating-point and LNS, are investigated. The resulting designs, with uniform or non-uniform bit widths, are synthesised using ASC, placed and routed using Xilinx tools to acquire area results. Designs are mapped into Xilinx Virtex-II XC2V6000 as long as they can fit in its 33792 slices, 144 BRAMs and 144 HMULs. Some large designs are mapped into Xilinx Virtex 4 FX100, which provides 42176 slices, 376 BRAMs and 160 HMULs. Unless specified, all the designs are optimised with the accuracy requirement that the absolute error is kept below 2^{-16}.

Figure 7.5 shows the area costs of designs that use optimised uniform bit width over the entire design, compared with designs using optimised non-uniform bit widths, which are generated by R-Tool. Note that R-Tool provides both static and dynamic methods for range and precision analysis, and the results shown in this figure are average values of the static approach and dynamic approach. For LNS numbers, as BRAMs and HMULs are also used in the designs (for implementation of LNS ADD/SUB), we use the same ratios as in Section 4.4.1 to convert various consumed resources into number of slices.

For all five application kernels with three different number representation systems, when compared to designs using uniform bit width, our designs using non-uniform bit widths on average provide 9.7% reduction in area, which amounts to 1469 slices. For maximum cases, our approach provides 7066 slices reduction in the LNS **dct8** design, and 17.3% area reduction for the LNS **matrix2** design.

7.4.3 Comparison of Static and Dynamic Approaches

For both range analysis and precision analysis, R-Tool provides options to use the static or dynamic approach to analyse the application. At the range analysis stage, the users can use the static affine arithmetic approach to determine the range of variables, while they can also use the dynamic simulation approach to find out the maximum/minimum values of variables based on a specific input data set. At the precision analysis stage, the users can either use the static error model based on affine arithmetic to express the errors of the output variables, or

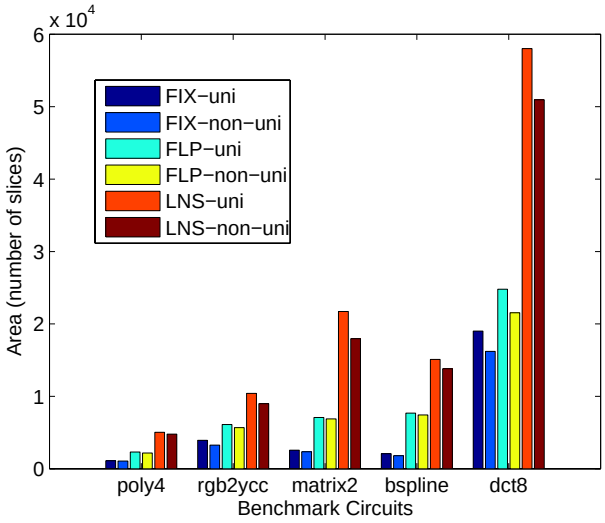


Figure 7.5: Areas of designs using optimised uniform bit width, compared to designs using optimised non-uniform bit widths. 'FIX', 'FLP', and 'LNS' stands for designs using fixed-point, floating-point and LNS numbers respectively. 'uni' refers to optimised designs using uniform bit widths, while 'non-uni' refers to designs using non-uniform bit widths.

use the dynamic automatic differentiation approach to determine the error function through simulation based on input data.

These options enable the users to make choices on the tradeoff between different guarantees of accuracy and resource cost of the hardware designs. Using static approaches, the resulting design meets the required accuracy with any input values, but it may consume a significant amount of resources to handle rare situations. On the other hand, using dynamic approaches, the resulting design only meets the required accuracy with the tested input values. As long as the tested values provide a good coverage of general cases, the design can still perform well, and the resource costs can be reduced.

To obtain some practical results on this problem, we perform optimisation of the application kernels using both static and dynamic approaches and compare their results. In the static approach, the optimisation is performed based on the specified input value range; while in the dynamic approach, the optimisation is performed based on an input file, which contains over 10000 input values that are generated within the input range using a uniform random pattern.

Figure 7.6 shows the area usage of the fixed-point designs optimised for non-uniform bit widths,

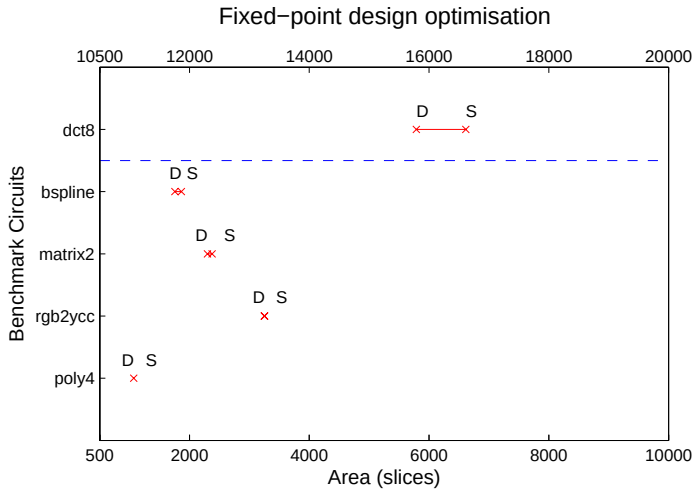


Figure 7.6: Post-place-and-route area comparison for fixed-point designs, where D and S represent the dynamically and statically optimised designs respectively. As the area cost of **dct8** is much larger than the other designs, it uses separate x axis on the top.

using both dynamic and static techniques. The dynamic approach gives similar area results to the static approach for the small benchmark circuits **poly4**, **rgb2ycc**, **matrix2** and **bspline**. For benchmark **dct8**, which is larger, dynamic approach produces an optimised design with 5% area reduction (825 slices).

Figure 7.7 provides a similar comparison for floating point implementations of the case study examples. Here the dynamic optimised designs provide an area reduction from 2.6% to 13.9% (from 98 to 1027 slices) compared to the static optimised designs.

Figure 7.8 shows the comparison for LNS implementations. The performance is not as good as the cases for floating-point designs. This is due to the BRAM sharing mechanism in our LNS arithmetic units, as described in Section 4.3.3. As one BRAM supports two concurrent reading ports, in our LNS arithmetic library, a pair of units with the same bit width can share the same BRAMs to store the same coefficients. Therefore, if there are arithmetic units using the same bit width in the design, they can share their BRAMs and reduce the total resource cost. Otherwise, each arithmetic unit has its own BRAM and the total resource becomes higher. This brings some randomness into the resource cost of the design. As shown in the figure, for **rgb2ycc** and **bspline** (the two circled with dots), the static approach happens to generate

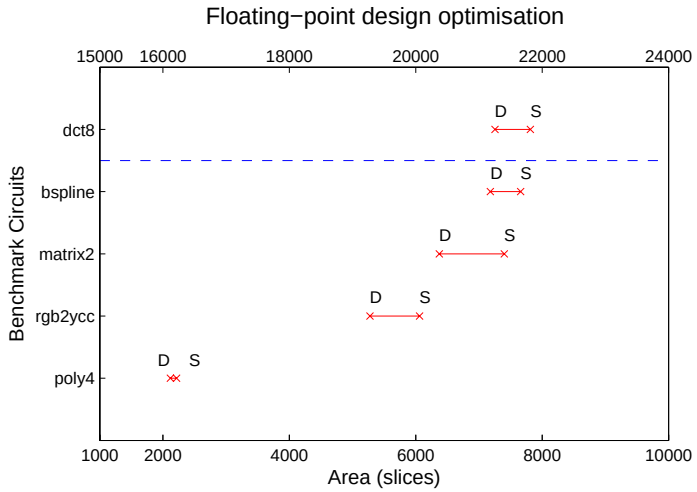


Figure 7.7: Post-place-and-route area comparison for floating-point designs, where D and S represent the dynamically and statically optimised designs respectively. As the area cost of **dct8** is much larger than the other designs, it uses a separate x axis on the top.

designs with several pairs of uniform bit widths, while the dynamic approach generates designs with bit widths that are different from each other. As a result, the design optimised using the dynamic approach consumes more area usage than the static approach. Thus, when we consider designs using LNS arithmetic, or other designs that apply a similar resource-sharing methodology, minimising the bit widths of the variables may not be enough to minimise the total resource cost. In these situations, we also need to consider how to maximise the resource sharing in the entire design. Especially for the bit-width optimisation problem, we want to use non-uniform bit widths over the circuit, but for some parts of the design, we can apply uniform bit widths to enable resource sharing.

For the other cases of LNS designs (**poly4**, **matrix2** and **dct8**), the static approach does not provide advantage on the sharing of BRAMs, and the dynamic approach gives smaller area costs. The best performance is achieved with **matrix2**, with 10.1% reduction in area (1919 slices).

R-Tool enables the users to investigate the variation of design area usage with different design error constraints. Figure 7.9 shows a case study for the floating-point implementation of the **matrix2** benchmark. The resulting area increases with the increase in design output error

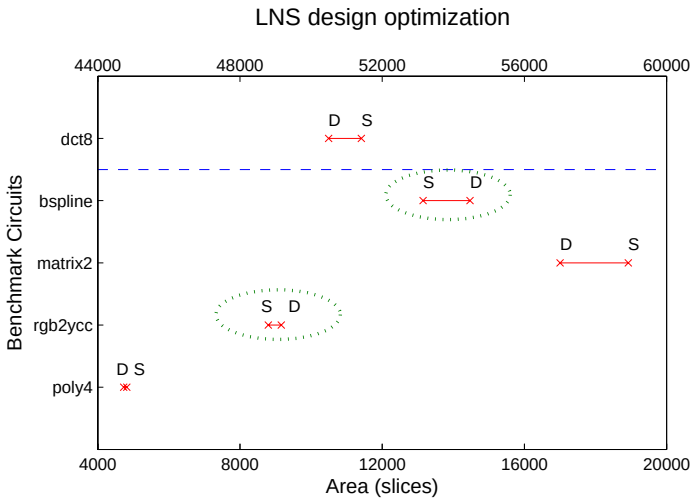


Figure 7.8: Post-place-and-route area comparison for LNS designs, where D and S represent the dynamically and statically optimised designs respectively. As the area cost of **dct8** is much larger than the other designs, it uses separate x axis on the top.

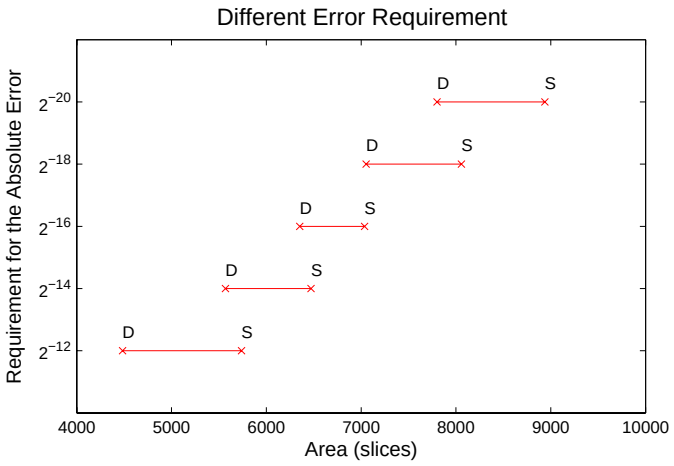


Figure 7.9: Post-place-and-route area comparison for floating-point **matrix2** circuit with different output error specifications. D and S represent the dynamically and statically optimised designs respectively.

specification, and the dynamically optimised designs provide a constant area reduction from 9.7% to 13.9% (around 1000 slices), compared to the static optimised designs.

Besides error requirement, we can also use the tool to figure out the effect from variation of the input variables' ranges. Figure 7.10 shows a case study for bit-width optimisation of the floating-point **matrix2** design, with different input ranges. Similar to error requirements, the

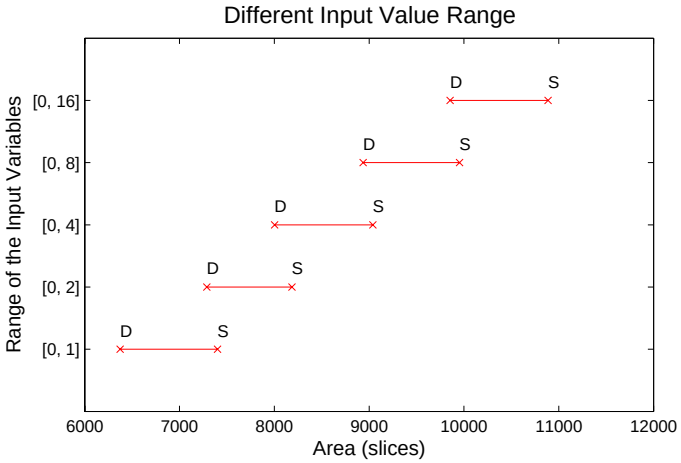


Figure 7.10: Post-place-and-route area comparison for floating-point **matrix2** circuit with different input value ranges. D and S represent the dynamically and statically optimised designs respectively.

increase in input variables' range also causes the increase of the resulted designs. The dynamic approach still shows 9.5% to 13.9% reduction in area (898 to 1037 slices) compared to the static approach.

7.5 Summary

This chapter presented case studies of exploring practical applications or computation kernels for different number representations.

We have compared floating-point and LNS numbers on realistic applications, such as digital sine/cosine waveform generator, matrix multiplication, QR decomposition, and radiative Monte Carlo simulation. LNS shows better efficiency than FLP in QR decomposition with 73.5% fewer slices and 34.1% higher throughput. On the accuracy side, LNS achieves better results than FLP with two fewer bits in the radiative Monte Carlo simulation.

We have also compared integer/fixed-point and RNS numbers on matrix multiplication and FIR filter. In applications that involves a large number of multiplications, the RNS representation can reduce by up to a half the number of HMULs for large bit-width settings, and thus fit more

designs into the FPGA device.

The above comparisons did not include bit-width optimisation. To evaluate the performance of our bit-width optimisation tool, we explored practical application kernels (polynomial approximation, colour space conversion, matrix multiplication, B-Splines, and DCT transform) with fixed-point, floating-point, and LNS numbers, perform bit-width optimisation for the three number representations, and produce designs with optimised bit widths. The results show that both static and dynamic analysis approaches achieve significant area reduction in non-uniform bit-width optimisation. Compared to the static approach, in most cases the dynamic approach provides a further reduction (up to 13.9%) in area consumption.

Chapter 8

Case Study: Seismic Computations

8.1 Introduction

This chapter examines some case studies which illustrate real challenges in seismic computations.

Seismic imaging applications in the oil and gas industry involves tera-bytes of data collected from fields. For each data sample, the imaging algorithm usually tries to improve the image quality by performing more costly computations. Thus, there is an increasingly large demand for high performance computation over huge volumes of data. Among all the different kinds of imaging algorithms, downward-continued-based migration [GS85] is the most prevalent high-end imaging technique today and reverse-time migration appears to be one of the most promising future imaging techniques.

Compared to conventional microprocessors, FPGAs offer a different streaming computation architecture. Computations we want to perform are mapped into circuit units on the FPGA board. Thus, as long as the FPGA has enough area for the computation units, we can perform many tasks in parallel and achieve a high throughput of the application. Previous work has already achieved 20x acceleration for pre-stack Kirchhoff time migration [HLS02] and 40x acceleration for subsurface offset gathers [PC07].

Besides the capability to perform computations in parallel, FPGAs also support application-specific number representations. Different number representations lead to different complexities of the arithmetic units, and thus to different costs and performance of the resulting circuit design [DBS06]. Switching to a number representation that fits a given application better can sometimes greatly improve the performance or reduce the cost.

In the case studies for seismic computations, we apply our platform to explore fixed-point and floating-point designs. As the computation involves many additions, and some square root, sine/cosine, and comparison operations, LNS and RNS numbers are not efficient choices.

Besides exploring different number systems, we can also perform bit-width analysis to study the tradeoff between precision and computing speed. For example, by reducing the precision from 32-bit floating-point to 16-bit fixed-point, the number of arithmetic units that fit into the same area can be increased by many times. The performance of the application is also improved significantly. Meanwhile, we also need to watch for the possible degradation of accuracy in the computation results. We need to check whether the accelerated computation using reduced precision is still generating meaningful results.

As mentioned in Section 6.3.4, seismic computation involves numerous iterations on the image data. The error function or range information derived from data-flows can easily become meaningless due to the excessive propagation of errors in each step. Thus, we need to make modifications to R-Tool, use dynamic simulations to perform range and precision analysis, and determine the minimum precision that can still generate good enough seismic results. Section 8.3 provides more detailed discussion about bit-width exploration for seismic computations.

By using the most appropriate number format with minimised bit widths, we implement core algorithms in seismic applications (complex exponential step in downward-continued-based migration) on an FPGA and show speedups of around 7 times. Provided sufficient bandwidth between CPU and FPGA, we show that a further increase to 48x speedup is possible.

8.2 Computation Bottlenecks in Seismic Applications

Downward-continued-based migration comes in various flavours including common azimuth migration [BP96], shot-profile migration, source-receiver migration, plane-wave or delayed-shot migration, and narrow-azimuth migration. Depending on the flavour of the downward-continuation algorithm, there are four potential computation bottlenecks:

- In many cases the dominant cost is the FFT step. The dimension of the FFT varies from 1-D (tilted plane-wave migration [SB06]) to 4-D (narrow-azimuth migration [Bio03]. The FFT cost is often dominant due to its $n \cdot log(n)$ asymptotic cost, n being the number of points in the transform, and the cache-unfriendly nature of multi-dimensional FFTs.

- The FK step, which involves evaluating (or looking up) a square root function and performing a complex exponential is a second potential bottleneck. The high operation count per sample can consume significant cycles.

- The FX step, which involves a complex exponential, or sine/cosine multiplication, has a similar, but computationally less demanding, profile. Subsurface offset gathers for shot-profile or plane-wave migration, particularly 3D subsurface offset gathers, can be an overwhelming cost. The large operation count per sample and the cache-unfriendly nature of the data usage pattern can be problematic.

- For finite-difference-based schemes a significant convolution cost can be involved.

The primary bottleneck of reverse time migration is applying the finite-difference stencil. In addition to the large operation count (5 to 31 samples per cell) the access pattern has poor cache behaviour for real size problems. Beyond applying the 3-D stencil the next most dominant cost is in implementing damping boundary conditions. Methods such as Perfectly Matched Layers (PML) can be costly [ZC96]. Finally, where reverse time migration is used for velocity analysis, subsurface offset gathers need to be generated. The same cost profile that exists in downward-continued-based migration exists for reverse-time migration.

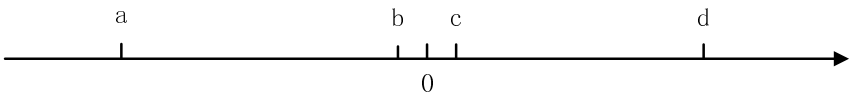


Figure 8.1: Four points to record in the profiling of range information.

In this work, we focus on the computation bottleneck of the FK step in downward-continued-based migration, which includes a square root function and a complex exponential operation. We perform automated precision exploration of this computation core, so as to determine the minimum precision that can still generate accurate enough seismic images.

8.3 Bit-width Exploration for Seismic Computation

8.3.1 Range Analysis

As mentioned above, seismic computation involves numerous iterations, and is not suitable for static analysis techniques. Instead, we apply dynamic simulation methods to collect range and distribution information for the variables. The idea of our approach is to instrument every target variable in the code, adding function calls to initialise data structures for recording range information, and to modify the recorded information when the variable value changes.

For the range information of the target variables (variables to map into the circuit design), we keep a record of four specific points on the axis, shown in Figure 8.1. The points a and d represent the values furthest from zero, i.e., the maximum absolute values that need to be represented. Based on their values, the integer bit width of fixed-point numbers can be determined. Points b and c represent the values closest to zero, i.e., the minimum absolute values that need to be represented. Using both the minimum and maximum values, the exponent bit width of floating-point numbers can be determined.

For the distribution information of each target variable, we keep a number of buckets to store the frequency of values at different intervals. Figure 8.2 shows the distribution information recorded for the real part of variable 'wfld' (a complex variable). In each interval, the frequency

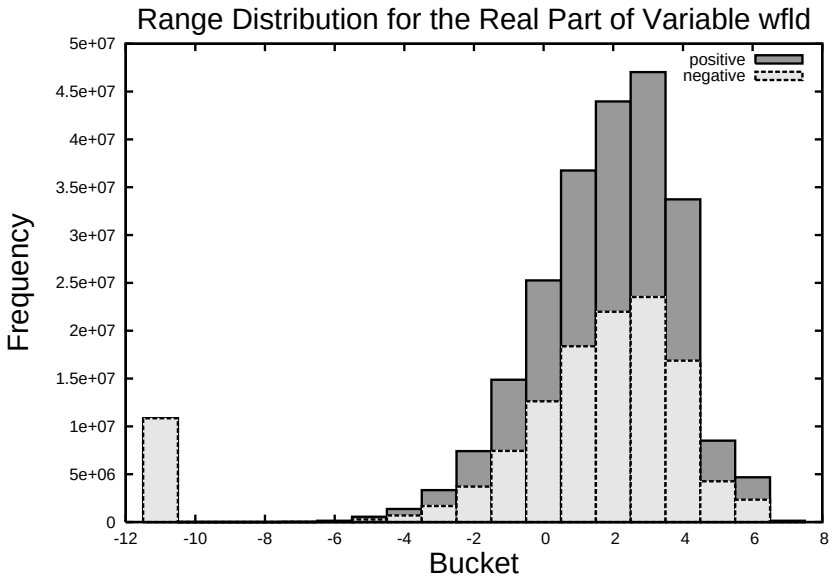


Figure 8.2: Range distribution of the real part of variable 'wfld'. The leftmost bucket with index= -11 is reserved for zero values. The other buckets with index= x store the values in the range $(10^{x-1}, 10^x]$.

of positive and negative values are recorded separately. The results show that, for the real part of variable 'wfld', in each interval, the frequencies of positive and negative values are quite similar, and the major distribution of the values falls into the range 10^{-1} to 10^4.

The distribution information provides a rough metric for the users to make an initial guess about which number representations to use. If the values of the variables cover a wide range, floating-point and LNS number formats are usually more suitable. Otherwise, fixed-point numbers are enough to handle the range.

8.3.2 Precision Analysis

In the precision analysis of R-Tool (as shown in Section 6.3.3), a requirement of the absolute error is used as a constraint of the optimisation process. However, a specified requirement for absolute error does not work for seismic processing. To find out whether the current configuration of precision bit width is accurate enough, we need to run the entire program to produce the seismic image, and find out whether the image contains the correct pattern information. Thus, we need to run simulations to produce different image results for different bit widths.

On the other hand, the original R-Tool applies heuristic algorithms, such as ASA [Ing04], to find a close-to-optimal set of bit widths for different variables. These heuristic algorithms may require millions of test runs to check whether a specific set of values meet the constraints or not. This is acceptable when the test run is only a simple error function and can be processed in nanoseconds. In our seismic processing application, depending on the problem size, it takes half an hour to several days to run one test set and achieve the result image. Thus, heuristic algorithms become impractical.

A simple and straightforward method to solve the problem is to use uniform bit width for all variables, and either iterate over a set of possible values or use a binary search algorithm to select an appropriate bit width. Based on the range information and the internal behaviour of the program, we can also try to divide the variables in the target Fortran code into several different groups, and assign a different uniform bit width for each different group. For instance, in the FK step, there is a clear boundary that the first half performs square, square root and division operations to calculate an integer value, and the second half uses the integer value as a table index, and performs sine, cosine and complex multiplications to get the final result. Thus, in the hardware circuit design, we divide the variables into two groups based on which half it belongs to. Further more, in the second half of the function, some of the variables are trigonometric values in the range of $[-1, 1]$, while the other variables represent the seismic data in the image and scale up to 10^6. Thus they can be further divided into two parts and assigned bit widths separately.

8.3.3 Evaluating the Accuracy of Seismic Images

The accuracy of a generated seismic image depends on whether the image has a good geophysical estimation of the underlying earth. To judge whether the image is accurate enough, we compare it to a 'target' image, which is processed using single-precision floating-point and assumed to contain the correct pattern.

To perform this pattern comparison automatically, we use techniques based on Prediction Error Filters (PEFs) [Cla99] to highlight differences between two images. The PEF is a convolution

operator, which can be used to capture the inverse spectrum of a seismic data array.

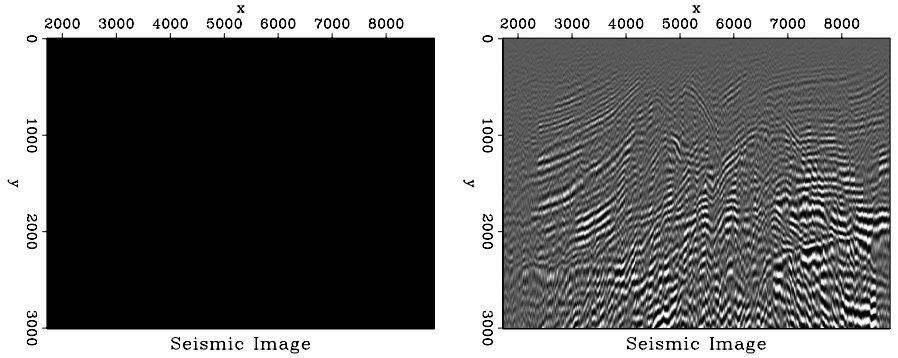
The basic workflow of comparing image a to image b (assume image a is the 'target' image) is as follows:

- Divide image a into overlapping small regions of 40×40 pixels, and estimate PEFs for these small regions.
- Apply these PEFs to both image a and image b to get results a' and b'.
- Apply the following algebraic combinations of the images a' and b' to get result image c: $c = ABS\left(\left(\frac{Smooth(ABS(b'))}{Smooth(ABS(a'))} - 1\right) \cdot Envelope(a)\right)$, where ABS takes the absolute value of the seismic data, $Smooth$ is a smoothing filter, and $Envelope$ is a smoothed envelope function.
- In the last step, stack the data items in c to obtain a value indicating the image differences.

By the end of the above workflow, we achieve a single value which describes the difference from the generated image to the 'target image'. For convenience of discussion afterwards, we call this value as 'Difference Indicator' (DI). The major reason that we use the DI value rather than a simple subtraction of the two images is that the DI value filters the differences caused by random noises in the images and magnifies the differences caused by the actual calculation.

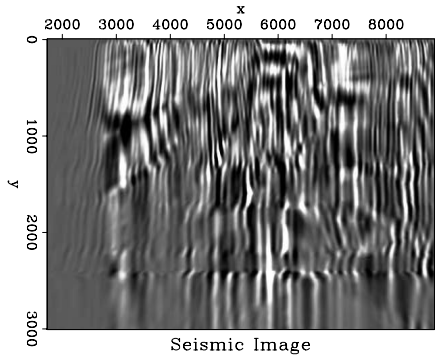
Figure 8.3 shows a set of different seismic images calculated from the same data set, and their DI values compared to the image with correct pattern. The image labelled "correct pattern" is calculated using single-precision floating-point, while the other images are calculated using fixed-point designs with different bit-width settings. All these images are results of the bit-accurate value simulator introduced in Section 4.4.3.

If the generated image contains no information at all (as shown in Figure 8.3(a)), the comparison does not return an finite value. This is mostly because a very low precision is used for the calculation. The information is lost during numerous iterations and the result only contains zeros or infinities. If the comparison result is in the range of 10^4 to 10^5 (Figure 8.3(b) and

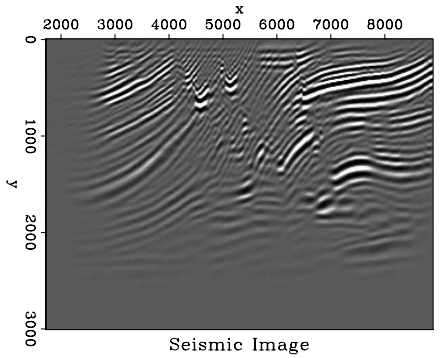


(a) DI = Inf

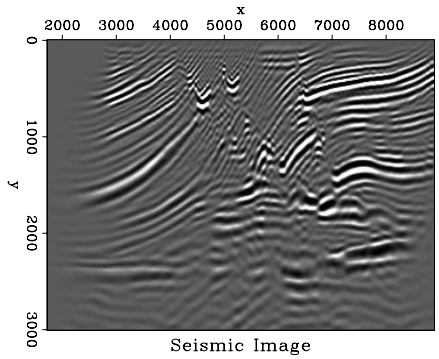
(b) DI = 10^5



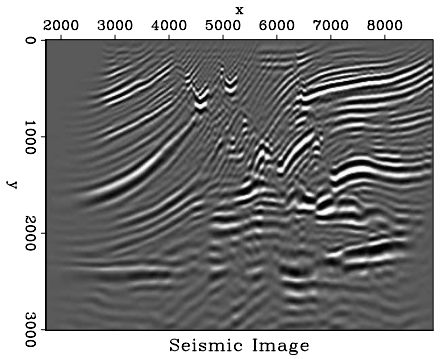
(c) DI = 10^4



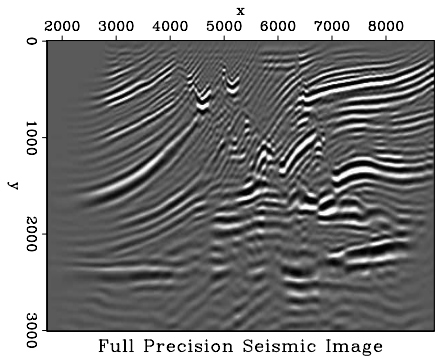
(d) DI = 10^3



(e) DI = 10^2



(f) DI = 10



(g) Correct Pattern

Figure 8.3: Examples of seismic images with different DI (Difference Indicator) values. 'Inf' means the approach does not return a finite difference value. '10^x' means the difference value is in the range of $[1 \times 10^x, 1 \times 10^{x+1})$.

8.3(c)), the image contains random patterns which are far different from the correct one. With a comparison result in the range of 10^3 (Figure 8.3(d)), the image contains similar pattern to the correct one, but information in some parts is lost. With a comparison result in the range of 10^2 or smaller, the generated image contains almost the same pattern as the correct one.

Note that the DI value is calculated from algebraic operations on the two images being compared. The magnitude of DI value is only a relative indication of the difference between the two images. The actual usage of the DI value is to determine the boundary between the images that contains mostly noise and the images that provide useful patterns of the earth model. From the samples shown in Figure 8.7, in this specific case, the DI value of 10^2 is a good guidance value for acceptable accuracy of the design. From the bit-width exploration results shown in Section 8.4, we can see that the DI value of 10^2 also happens to be a precision threshold where the image turns from noise into an accurate pattern as the bit width increases.

8.4 Accelerating Seismic Computation on FPGAs

8.4.1 Brief Introduction

In this section, we will provide a practical case study that tries to accelerate one of the computation bottlenecks in seismic applications, the FK step in a downward-continued-based migration. The major computation of the FK step involves evaluating a square root function and performing a complex exponential.

The governing equation for the FK step is the Double Square Root Equation (DSR) [Cla00], which is shown in Equation (8.1). The DSR equation describes how to downward continue a wave-field U one depth Δz step. The equation is valid for a constant velocity medium v and is based on the wave number of the source k_s and receiver k_g. The variable ω denotes the frequency. The code takes the approach of building *a priori* a relatively small table of the possible values of $\frac{vk}{\omega}$. The code then performs a table lookup that converts a given $\frac{vk}{\omega}$ value to an approximate value of the square root.

```
! generation of table step%ctable

do i=1,size(step%ctable)
    k=ko*step%dstep*dsr%phase(i)
    step%ctable(i)=dsr%amp(i)*cmplx(cos(k),sin(k))
end do

! the core part of function wei_wem

do i4=1,size(wfld,4)
    do i3=1,size(wfld,3)
        do i2=1,size(wfld,2)
            do i1=1,size(wfld,1)

                k = sqrt(step%kx(i1,i3)**2 + step%ky(i2,i4)**2)
                itable =max(1, min(int(1 + k/ko / dsr%d) , dsr%n))
                wfld(i1,i2,i3,i4,i5)=wfld(i1,i2,i3,i4,i5)*step%ctable(itable)

            end do
        end do
    end do
end do
```

Figure 8.4: The code for the major computations of the FK step.

$$U(\omega, k_s, k_g, z + \Delta z) = \exp\left[-i\omega v\left(\sqrt{1 - \frac{vk_g}{\omega}} + \sqrt{1 - \frac{vk_s}{\omega}}\right)\right] U(\omega, k_s, k_g, z) \qquad (8.1)$$

The code shown in Figure 8.4 is the computationally intensive portion of the FK step. Outside the core part, a small loop is performed to pre-compute the sine and cosine values. The core part is a four-level nested loop, which calculates a square root function and performs a complex exponential operation. Although there are not so many operations in each iteration, the same operations have to be performed on each data item of the 'wfld' array. The 'wfld' array usually has a size of several million data items in practical applications. The computation pattern of this step makes it an ideal target to map to a streaming hardware circuit on an FPGA.

8.4.2 Circuit Design

The mapping from the software code to a hardware circuit design is straightforward for most part. Figure 8.5 shows the general structure of the circuit design. Compared with the software

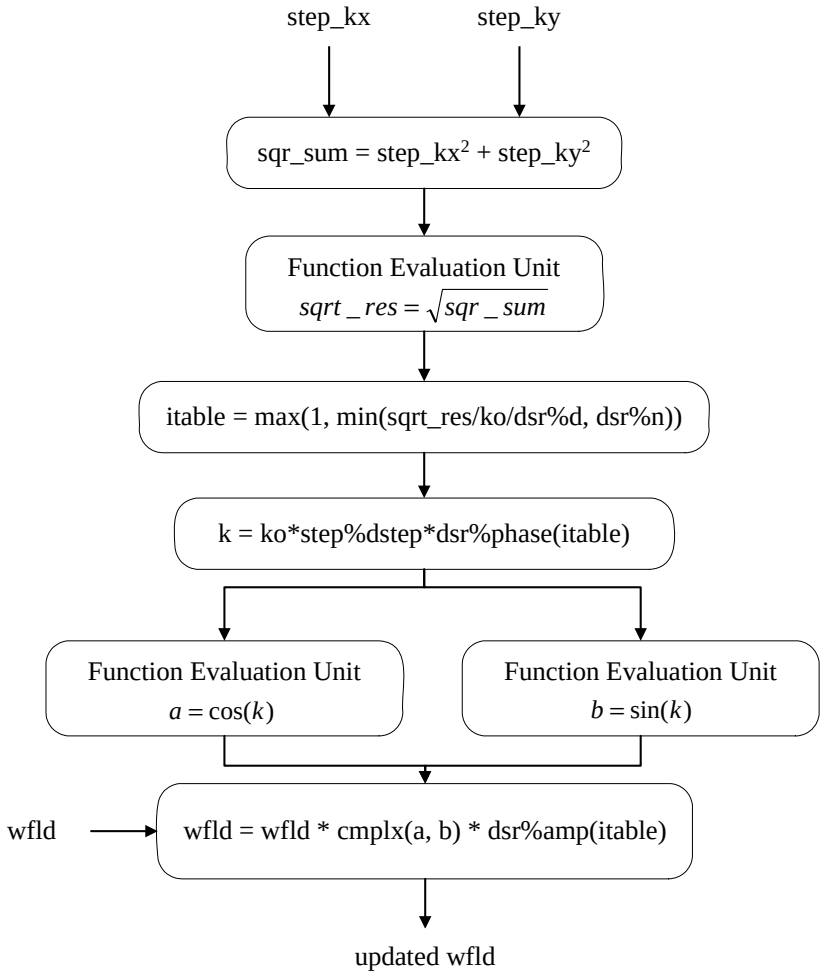


Figure 8.5: General structure of the circuit design for the FK step.

Fortran code shown in Figure 8.4, one big difference is the handling of the sine and cosine functions. In the software code, the trigonometric functions are calculated outside of the four-level nested loop, and stored as a look-up table. In the hardware design, to take advantage of the parallel calculation capability provided by the numerous logic units on the FPGA, the calculation of the sine/cosine functions are merged into the processing core of the inner loop. Three function evaluation units are included in this design, to produce values for the square root, cosine and sine functions separately.

To map these functions into efficient units on the FPGA board, we use a table-based uniform polynomial approximation approach, based on Dong-U Lee's work on optimising hardware function evaluation [LGML05]. The evaluation of the two functions can be divided into three

different phases [Mul97]:

- Range Reduction: reduce the range of the input variable x into a small interval that is convenient for the evaluation procedure. The reduction can be multiplicative (e.g. $x' = x/2^{2n}$ for square root function) or additive (e.g. $x' = x - 2\pi \cdot n$ for sine/cosine functions).

- Function Evaluation: approximate the value of the function using a polynomial within the small interval.

- Range Reconstructions: map the value of the function in the small interval back into the full range of the input variable x.

To keep the whole unit small and efficient, we use a degree-one polynomial so that only one multiplication and one addition are needed to produce the evaluation result. Meanwhile, to preserve the approximation error at a small scale, the reduced evaluation range is divided into uniform segments. Each segment is approximated with a degree-one polynomial, using the minimax algorithm. In the FK step, the square root function is approximated with 384 segments in the range of [0.25, 1] with a maximum approximation error of 4.74×10^{-7}, while the sine and cosine functions are approximated with 512 segments in the range of [0, 2] with a maximum approximation error of 9.54×10^{-7}.

8.4.3 Special Handling for Fixed-point Designs

Table 8.1 shows the value range of some typical variables in the FK step. Some of the variables (in the part of square root and sine/cosine function evaluations) have a small range within [0, 1], while other values (especially 'wfld' data) have a wide range from 10^{-14} to 10^{6}. If we use floating-point representations, their wide representation ranges are enough to handle these variables. If we use fixed-point number representations instead, special handling is needed to achieve acceptable accuracy over wide ranges.

Table 8.1: Profiling results for the ranges of typical variables in function 'wei_wem'. 'wfld_real' and 'wfld_img' refer to the real and imaginary parts of the 'wfld' data. 'Max' and 'Min' refer to the maximum and minimum absolute values of variables.

Variable	step%x	ko	wfld_real	wfld_img
Max	0.377	0.147	3.918e6	3.752e6
Min	0	7.658e-3	4.168e-14	5.885e-14

The first issue to consider in fixed-point designs is the enlarged error caused by the division after the evaluation of the square root ($\frac{\sqrt{step\%x^2+step\%y^2}}{ko}$). The values of $step\%x$, $step\%y$ and ko come from the software program as input values to the hardware circuit, and contain errors propagated from previous calculations or caused by the truncation/rounding into the specified bit width on hardware. Suppose the error in the square root result $sqrt\_res$ is E_{sqrt}, and the error in variable ko is E_{ko}, assuming the division unit itself does not bring extra error, the error in the division result is given by $E_{sqrt} \cdot \frac{sqrt\_res}{ko} + E_{ko} \cdot \frac{sqrt\_res}{ko^2}$. According to the profiling results, ko holds a dynamic range from 0.007658 to 0.147, and $sqrt\_res$ has a maximum value of 0.533 (variables step%x and step%y have similar ranges). In the worst case, the error from $sqrt\_res$ can be magnified by 70 times, and the error from ko magnified by approximately 9000 times.

To solve the problem of enlarged errors, we perform shifts at the input side to keep the three values $step\%x$, $step\%y$ and ko in a similar range. The variable ko is shifted by the distance d_1 so that the value after shifting falls in the range of $[0.5, 1)$. The variables $step\%x$ and $step\%y$ are shifted by another distance d_2 so that the larger value of the two also falls in the range of $[0.5, 1)$. The difference between d_1 and d_2 is recorded, so that after the division, the result can be shifted back into the correct scale. In this way, the $sqrt\_res$ has a range of $[0.5, 1.414]$ and ko has a range of $[0.5, 1]$. Thus the division only magnifies the errors by an order of 3 to 6. Meanwhile, as the three variables $step\%x$, $step\%y$ and ko are originally in single precision floating-point representation in software, when we pass their values after shifts, a large part of the information stored in the mantissa part can be preserved. Thus, a better accuracy is achieved through the shifting mechanism for fixed-point designs.

Figure 8.6 shows experimental results about the accuracy of the table index calculation when using shifting compared to not using shifting, with different uniform bit widths. The possible range of the table index result is [1, 2001]. As it is the index for tables of smooth sequential

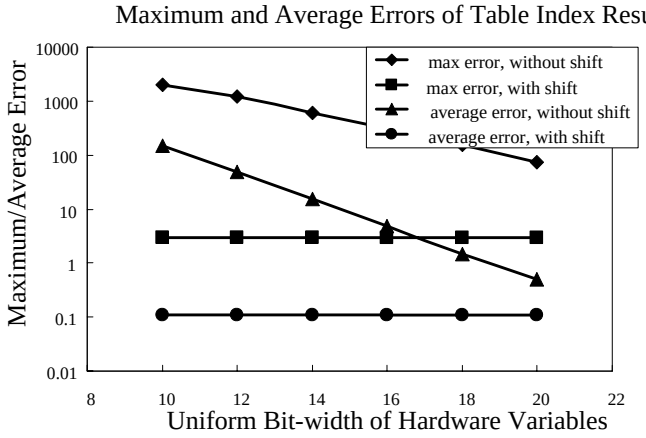


Figure 8.6: Maximum and average errors for the calculation of the table index when using and not using the shifting mechanism in fixed-point designs, with different uniform bit width values from 10 to 20.

values, an error less than five is generally acceptable. We use the table index results calculated with single-precision floating-point as the true values for error calculation. When the uniform bit width of the design changes from 10 to 20, designs using the shifting mechanism show a stable maximum error of 3, and an average error around 0.11. On the other hand, the maximum errors of designs without shifting vary from 2000 to 75, and the average errors vary from approximately 148 to 0.5. These results show that, the shifting mechanism provides much better accuracy for the table index calculation part in fixed-point designs.

The other issue to consider is the representation of 'wfld' data variables. As shown in Table 8.1, both the real and imaginary parts of 'wfld' data have a wide range from 10^{-14} to 10^{6}. Generally, fixed-point numbers are not suitable to represent such wide ranges. However, in this seismic application, the 'wfld' data is used to store the processed image information. It is more important to preserve the pattern information shown in the data values rather than the data values themselves. Thus, by omitting the small values, and using the limited bit width to store the information contained in large values, fixed-point representations still have a big chance to achieve an accurate image in the final step. In our design, for convenience of bit-width exploration, we scale down all the 'wfld' data values by a ratio of 2^{-22} so that they fall into the range of [0, 1).

8.4.4 Bit-width Exploration Results

The original software Fortran code of the FK step performs the whole computation using single-precision floating-point. We use the image generated by the Fortran software as the full-precision "true" image. To investigate the effect of different number representations on the accuracy of the whole application, we replace the code of the FK step with bit-accurate simulation code (as described in Section 4.4.3) that can be configured with different number representations and different bit widths, and generate results for different settings. The approach for accuracy evaluation, introduced in Section 8.3.3, is used to provide DI values that indicate the differences in the patterns of the resulting seismic images from the pattern in full-precision image.

Fixed-point Designs

In the first step, we apply a uniform bit width over all the variables in the design. We change the uniform bit width from 10 to 20. With a uniform bit width of 16, the design provides a DI value around 100, which means the image contains a pattern almost the same as the correct one.

In the second step, as mentioned in Section 8.3.2, according to their characteristics in range and operational behaviour, we can divide the variables in the design into different groups and apply a uniform bit width in each group. In the hardware design for the FK step, the variables are divided into three groups: SQRT, the part from the beginning to the table index calculation, which includes an evaluation of the square root; SINE, the part from the end of SQRT to the evaluation of the sine and cosine functions; and WFLD, the part that multiplies the complex values of 'wfld' data with a complex value consisting of the sine and cosine values (for phase modification), and a real value (for amplitude modification). To perform the accuracy investigation, we keep two of the bit widths constant, and change the other one gradually to see its effect on the accuracy of the entire application.

Figure 8.7(a) shows the DI values of the generated images when we change the bit width of the SQRT part from 6 to 20. The bit widths of the SINE and WFLD parts are set to 20 and 30

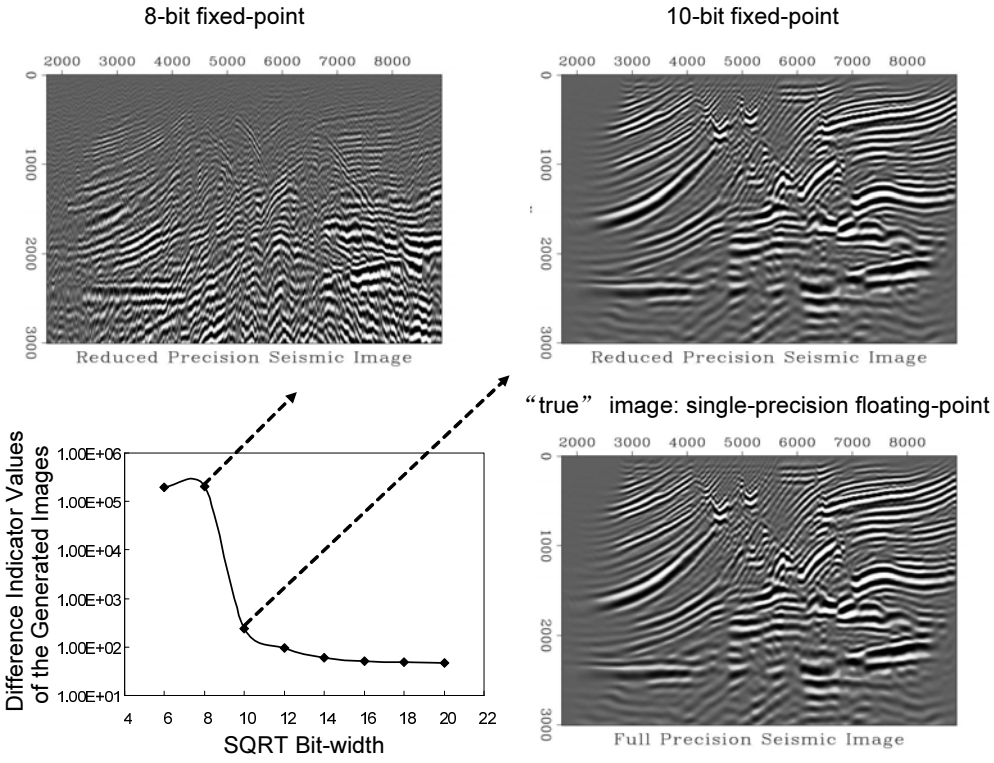
respectively. Large bit widths are used for the other two parts so that they do not contribute much to the errors and the effect of variables' bit width in SQRT can be extracted. The case of SQRT bit widths shows a clear precision threshold at the bit width of 10. When the SQRT bit width increases from 8 bits to 10 bits, the DI value decreases from around 10^5 to around 10^2. The significant improvement in accuracy is also seen in the generated seismic images. The image on the left of Figure 8.7(a) is generated with the 8-bit design. Compared to the "true" image calculated with single-precision floating-point, the lower part of the image is mainly noise signals, while the lower part starts to show a similar pattern as the correct one. The difference between the qualities of the lower and upper parts is because of the imaging algorithm, which calculates the image from summation of a number of points at the corresponding depth. In acoustic models, there are generally more sample points when we go deeper into the earth. Therefore, using the same precision, the lower part shows a better quality than the upper part. The image on the right of Figure 8.7(a) is generated with 10-bit design, and already contains almost the same pattern as the "true" image.

In a similar way, we perform the exploration for the other two parts, and acquire the precision threshold 10, 12 and 16 for the SQRT, SINE and WFLD parts respectively. However, as the above results are acquired with two out of the three bit widths set to very large values, the practical solution may be slightly larger than these values. Meanwhile, constrained by the current I/O bandwidth of 64 bits per cycle, the sum of the bit widths for SQRT and WFLD parts shall be less than 30. We perform further experiments for bit widths around the initial guess point, and find out that bit widths of 12, 16, 16 for the three parts provide a DI value of 131.5, and also meet the bandwidth requirement.

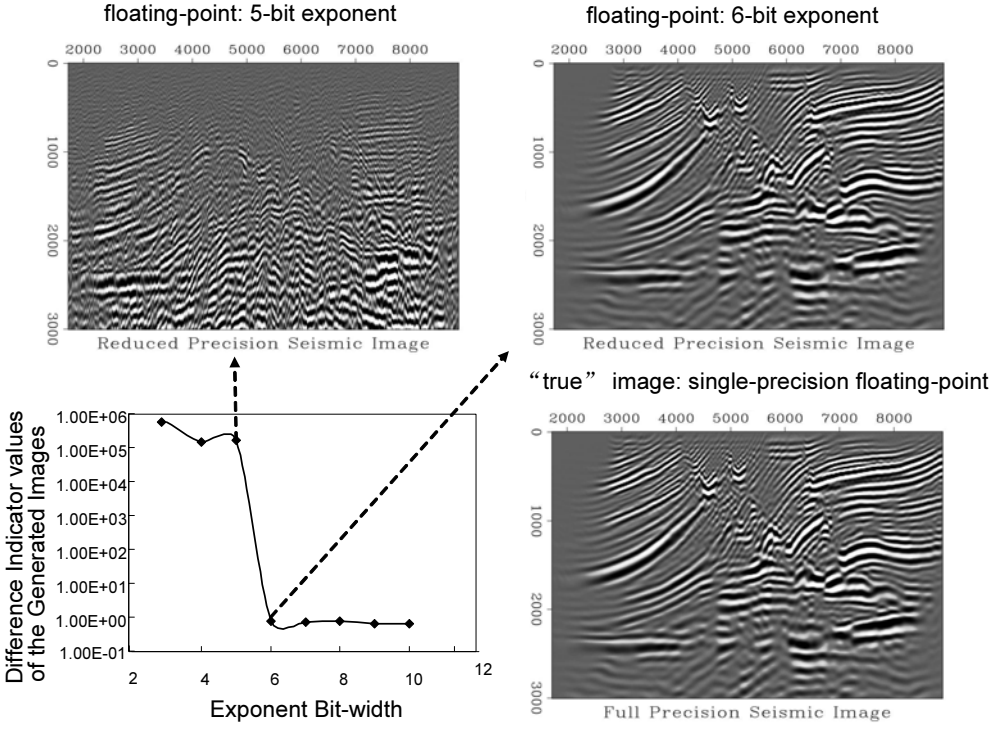
Floating-point Designs

In floating-point design of the FK step, we perform an exploration of different exponent and mantissa bit widths. We use a uniform bit width for all the variables. When we investigate one of them, we keep the other one with a constant high value.

Figure 8.7(b) shows the case that we change the exponent bit width from 3 to 10, while we keep the mantissa bit width as 24. There is again a clear cut at the bit width of 6. When the



(a) DI values for different SQRT bit widths in a fixed-point design.



(b) DI values for different exponent bit widths in a floating-point design.

Figure 8.7: Exploration of fixed-point and floating-point designs with different bit widths.

Table 8.2: Speedups of the FK step achieved on FPGA compared to software solutions (including data transfer time).

Size of Data Set	Software Time	FPGA Time	Speedup
43056	5.32 ms	0.84 ms	**6.3**
216504	26.1 ms	3.77 ms	**6.9**

exponent bit width is smaller than 6, the DI value of the generated image is at the level of 10^5. When the exponent bit width increases to 6, the DI value decreases to around 1.

With a similar exploration of the mantissa bit width, we figure out that exponent bit width of 6 and mantissa bit width of 16 provide the minimum bit widths needed to achieve a DI value around 10^2. Experiment confirms that this combination produces an image with a DI value of 43.96.

8.4.5 Hardware Acceleration Results

The hardware acceleration tool used in this project is the FPGA computing platform MAX-1, provided by Maxeler Technologies [Max]. It contains a high performance Xilinx Virtex IV FX100 FPGA, which consists of 42176 slices, 376 BRAMs, and 192 embedded multipliers. Meanwhile, it provides a high bandwidth interface (PCI Express x8: 2G bytes per second) to the software side residing in CPUs.

Based on the exploration results of different number representations, the fixed-point design with bit widths of 12, 16, and 16 for three different parts is selected in our hardware implementation. The design produces images containing the same pattern as the single-precision floating-point implementation, and has the smallest bit widths, i.e., the lowest resource cost among all the different number representations.

Table 8.2 shows the speedups we can achieve on FPGA compared to software solutions running on Intel Xeon CPU of 1.86GHz. We experiment with two different sizes of data sets. For each of the data sets, we record the processing time for 10000 times and calculate the average as the result. Speedups of 6.3 and 6.9 times are achieved for the two different data sets respectively.

Table 8.3: Resource Cost of the FPGA Design for the FK step.

Type of Resource	Used Units	Percentage
slices	12032	28%
BRAMs	59	15%
Embedded Multipliers	16	10%

Table 8.3 shows the resource cost to implement the FK step on the FPGA card. It utilises 28% of the logic units, 15% of the BRAMs (memory units) and 10% of the arithmetic units. Considering that a large part (around 20%) of the used logic units are circuits handling PCI-Express I/O, there is still much potential to put more processing cores onto the FPGA card and to gain even higher speedups.

8.5 Further Potential Speedups

One of the major constraints for achieving higher speedups on FPGAs is the limited bandwidth between the FPGA card and the CPU. For the current PCI-Express interface provided by the MAX-1 platform, in each cycle, we can only read 8 bytes into the FPGA card and write back 8 bytes to the system.

An example is the implementation of the FK step, described in Section 8.4. As shown in Figure 8.4, in our current designs, we take $step\%kx$, $step\%ky$ and both the real and imaginary parts of $wfld$ as inputs to the circuit on FPGA, and take the modified real and imaginary parts of $wfld$ as outputs. Therefore, although there is plenty of space on the FPGA card to support multiple cores, the interface bandwidth can only support one single core (getting a speedup of around 7 times).

However, in the specific case of FK step, there are further techniques we can utilise to gain some more speedups. From the codes in Figure 8.4, we can find out, $wfld$ varies with all the four different loop indices, while $step\%kx$ and $step\%ky$ only vary with two of the four loop indices. To take advantage of this characteristic, we can divide the processing of the loop into two parts: in the first part, we use the bandwidth to read in the $step\%kx$ and $step\%ky$ values, without doing any calculation; in the second part, we can devote the bandwidth to read in

$wfld$ data only, and start the processing as well. In this pattern, suppose we are processing a 100-by-100-by-100-by-100 four-level loop, the bandwidth can support two cores processing concurrently while spending 1 out of 100 cycles to read in the $step\%kx$ and $step\%ky$ values in advance. In this way, we are able to achieve a speedup of $6.9 \times 2 \times \frac{100}{101} \approx 13.7$ times. Furthermore, if we assume there is unlimited communication bandwidth, the cost of BRAMs (15%) becomes the major constraint. We can then put 6 concurrent cores on the FPGA card and achieve a speedup of $6.9 \times 7 \approx 48$ times.

Another possibility is to put as much computation as possible onto the FPGA card, and reduce the communication cost between FPGA and CPU. If multiple portions of the algorithm are performed on the FPGA without returning to the CPU the additional speedup can be considerable. For instance, as mentioned in Section 8.2, the major computation cost in downward-continued-based migration lies in the multi-dimensional FFTs and the FK step. If the FFT and the FK step can reside simultaneously on the FPGA card, the communication cost between the FFT and the FK step can be eliminated completely. In the case of 3D convolution in reverse time migration, multiple time steps can be applied simultaneously.

8.6 Summary

This chapter described our work on accelerating seismic applications by using customised number representations on FPGAs. The focus was to improve the performance of the FK step in downward-continued-based migration. To investigate the tradeoff between precision and speed, we developed a tool that performs an automated precision exploration of different number formats, and determine the minimum precision that can still generate good enough seismic results. By using the minimised number format, we implement the FK step in downward-continued-based migration on FPGA and show speedups around 7 times. We also show that there is further potential to accelerate these applications by above 10 or even 48 times.

Chapter 9

Conclusions

This thesis presented my work on investigating the design space of different number representations and achieving well-customised number representations for applications, i.e. application-specific number representations, on FPGAs.

The major conclusions are as follows:

- **Number representations have a very big impact on the area costs, accuracies, and performances of the applications.** By simply switching from floating-point to LNS, the area cost of the QR decomposition benchmark (Section 7.2) is reduced by 73.5%, with performance improved by 34.1%. For the radiative Monte Carlo simulation benchmark (Section 7.2), the LNS design with two fewer bits manages to provide the same or even better accuracy than floating-point designs. For a 11-tap FIR filter (Section 7.3), compared to fixed-point designs, RNS designs reduce the number of HMULs (hardware multipliers) by half in certain cases.

- **The most appropriate number format is different for each specific applications.** While LNS fits the QR decomposition case much better than floating-point numbers, the floating-point designs become the better one for digital sine-cosine waveform generator, two-by-two matrix multiplication, and radiative Monte Carlo simulation (Section 7.2).

Therefore, we need some kind of method, such as our automated exploration platform, to determine the most appropriate number format for a given application.

- **Bit width is an important parameter to consider when customising the number representation for an application.** Benchmarks in Section 7.4 demonstrated that by performing an automated bit-width optimisation under a given error requirement, the area costs can be reduced by 9.7% on average. The practical example in seismic computation also showed that under the constraint of producing seismic images of the same quality, we can reduce the bit width from 32 to 16, and fill much more computation into the same chip.

- **An automated exploration tool can be quite helpful for determining the good solutions in a large design space.** One example is our LNS arithmetic unit design in Chapter 4. The design involves up to five different design options, and we want a systematic optimisation for all of them. While some of the design options can be optimised through analytical methods, for some other options, the automated design generation process enables us to explore the possible choices and combinations of them and find out the best solution from an empirical point of view. In Chapter 8, we also tried to find the most appropriate number format through an automated exploration of different number formats and different bit widths.

The following sections in this chapter provide a summary of the major contents in this thesis, and points out the topics that we plan to investigate for the future research work.

9.1 Summary of the Thesis

The first part of the thesis (Chapters 3, 4, 5, and 6) demonstrated a platform that enables automated exploration of the number representation design space.

Automation requires arithmetic unit generation. This thesis presented optimised arithmetic library generators for logarithmic and residue arithmetic units. The generators support a

wide range of bit widths and in most cases achieve area/latency reduction or performance improvement over previous designs. Combining the arithmetic library generators with A Stream Compiler (ASC), which supports fixed-point and floating-point arithmetic, our platform is able to generate efficient designs of an application using different number representations.

Chapter 4 presented our work on optimised LNS arithmetic unit generation. First, we introduced a general polynomial approximation approach for LNS arithmetic function evaluation. Second, we developed a library generator that produces LNS arithmetic libraries containing $+, -, *, /$ operators as well as converters between floating-point and LNS numbers. The generated libraries cover a wide range of bit widths and provide a systematic optimisation of LNS arithmetic. Third, to facilitate comparison between LNS and floating-point designs, we developed a two-level area cost estimation tool: the first level uses area models to acquire results in less than a second, with an average error around 5%; the second level performs an automated mapping of different designs onto FPGAs to acquire exact results. To evaluate accuracy, we developed a bit-accurate simulator based on truncation or rounding of 80-bit floating-point calculations. When compared with existing LNS designs in [HBUH05], our generated units provide in most cases 6% to 37% reduction in area and 20% to 50% reduction in latency.

Chapter 5 described our work on optimised RNS arithmetic unit generation.We provided improved designs for residue arithmetic units. Especially for reverse converters from RNS to binary numbers, we proposed a novel design that uses only n-bit additions and consumes 14.3% less area than the previous work in [WSAS02]. We also developed an RNS arithmetic library generator for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. The generator supports a wide range of RNS numbers, and enables us to perform an extensive comparison between RNS and other number representations at both the arithmetic unit level and the application level.

Generation of arithmetic units requires us to know how many bits to use for each variable. To address this problem, Chapter 6 showed an automatic bit-width optimisation tool called R-Tool. R-Tool supports both dynamic and static analysis methods to provide optimised bit widths for different number systems (fixed-point, floating-point, and LNS numbers) and different guarantees of accuracy. The static method tackles the problem conservatively to

assure the required accuracy over all possible inputs, while the dynamic approach reduces resource costs by only guaranteeing accuracy for tested input values. Experiments on practical application kernels showed that both dynamic and static approaches achieve significant area reduction for designs using optimised non-uniform bit widths. Compared to the static approach, the dynamic approach in most cases provides a further area reduction up to 13.9%.

Putting it all together, the second part (Chapters 7 and 8) of the thesis explored the effects of application-specific number representations with case studies on practical benchmarks.

Chapter 7 presented case studies exploring practical applications or computation kernels for different number representations, which can be summarised into three different categories:

- We compared floating-point and LNS numbers on realistic applications, such as digital sine/cosine waveform generation, matrix multiplication, QR decomposition, and radiative Monte Carlo simulation. LNS shows better efficiency than FLP in QR decomposition with 73.5% fewer slices and 34.1% higher throughput. On the accuracy side, LNS achieves better results than FLP with two fewer bits in the radiative Monte Carlo simulation.

- We also compared integer/fixed-point and RNS numbers on matrix multiplication and FIR filtering. In applications that involve a large number of multiplications, the RNS representation can reduce by up to half the number of HMULs required for large bit-width settings, thus enabling more designs to fit into the FPGA device.

- We explored practical application kernels (polynomial approximation, colour space conversion, matrix multiplication, B-Splines, and DCT transforms) with different bit widths for fixed-point, floating-point, and LNS numbers. We applied our bit-width optimisation tool, R-Tool, to the benchmarks, to produce designs with optimised bit widths. The results showed that both static and dynamic analysis approaches achieve significant area reduction in non-uniform bit-width optimisation. Compared to the static approach, in most cases the dynamic approach provides a further reduction (up to 13.9%) in the area usage.

Chapter 8 described our work on solving the real challenges in seismic computations using customised number representations on FPGAs. The focus is to improve the performance of the FK step in downward-continued-based migration . To investigate the tradeoff between precision and speed, we performed an automated precision exploration of different number formats, and calculated the minimum precision that can still generate good enough seismic results. By using the most appropriate number format with minimised bit widths, we implemented the FK step in forward continued based migration on an FPGA and showed speedups around 7 times. We also showed that there is further potential to accelerate these applications by above 10 or even 48 times.

In summary, this thesis presented an automated design space exploration platform that tries to achieve application-specific number representation. The platform provides optimised arithmetic unit generation and bit-width optimisation for different number systems. By applying the platform on practical applications or computation kernels, we showed that application-specific number representation can bring benefits to area cost, accuracy, and throughput for various applications. The exploration of different number representations also helps us to understand the characteristics of different number systems, and the tradeoffs between between different metrics and constraints.

9.2 Future Work

Based on the results collected from current experiments, we plan to focus future research work on the following topics:

- Further optimisation of LNS and RNS arithmetic units.
- Using multiple number representations in one FPGA application.
- Power consumption of different number representations.
- New FPGA architectures for application-specific number representations.

The following sections provide more detailed descriptions of these topics.

9.2.1 Further Optimisation of LNS and RNS Arithmetic Units

As shown in Chapters 4 and 7, although the LNS arithmetic units generated in our exploration platform already show significant improvement from previous LNS designs, for applications with a large ratio of additions or subtractions, LNS is still not quite competitive with floating-point. The key problem is that LNS units for large bit widths generally require a high number of multipliers. This brings a large cost in either HMULs or slices (if we map the multiplier into logic cells), and makes it very difficult to map LNS adders or subtractors of large bit widths into an FPGA.

To solve this problem, we need to further optimise the current LNS arithmetic units in our exploration platform. Within the current LNS designs, one factor that we may further optimise is the implementation of multipliers. In the current platform, the multipliers are either implemented with HMULs or look-up tables. However, in most cases, the most efficient implementation is a mixture of HMULs and look-up tables. Thus, we can try to find out the optimal mixing mechanism of HMULs and look-up tables, so as to produce a multiplier design that consumes the least amount of HMULs and look-up tables. On the other hand, we can also try other entirely different algorithms or architectures that can bring further reduction to the current LNS addition and subtraction designs.

For RNS arithmetic (discussed in Chapter 5), one major problem we want to investigate is the range constraint. As RNS numbers are only determinable within one modular range, multiplication or addition of RNS numbers can easily cross the modular bound and produce meaningless results. In future, we plan to find out certain mechanism that provides a more dynamic range for RNS numbers while keeping the original properties of RNS arithmetic operations.

As part of this further optimisation, we also plan to extend the current arithmetic unit generation tools to produce design descriptions in VHDL. As VHDL is supported by more tools and frameworks, generation of VHDL arithmetic units makes it more convenient for users to apply

our arithmetic designs in their design. Additionally, VHDL provides a lower-level description of the circuit units, which is more suitable for optimisation of the arithmetic units at a bit level.

9.2.2 Using Multiple Number Representations in One Application

As all the different number representations have their advantages over each other, there is great potential to further improve the performance of the application by using multiple number representations, rather than using the same one for the entire application.

Suppose an application has a clear boundary for different kinds of arithmetic operations. E.g., before the bound, most of the operations are $+/-$, and after the bound, most of the operations are $\times/\div$. In this case, we can consider implementing the first part with floating-point numbers while implementing the second part with LNS numbers.

As shown in this simple example, to implement an application with multiple number representations, one major problem to solve is how do we locate the point where we convert from one number representation into another?

The plan is to apply an assembly-level code analyser to profile the program, so as to obtain useful information, such as number of $+, -, \times, \div$ operations and value range of the variables. Based on the characteristics of different code sections, we can then make initial guesses about turning points where we can switch into a different number representation. The initial guesses can be later refined and verified with our exploration platform.

In this way, we can gradually converge to a suitable arrangement of different number representations for different code sections, and achieve overall improvement on performance or reduction on cost.

9.2.3 Power Consumption of Different Number Representations

As the density of silicon circuit chips keeps increasing, the power consumption of circuit designs is becoming an important metric that is even more urgent to optimise than the area

consumption. High power consumption does not only bring problems for the energy side, but also produces heat that damages the system.

Similar to their effects on area consumption, different number representations can also bring big differences on power consumptions. Based on this consideration, we plan to extend the current exploration platform to investigate power consumption of different number representations, as well as area cost, accuracy, and performance.

The first step is to provide measurement approaches that can evaluate the power consumption of different designs. A straightforward method is to attach an ampmeter to the power supply jumpers of the FPGA, so as to measure the power consumption, as shown in the work by W.G. Osborne, et al. [OCLM08]. Other existing methods include a power model for different FPGA architectures [PWY05], and also a web-based power estimation tool provided by Xilinx [Inc], which is used to estimate the power consumption of fine-grained units based on the number of registers and look-up tables used in [HLLW08]. In our platform, we plan to provide a similar software tool that estimates the power consumption of the design based on the utilisation of different categories of resources on the FPGA and the frequency that the circuit runs at. An alternative is to develop a simple hardware device that reads the current of the circuit design and calculates the power consumption automatically.

With a tool to measure or estimate the power consumption of FPGA designs, the second step is to apply the tool to the arithmetic units of different number representations, collect results for different bit widths, and try to formulate power consumption models for different number representations. The model can then be used to estimate different designs that use different number representations to implement the same target application.

9.2.4 New FPGA Architectures

As mentioned in Section 9.2.1, the current LNS addition/subtraction units consume a large number of HMULs and BRAMs for high precision requirements. One method to reduce the consumption of resources is to develop new architectures or algorithms to implement these two

functions. The other method is to change the underlying FPGA architecture instead.

C. Ho, et al. [HLL+06, HYL+07, HLLW08] propose a hybrid FPGA architecture that includes coarse-grained units that are specifically designed for floating-point arithmetic, such as floating-point adders and multipliers. Using precise simulation tools, the authors show that such a new FPGA architecture with specialised floating-point arithmetic units can greatly improve the performance and reduce the area cost and power consumption of floating-point designs.

We plan to explore similar ideas for other number representations, such as LNS and RNS, so as to find out whether new FPGA architectures can further boost the benefits brought by application-specific number representations.

Bibliography

[ABCC90] M.G. Arnold, T.A. Bailey, J.R. Cowles, and J.J. Cupal. Redundant Logarithmic Arithmetic. *IEEE Trans. Comput.*, 39(8):1077–1086, August 1990.

[ACS94] M. V. A. Andrade, J. L. D. Comba, and Jorge Stolfi. Affine Arithmetic. In *Interval and Computer Algebraic Methods in Science and Engineering (INTERVAL'94), St. Petersburg (Russia)*, 1994. Available from http://graphics.stanford.edu/ comba/papers/aa-93-12-petersburg-paper.pdf.

[Alt06] Altera. *Stratix Device Handbook: Complete Two-volume Set*, January 2006. Available from http://www.altera.com.

[AM84] Guiseppe Alia and Enrico Martinelli. A VLSI Algorithm for Direct and Reverse Conversion from Weighted Binary Number System to Residue Number System. *IEEE Trans. Circuits Syst.*, 31(12):1033–1039, 1984.

[Arn01] M.G. Arnold. Design of a Faithful LNS Interpolator. In *Proc. Euromicro Symposium on Digital Systems Design (DSD)*, pages 237–246, 2001.

[Arn02] M.G. Arnold. Improved Cotransformation for LNS Subtraction. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 2, pages 752–755, 2002.

[Arn03] M.G. Arnold. Iterative Methods for Logarithmic Subtraction. In *Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 315–325, 2003.

[Arn05] M.G. Arnold. The Residue Logarithmic Number System: Theory and Implementation. In *Proc. IEEE Symposium on Computer Arithmetic (ARITH)*, pages 196–205, 2005.

[AW01a] M.G. Arnold and C. Walter. Unrestricted Faithful Rounding is Good Enough for Some LNS Application. In *Proc. IEEE Symposium on Computer Arithmetic (ARITH)*, pages 237–246, 2001.

[AW01b] M.G. Arnold and M.D. Winkel. A Single-Multiplier Quadratic Interpolator for LNS Arithmetic. In *Proc. IEEE International Conference on Computer Design (ICCD)*, pages 178–183, 2001.

[Beu03] Jean-Luc Beuchat. Some Modular Adders and Multipliers for Field Programmable Gate Arrays. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, page 190b, 2003.

[BFP+06] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto. Affinity-Driven System Design Exploration for Heterogeneous Multiprocessor SoC. *IEEE Trans. Comput.*, 55(5):508–519, 2006.

[BGWS00] M. Budiu, S.C. Goldstein, K. Walker, and M. Sakr. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Compilations. In *Proc. European Conference on Parallel Processing (Euro-Par)*, pages 969–979. Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2000.

[Bio03] B. Biondi. Narrow-azimuth Migration of Marine Streamer Data. In *Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts*, pages 897–900, 2003.

[BP96] B. Biondi and G. Palacharla. 3-D Prestack Migration of Common-Azimuth Data. *Geophysics*, 61:1822–1832, 1996.

[BP00] A. Benedetti and P. Perona. Bit-width Optimisation for Configurable DSP's by Multi-interval Analysis. In *Conference Record of the Asilomar Conference on Signals, Systems and Computers (ASILOMAR-SSC)*, volume 1, pages 355–359, 2000.

[CC03] C. Chen and R.L. Chen. Performance-Improved Computation of Very Large Word-Length LNS Addition/Subtraction Using Signed-Digit Arithmetic. In *Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 337–347, 2003.

[CCL04] George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk. *Synthesis and Optimization of DSP Algorithms*. Springer, 2004.

[CCSK00] J.N. Coleman, E.I. Chester, C.I. Softley, and J. Kadlec. Arithmetic on the European Logarithmic Microprocessor. *IEEE Trans. Comput.*, 49(7):702–715, July 2000.

[CCY00] C. Chen, R.L. Chen, and C.H. Yang. Pipelined Computation of Very Large Word-Length LNS Addition/Subtraction with Polynomial Hardware Cost. *IEEE Trans. Comput.*, 49(7):716–726, July 2000.

[Cel03] Celoxica. *Handel-C Language Reference Manual for DK2.0*, 2003. Available from http://www.celoxica.com.

[CG88] R.M. Capocelli and R. Giancarlo. Efficient VLSI Networks for Converting an Integer from Binary System to Residue Number System and Vice Versa. *IEEE Trans. Circuits Syst.*, 35(11):1425–1430, 1988.

[CH02] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[Cla99] J. Claerbout. *Geophysical estimation by example: Environmental soundings image enhancement: Stanford Exploration Project*. 1999. Available from http://sepwww.stanford.edu/sep/prof/.

[Cla00] J. Claerbout. *Basic Earth Imaging (BEI)*. 2000. Available from http://sepwww.stanford.edu/sep/prof/.

[CNPQ03] Mathieu Ciet, Michael Neve, Eric Peeters, and Jean-Jacques Quisquater. Parallel FPGA Implementation of RSA with Residue Number Systems. In *Proc. IEEE*

International Midwest Symposium on Circuits and Systems (MWSCAS), pages 806–810, 2003.

[Con06] G. A. Constantinides. Word-length Optimization for Differentiable Nonlinear Systems. *ACM Trans. Des. Autom. Electron. Syst.*, 11(1):26–43, 2006.

[CRNR04] G.C. Cardarilli, A. Del Re, A. Nannarelli, and M. Re. Low-Power Implementation of Polyphase Filters in Quadratic Residue Number System. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 2, pages 725–728, 2004.

[CS03] R. Charles and L. Sousa. RDSP: A RISC DSP Based on Residue Number System. In *Proc. Euromicro Symposium on Digital Systems Design (DSD)*, pages 128–135, 2003.

[CTLC05] R.C.C. Cheung, N. Telle, W. Luk, and P.Y.K. Cheung. Customizable Elliptic Curve Cryptosystems. *IEEE Trans. VLSI Syst.*, 13(9):1048–1059, 2005.

[CW02] George A. Constantinides and G. J. Woeginger. The Complexity of Multiple Wordlength Assignment. *Applied Mathematics Letters*, 15(2):137–140, 2002.

[DBS06] J. Deschamps, G. Bioul, and G. Sutter. *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*. Wiley-Interscience, 2006.

[DD07] J. Detrey and F. Dinechin. A Tool for Unbiased Comparison between Logarithmic and Floating-point Arithmetic. *The Journal of VLSI Signal Processing*, 49(1):161–175, October 2007.

[DdD03] J. Detrey and F. de Dinechin. A VHDL Library of LNS Operators. In *Conference Record of the Asilomar Conference on Signals, Systems and Computers (ASILOMAR-SSC)*, volume 2, pages 2227–2231, 2003.

[DIP93] G. Dimauro, S. Impedovo, and G. Pirlo. A New Technique for Fast Number Comparison in the Residue Number System. *IEEE Trans. Comput.*, 42(5):608–612, 1993.

[DMST08] M. Dasygenis, K. Mitroglou, D. Soudris, and A. Thanailakis. A Full-Adder-Based Methodology for the Design of Scaling Operation in Residue Number System. *IEEE Trans. Circuits Syst. I*, 55(2):546–558, 2008.

[FML07] H. Fu, O. Mencer, and W. Luk. Optimizing Logarithmic Arithmetic on FPGAs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 163–172, 2007.

[FO01] Michael Flynn and Stuart Oberman. *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.

[FRC03] C. Fang, R.A Rutenbar, and T. Chen. Fast, Accurate Static Analysis for Fixed-point Finite-precision Effects in DSP Designs. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 275–282, 2003.

[GCC06] A. A. Gaffar, J.A. Clarke, and G.A. Constantinides. PowerBit - Power Aware Arithmetic Bit-width Optimization. In *Proc. IEEE International Conference on Field-Programmable Technology (FPT)*, pages 289–292, 2006.

[GF91] Andreas Griewank and George F.Corlis, editors. *Automatic Differentiation of Algorithms: Theory, Implementation and Application*. Society for Industrial and Applied Mathematics, 1991.

[GFA+04] M. Gokhale, J. Frigo, C. Ahrens, J. Tripp, and R. Minnich. Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer. In *Proc. IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, pages 95–104, 2004.

[GL98] A. Garcia and A. Lloris. RNS Scaling based on Pipelined Multipliers for Prime Moduli. In *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, pages 459–468, 1998.

[GMLC04] A. A. Gaffar, O. Mencer, W. Luk, and P.Y.K. Cheung. Unifying Bit-width Optimisation for Fixed-point and Floating-point Designs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 79–88, 2004.

[GS85] J. Gazdag and P. Sguazzero. Migration of Seismic Data by Phase Shift plus Interpolation. In *G. H. F., Ed., Migration of seismic data: Society Of Exploration Geophysicists*, pages 323–330, 1985.

[GSAK00] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 49–58, 2000.

[GST89] Mike Griffin, Mike Sousa, and Fred Taylor. Efficient Scaling in the Residue Number System. In *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 2, pages 1075–1078, 1989.

[HBUH05] M. Haselman, M. Beauchamp, K. Underwood, and K.S. Hemmert. A Comparison of Floating Point and Logarithmic Number Systems for FPGAs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–190, 2005.

[HC94] S.C. Huang and L.G. Chen. The Chip Design of A 32-b Logarithmic Number System. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 167–170, 1994.

[HKB+08] A.H. Hormati, M. Kudlur, D. Bacon, S. Mahlke, and R. Rabbah. Optimus: Efficient Realization of Streaming Applications on FPGAs. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2008.

[HLL+06] C.H. Ho, P.H.W. Leong, W. Luk, S.J.E. Wilton, and S. Lopez-Buedo. Virtual Embedded Blocks: A Methodology for Evaluating Embedded Elements in FPGAs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 35–44, 2006.

[HLLW08] C.H. Ho, P.H.W. Leong, W. Luk, and S.J.E. Wilton. Rapid Estimation of Power Consumption for Hybrid FPGAs. In *Proc. IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, 2008.

[HLS02] C. He, M. Lu, and C. Sun. Accelerating Seismic Migration Using FPGA-based Coprocessor Platform. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 207–216, 2002.

[HNS+01] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, and P. Banerjee. FPGA Hardware Synthesis from MATLAB. In *Proc. International Conference on VLSI Design*, pages 299–304, 2001.

[Hua83] C.H. Huang. A Fully Parallel Mixed-Radix Conversion Algorithm for Residue Number Applications. *IEEE Trans. Comput.*, C-32(4):398–402, 1983.

[HYL+07] C.H. Ho, C.W. Yu, P.H.W. Leong, W. Luk, and S.J.E. Wilton. Domain-Specific Hybrid FPGA: Architecture and Floating-point Applications. In *Proc. IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, pages 196–201, 2007.

[IEE05] IEEE Computer Society. *IEEE Standard SystemC Language Reference Manual*, March 2005. Available from http://standards.ieee.org.

[Inc] Xilinx Inc. *Web Power Tool User Guide*. Available from http://www.xilinx.com.

[Inc03a] Xilinx Inc. *Development System Reference Guide*, April 2003. Available from http://www.xilinx.com.

[Inc03b] Xilinx Inc. *PowerPC 405 Processor Block Reference Guide*, October 2003. Available from http://www.xilinx.com.

[Inc04] Xilinx Inc. *Microblaze Processor Reference Guide*, June 2004. Available from http://www.xilinx.com.

[Inc05] Xilinx Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, March 2005. Available from http://www.xilinx.com.

[Inc07] Xilinx Inc. *XST User Guide*, December 2007. Available from http://www.xilinx.com.

[Ing04] L. Ingber. *Adaptive Simulated Annealing (ASA) 25.15*, 2004. Available from http://www.ingber.com.

[JLR03] J. Jiang, W. Luk, and D. Rueckert. FPGA-based Computation of Free-form Deformations in Medical Image Registration. In *Proc. IEEE International Conference on Field-Programmable Technology (FPT)*, pages 234–241, 2003.

[KKS00] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. AUTOSCALER for C: an Optimizing Floating-point to Integer C Program Converter for Fixed-point Digital Signal Processors. *IEEE Trans. Circuits Syst. II*, 47(9):840–848, 2000.

[KS99] J. Kadlec and J. Schier. Analysis of a Normalized QR Filter Using Bayesian Description of Propagated Data. *International Journal of Adaptive Control and Signal Processing*, 13(6):487–505, 1999.

[KS01] Ki-Il Kum and Wonyong Sung. Combined Word-length Optimization and High-level Synthesis of Digital Signal Processing Systems. *IEEE Trans. Computer-Aided Design*, 20(8):921–930, 2001.

[LA04] Lay Rong Lam and Tian Se Ang. *Fleeting Footsteps: Tracing the Conception of Arithmetic and Algebra in Ancient China*. World Scientific Publishing Company, 2004.

[LB03] B.R. Lee and N. Burgess. A Parallel Look-up Logarithmic Number System Addition/Subtraction Scheme for FPGA. In *Proc. IEEE International Conference on Field-Programmable Technology (FPT)*, pages 76–83, 2003.

[Lew93] D.M. Lewis. An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator. In *Proc. IEEE Symposium on Computer Arithmetic (ARITH)*, pages 2–9, 1993.

[LGC+06] D. Lee, A. Abdul Gaffar, R.C.C. Cheung, O. Mencer, W. Luk, and G.A. Constantinides. Accuracy Guaranteed Bit-Width Optimization. *IEEE Trans. Computer-Aided Design*, 25(10):1990–2000, November 2006.

[LGML05] D. Lee, A. Abdul Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Trans. Comput.*, 54(12):1520–1531, December 2005.

[LMY04] Miriam Leeser, Shawn Miller, and Haiqian Yu. Smart Camera Based on Reconfigurable Hardware Enables Diverse Real-Time Applications. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 147–155, 2004.

[Mat05] The MathWorks Inc. *MATLAB The Language of Technical Computing*, 2005. http://www.mathworks.com.

[Max] Maxeler Technologies, http://www.maxeler.com.

[MBS03] Uwe Meyer-Base and Thanos Stouraitis. New Power-of-2 RNS Scaling Scheme for Cell-Based IC Design. *IEEE Trans. VLSI Syst.*, 11(2):280–283, 2003.

[MDJM05] R. Muscedere, V. Dimitrov, G.A. Jullien, and W.C. Miller. Efficient Techniques for Binary-to-Multidigit Multidimensional Logarithmic Number System Conversion Using Range-Addressable Look-Up Tables. *IEEE Trans. Comput.*, 54(3):257–271, March 2005.

[Men06] O. Mencer. ASC, A Stream Compiler for Computing with FPGAs. *IEEE Trans. Computer-Aided Design*, 25(9):1603–1617, September 2006.

[Mit05] S. Mitra. *Digital Signal Processing: A Computer-based Approach*. McGraw-Hill, 2005.

[MJ95] A. Madisetti and A.N. Willson Jr. A 100 MHz 2-D 8×8 DCT/IDCT processor for HDTV applications. *IEEE Trans. Circuits Syst. Video Technol.*, 5(2):158–165, 1995.

[MM99] M.N. Mahesh and M. Mehendale. Improving Performance of High Precision Signal Processing Algorithms on Programmable DSPs. In *Proc. IEEE International Symposium on Circuits and Systems*, volume 3, pages 488–491, 1999.

[Moo66] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[MRS+01] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators. *IEEE Trans. Computer-Aided Design*, 20(11):1355–1371, November 2001.

[MSH99] Laurent Moll, Mark Shand, and Alan Heirich. Sepia: Scalable 3D Compositing Using PCI Pamette. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 146–155, 1999.

[MTP+02] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman. Logarithmic Number System and Floating-point Arithmetic on FPGA. In *Proc. IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, pages 627–636, 2002.

[Mul97] J. Muller. *Elementary functions: algorithms and implementation*. Birkhauser Boston, Inc., Secaucus, NJ, USA, 1997.

[NHCB01] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. Precision and Error Analysis of MATLAB Applications during Automated Hardware Synthesis for FPGAs. In *Proc. Design Automation and Test in Europe Conference (DATE)*, pages 722–728, 2001.

[OCLM08] W.G. Osborne, J.G.F. Coutinho, W. Luk, and O. Mencer. Reconfigurable Design with Clock Gating. In *Proc. Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2008.

[OPS95] I. Orginos, V. Paliouras, and T. Stouraitis. A Novel Algorithm for Multi-operand Logarithmic Number System Addition and Subtraction Using Polynomial Approximation. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1992–1995, 1995.

[otICS08] Standards Committee of the IEEE Computer Society. *IEEE standard for binary floating-point arithmetic*. 2008.

[Pal02] V. Paliouras. Optimization of LNS Operations for Embedded Signal Processing Applications. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 2, pages 744–747, 2002.

[PAL06] A.B. Premkumar, E.L. Ang, and E.M.K. Lai. Improved Memoryless RNS Forward Converter Based on the Periodicity of Residues. *IEEE Trans. Circuits Syst. II*, 53(2):133–137, 2006.

[PC07] O. Pell and R. G. Clapp. Accelerating Subsurface Offset Gathers for 3D Seismic Applications using FPGAs. In *Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts*, pages 2383–2387, 2007.

[PCGL04] Luis Parrilla, Encarnacion Castillo, Antonio Garcia, and Antonio Lloris. Intellectual Property Protection for RNS Circuits on FPGAs. In *Proc. IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, pages 1139–1141, 2004.

[Pie94] S.J. Piestrak. Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders. *IEEE Trans. Comput.*, 43(1):68–77, 1994.

[Pre02] A.B. Premkumar. A Formal Framework for Conversion From Binary to Residue Numbers. *IEEE Trans. Circuits Syst. II*, 49(2):135–144, 2002.

[PS96] V. Paliouras and T. Stouraitis. A Novel Algorithm for Accurate Logarithmic Number System Subtraction. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 268–271, 1996.

[PWY05] K.K.W. Poon, S.J.E. Wilton, and A. Yan. A Detailed Power Model for Field-Programmable Gate Arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 10(2):279–302, 2005.

[Raz94] Rahul Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, 1994.

[SB04] Changchun Shi and R.W. Brodersen. Floating-point to Fixed-point Conversion with Decision Errors due to Quantization. In *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 5, pages 41–44, 2004.

[SB06] G. Shan and B. Biondi. Imaging Steep Salt Flank with Plane-wave Migration in Tilted Coordinates. In *Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts*, pages 2372–2376, 2006.

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 108–120, 2000.

[SCE01] A. Skodras, C. Christopoulos, and T. Ebrahimi. The JPEG 2000 Still Image Compression Standard. *IEEE Signal Processing Magazine*, 18(5):36–58, 2001.

[SCNS83] E.E. Swartzlander, D.V.S. Chandra, H.T. Nagle, and S.A. Starks. Sign/Logarithm Arithmetic for FFT Implementation. *IEEE Trans. Comput.*, 32(6):526–534, 1983.

[SdF97] Jorge Stolfi and Luiz H. de Figueiredo. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monograph, IMPA, 1997.

[SDH03] B. So, P. Diniz, and M. Hall. Using Estimates from Behavioral Synthesis Tools in Compiler-Directed Design Space Exploration. In *Proc. Design Automation Conference (DAC)*, pages 514–519, 2003.

[SFMR06] A. Singhee, C. Fang, J. Ma, and R. Rutenbar. Probabilistic Interval-Valued Computation: Toward a Practical Surrogate for Statistics Inside CAD Tools. In *Proc. Design Automation Conference (DAC)*, pages 167–172, 2006.

[SHD02] B. So, M. Hall, and P. Diniz. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In *Proc. ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI), pages 165–176, 2002.

[SK89] M.A.P. Shenoy and R. Kumaresan. A Fast and Accurate RNS Scaling Technique for High Speed Signal Processing. *IEEE Trans. Acoust., Speech, Signal Processing*, 37(6):929–937, 1989.

[Str69] V. Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13:354–356, 1969.

[SunAD] Tzu Sun. *Sun Zi Suan Jing (The Mathematical Classic by Sun Zi)*. around 400 AD.

[Syn06] Synplicity. *Synplify Pro FPGA Solution Datasheet*, 2006. Available from http://www.synplicity.com.

[TCW+05] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung. Reconfigurable Computing: Architectures and Design Methods. *IEE Proceedings on Computers and Digital Techniques*, 152(2):193–207, 2005.

[TKA02] W. Thies, M. Karczmarek, and S.P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. International Conference on Compiler Construction (CC)*, pages 179–196, 2002.

[TMHG05] Justin L. Tripp, Henning S. Mortveit, Anders A. Hansson, and Maya Gokhale. Metropolitan road traffic simulation on fpgas. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 117–126, 2005.

[Tom06] Tadeusz Tomczak. Optimizing Residue Arithmetic on FPGAs. In *Proc. International Conference on Dependability of Computer Systems*, pages 297–305, 2006.

[Wan98] Yuke Wang. New Chinese Remainder Theorems. In *Conference Record of the Asilomar Conference on Signals, Systems and Computers (ASILOMAR-SSC)*, volume 1, pages 165–171, 1998.

[Wat] Waterloo Maple Inc. *Maple 9, for Academics*. http://www.maplesoft.com.

[WBGM97] M. Willems, V. Bursgens, T. Grotker, and H. Meyr. FRIDGE: An Interactive Code Generation Environment for HW/SW CoDesign. In *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 1, pages 287–290, 1997.

[WBK+97] Markus Willems, Volker Biirsgens, Holger Keding, Thorsten Grotker, and Heinrich Meyr. System Level Fixed-point Design based on an Interpolative Approach. In *Proc. Design Automation Conference (DAC)*, pages 293–298, 1997.

[WHYC06] K. Whitton, X.S. Hu, C.X. Yi, and D.Z. Chen. An FPGA Solution for Radiation Dose Calculation. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–236, 2006.

[WP98] S.A. Wadekar and A.C. Parker. Accuracy Sensitive Word-length Selection for Algorithm Optimization. In *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 54–61, 1998.

[WSAS02] Yuke Wang, Xiaoyu Song, Mostapha Aboulhamid, and Hong Shen. Adder Based Residue to Binary Number Converters for $(2^n - 1, 2^n, 2^n + 1)$. *IEEE Trans. Signal Processing*, 50(7):1772–1779, 2002.

[Xil04] Xilinx Inc. *Xilinx System Generator User Guide v6.3*, 2004. Available from http://www.xilinx.com.

[Xil07a] Xilinx, Inc., http://www.xilinx.com. *Virtex-4 Complete Data Sheet*, 2007.

[Xil07b] Xilinx, Inc., http://www.xilinx.com. *Virtex-II Complete Data Sheet*, 2007.

[ZC96] L. Zhao and A. C. Cangellaris. GT-PML: Generalized Theory of Perfectly Matched Layers and Its Application to the Reflectionless Truncation of Finite-Difference Time-Domain Grids. *IEEE Transactions on Microwave Theory and Techniques*, 44(12):2555–2563, 1996.