Hardware Acceleration for Elementary Functions and RISC-V Processor

Jing Chen

Doctor of Philosophy

School of Computer Science

McGill University

Montreal, Quebec

2020-05-15

A Thesis Submitted to the Faculty of Graduate Studies and Research in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

DEDICATION

To my beloved parents

ACKNOWLEDGMENTS

First and foremost, I would like to express the gratitude to my supervisor, Prof. Xue Liu, for giving me the freedom, which allowed me to choose the Ph.D. thesis topics and engage in the Ph.D. thesis research on my own, without outside interference.
Furthermore, I would also like to express the gratitude to my collaborator, Prof. Jason H. Anderson and his research team at University of Toronto, for his extensive knowledge and experience on field-programmable gate array (FPGA) and high-level synthesis (HLS), which have significantly motivated my current research.
I am very grateful for the precious chance to study and work at School of Computer Science in McGill University. I sincerely appreciated the scholarships and travel grants offered by my Department to support my Ph.D. studies, and academic activities financially. I would like to thank Prof. Luc Devroye for organizing my proposal exam defense, and Prof. Muthucumaru Maheswaran, Prof. Clark Verbrugge for serving on my comprehensive/proposal exams, and doctoral defense committees. I would also like to thank Prof. Brett Meyer from the ECE Department for his valuable comments to my thesis final revision. Moreover, I would like to thank my course instructors Prof. Bettina Kemme (Distributed Systems), and Prof. Doina Precup (Machine Learning) for the wonderful lectures they delivered. In addition, I received kindly help and support from staffs in my Department and the University, especially Ms. Ann Jack (Graduate Program Coordinator), Ms. Cheryl Bethelmy (Thesis and Graduation Administrator) and Prof. Robin Beech (Associate Dean).
I enjoyed the fun days spent together with my friends in McGill. I will remember the company they provided during my difficult times. In the end, I

would like to acknowledge my dear parents, for their love, understanding and never-ending support to my Ph.D. study, and decisions for personal life. It is a pity that I may not be able to list all persons who supported me during the past six years, I wish you all a wonderful future!

ABSTRACT

Many scientific applications rely on the evaluation of elementary transcendental functions (e.g. $\log(x)$, $\frac{1}{x}$, $\sqrt{x}$, e^x). Software math libraries are a popular approach for realizing such functions, and frequently use series expansion and/or lookup-table-based (LUT-based) methods. However, software approaches necessarily suffer from the traditional overheads of fetching/decoding instructions, limited cache sizes, and so on. To this end, we present hardware accelerators for such functions that deliver high computational throughput, high accuracy and a small circuit.

We implement the reciprocal ($\frac{1}{x}$) and square root ($\sqrt{x}$) functions into pipelined hardware accelerators on FPGA. The proposed accelerators are designed with iterative, and LUT-based algorithms. Here, the LUT-based algorithm uses approximately 1 KB LUT along with a degree-2 polynomial interpolation. All algorithms are specified using C language, and synthesized into RTL with the LegUp HLS [5]. In an experimental study, we compare our LUT-based accelerators with IP cores from the Intel/Altera FPGA vendor. Results show that our LUT-based accelerators offer considerably better area usage, while Intel/Altera IP cores operate at a modestly higher throughput. Both ours and Intel/Altera IP cores achieve 1 ULP error. The LUT-based algorithm is generic in the sense that it could be used to implement an entire library of single-precision elementary functions into high-performance hardware accelerators.

We also implement a 32-bit integer RISC-V [32] multi-cycle processor on FPGA, which consists of 39 user instructions. The processor is specified using C language, and synthesized into RTL with the LegUp HLS [5]. Custom testing programs are developed to verify if each instruction adheres to the RISC-V specification. We demonstrate that through changes to the `C` specification and HLS constraints, RISC-V processors with different performance/area trade-offs can be explored rapidly. One implementation of the multi-cycle processor uses 795 ALMs (adaptive logic modules) on an Intel/Altera Cyclone V FPGA, and is operable at 124.1 MHz. We believe that, in the future, integrating high-performance elementary transcendental function accelerators into a RISC-V soft processor on FPGAs may bring significant performance benefits to accelerate compute-intensive applications.

ABRÉGÉ

De nombreuses applications scientifiques reposent sur l'évaluation des fonctions transcendantales élémentaires (par exemple $\log(x)$, $\frac{1}{x}$, $\sqrt{x}$, e^x). Les bibliothèques mathématiques de logiciels sont approche populaire pour réaliser de telles fonctions et utiliser fréquemment des méthodes d'extension de série et/ou basées sur des tables de recherche (basées sur LUT). Cependant, les approches logicielles souffrent nécessairement des frais généraux traditionnels de récupération / décodage instructions, tailles de cache limitées, etc. À cette fin, nous présentons le matériel accélérateur pour de telles fonctions qui fournissent un débit de calcul élevé, haut précision et un petit circuit.

Nous mettons en œuvre la réciproque ($\frac{1}{x}$) et racine carrée ($\sqrt{x}$) fonctionne en accélérateurs matériels pipelinés sur FPGA. Les accélérateurs proposés sont conçus avec des algorithmes itératifs et basés sur LUT. Ici, l'algorithme basé sur LUT utilise environ 1 KB LUT avec une interpolation polynomiale de degré 2. Tous les algorithmes sont spécifiés en langage C et synthétisés en RTL avec LegUp HLS [5]. Dans une étude expérimentale, nous comparons nos Accélérateurs basés sur LUT avec cœurs IP du fournisseur Intel / Altera FPGA. Les résultats montrent que nos accélérateurs basés sur LUT offrent une surface considérablement meilleure tandis que les cœurs IP Intel / Altera fonctionnent à un débit légèrement supérieur. Les cœurs IP nôtres et Intel / Altera obtiennent 1 erreur ULP. Le LUT algorithme est générique dans le sens où il pourrait être utilisé pour mettre en œuvre un ensemble bibliothèque de fonctions élémentaires simple précision en accélérateurs matériels haute performance.

Nous implémentons également un processeur multi-cycle RISC-V [32] 32 bits sur FPGA, qui se compose de 39 instructions d'utilisation. Le processeur est spécifié utilisant le langage C, et synthétisé en RTL avec LegUp HLS [5]. Des programmes de tests personnalisés sont développés pour vérifier si chaque instruction respecte la spécification RISC-V. Nous démontrons que grâce à des modifications du spécifications et contraintes HLS, les processeurs RISC-V avec différents compromis performances / zone peuvent être explorés rapidement. Une mise en œuvre du processeur multi-cycle utilise 795 ALM (modules logiques adaptatifs) sur un FPGA Intel / Altera Cyclone V et fonctionne à 124,1 MHz. Nous pensons qu'en l'avenir, intégrant la fonction transcendantale élémentaire à haute performance accélérateurs dans un processeur logiciel RISC-V sur FPGA peut apporter des des avantages en termes de performances pour accélérer les applications exigeantes en calcul.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Acronyms

ASIC	Application Specific Integrated Circuit. 23
AXI4	Advanced extensible Interface 4. 88
CORDIC	Coordinate Rotation Digital Computer. 64
DMIPS	Dhrystone Million Instructions executed per Second. 89
DRAM	Dynamic Random Access Memory. 85
eALM	effective Adaptive Logic Module. 47
FPGA	Field Programmable Gate Array. 3
GB	Giga Byte. 18
GCC	GNU Compiler Collection. 84
GPU	Graphical Processing Unit. 29
HDL	Hardware Description Level. 27
HLS	High Level Synthesis. 10
HMMs	Hidden Markov Models. 12
IC	Integrated Circuit. 25
IDE	Integrated Development Environment. 84
IEEE	Institute of Electrical and Electronics Engineers. 10
ILP	Instruction Level Parallelism. 89
IR	Intermediate Representation. 29
LLVM	Low Level Virtual Machine. 29, 84
LSB	Least Significant Bit. 41

LUT Look-up Table. 2

MMU Memory Management Unit. 90

MSB Most Significant Bit. 43

NOC Network of Chip. 88

OS Operating System. 89

QEMU Quick Emulator. 84

RAM Random Access Memory. 23

RoCC Rocket coprocessor interface. 88

RTL Register Transfer Level. 10

SAR Synthetic Aperture Radar. 65

SIMD Single Instruction Multiple Data. 30

TLB Translation Lookaside Buffer. 90

VHDL Very High-speed Hardware Description Language. 27

Glossary

Dhrystone	An integer computing benchmark. 89
Intel/Altera	A leading commercial FPGA vendor. 39
LegUp	A HLS startup at University of Toronto. 46
NVIDIA	A leading commercial GPU vendor. 83
Quartus	A FPGA design software developed by Intel/Altera. 47
Verilog	A hardware description language. 27
Xilinx	A leading commercial FPGA vendor. 39

Chapter 1
Introduction

Many scientific computing applications require frequent evaluation of elementary transcendental functions, such as logarithms ($\log x$), exponents (e^x), reciprocals ($\frac{1}{x}$) and, square roots ($\sqrt{x}$), etc. In the past, researchers implemented such functions within software math libraries (e.g. libm C99 [3]), typically by using a series approximation (e.g. a Taylor series). To produce accurate approximations, high-order polynomials are necessary, which require numerous addition and multiplication operations. Thus, series approximations may lead to lengthy execution times when high accuracy is a paramount goal.

In order to speed up function evaluation, researchers started to implement such functions using dedicated hardware accelerators [118, 120, 119, 59]. In the 1980s, the Intel 8087 became the first math co-processor to compute elementary transcendental functions [97]. However, its performance was less than satisfactory since the math functions on the co-processor were executed using micro-code: essentially another software implementation. Subsequently, as memory became inexpensive, the capacity of memory in computer systems increased dramatically [27]. Thanks to such memory improvements, look-up table (LUT) approximation has been extensively used by the software math libraries, which makes functions evaluation considerably faster and more accurate [107, 108]. Using this method, the function values have been pre-computed and stored in a LUT. Since a large LUT is likely to cause frequent data swapping between the cache and the main memory (so-called cache thrashing [99]), the LUT approach may still suffer from lengthy evaluation time.

Since the mid-to-late 2000s, CPU clock frequency scaling plateaued at around 3 GHz due to excessive power density [104]. As a result, it was not practical to accelerate software math library merely via CPU frequency scaling. In order to accommodate the growing needs of high-performance computing, researchers returned to the study of hardware acceleration of elementary transcendental functions. With the fast growth of field-programmable gate array (FPGA) technology, FPGA-based math function accelerators show numerous advantages over the software equivalents [57, 58]. As opposed to the design of the Intel 8087 math co-processor, the core algorithm of function evaluation has been implemented as hardware circuits instead of using the micro-instructions of the CPU. With such application-specific hardware, tailored specifically to the task, hardware math accelerators will have great potential to outperform software math libraries.

Field-programmable gate arrays (FPGAs) are integrated circuits that are configurable in the “field” by the end user to implement any digital circuit. FPGA chips are a \$7 billion US dollars industry today, and they are widely used in industrial, communications, consumer, and other applications. The advantage of an FPGA over a custom ASIC (whose function is fixed at the time of manufacture) is that an FPGA can be configured in seconds, and can later be reconfigured, to implement a different application or to fix bugs, etc. A recent development in the FPGA domain is that multiple cloud vendors, including Microsoft, Huawei, and Amazon, have deployed FPGAs into their clouds, where they are used to implement accelerators, working alongside standard processors. One can now rent time on a cloud-deployed FPGA, and use it for application-specific acceleration, from anywhere in the world. Further details on FPGA technology are provided in the background Chapter 2.

To use an FPGA, one has generally needed to have strong skills in computer hardware design. Circuit design for FPGAs involved the use of hardware description languages (HDLs), primarily Verilog or VHDL, which require the desired circuit behaviour to be specified at a very low level of abstraction. An HDL is a textual/software description of a hardware circuit, that describes hardware behaviours. In an HDL, one describes the hardware functionality at the register transfer level (RTL), which requires a designer to specify the circuit behaviour in each hardware clock cycle. While software programs can be compiled and executed to verify functionality, with an HDL, one must simulate the behaviour of the specified hardware to verify correctness, using a logic simulator such as ModelSim.

Recently, high-level synthesis (HLS) hardware design methodologies have gained popularity as an alternative to using HDLs and the RTL level of design abstraction. With HLS, one writes a `C/C++` software program to specify the desired behaviour. The program is then compiled (e.g. with `gcc`) and run on a standard microprocessor to verify its functionality and correctness. Following this, one uses an HLS tool to "compile" the `C/C++` specification automatically into a hardware circuit in RTL. HLS allows hardware design to happen at a higher level of abstraction, and makes hardware accessible to those with solely software skills. The combination of FPGAs and HLS is quite powerful for the implementation of hardware accelerators. We elaborate on digital circuit design methodologies further in Chapter 2.

In the following sections of this chapter, in section 1.1, clarification of research contribution is presented. The organization of the remainder of the thesis are summarized in section 1.2.

1.1 Clarification of Research Contribution

The purpose of writing this section is to avoid the potential intellectual property disputes arose from my Ph.D. thesis research in the future. To this end, I clarify my personal contribution from two other professors (Prof. Xue Liu, Prof. Jason H. Anderson) for the research presented in my Ph.D. thesis. During my Ph.D. study, I mainly have accomplished three research packages:

1. **Design and implementation of the reciprocal accelerator**.
2. **Design and implementation of the square root accelerator**.
3. **Design and implementation of the RISC-V soft processor**.

I am grateful to my supervisor, Prof. Xue Liu, for giving me the freedom, which allowed me to choose my Ph.D. thesis topics and conduct my Ph.D. thesis research on my own, without outside interference. Prof. Liu did not offer any guidance or help to: 1) my Ph.D. thesis topics selection, and 2) my Ph.D. thesis research. Since his research directions are different from mine, and he is not optimistic about my Ph.D. thesis research topics.

1.1.1 My Personal Contributions to the Packages from 1 to 3

1. **Design and implementation of the reciprocal accelerator**. The detailed contributions are summarized as follows:
   (a) I design the algorithm of a single-precision (32-bit) reciprocal accelerator, which is based on look-up table (LUT $\approx$ 1 KB) and degree-2 polynomial interpolation. The algorithm is implemented using C language.
   (b) I implement a single-precision (32-bit) reciprocal accelerator using C language, which is based on the trial-subtraction.
   (c) By treating reciprocal results generated by the GNU math.h library as a golden benchmark, our LUT-based and trial-subtraction accelerators are compared with the benchmark via exhaustive testing, to

generate error distribution report. It shows both LUT-based and trial-subtraction accelerators have 1 ULP maximum error.

   (d) Besides 1 ULP precision accelerators, I implement reduced-precision reciprocal accelerators using C language, for applications which are more critical on speed and circuit area.

   (e) The LegUp HLS is applied to interpret the reciprocal accelerators specified in C software to Verilog implementations.

   (f) The Quartus and ModelSim are used to synthesize the Verilog implementations into hardware targeting Intel/Altera Cycle V 45nm FPGA.

   I finished tasks (a), (b), (c), (d) only by myself. Prof. Jason H. Anderson helped on task (e). The LegUp HLS tool [5] is provided by Prof. Anderson's team at University of Toronto.

2. **Design and implementation of the square root accelerator**. The detailed contributions are summarized as follows:

   (a) I design the algorithm of a single-precision (32-bit) square root accelerator, which is based on look-up table (LUT $\approx$ 1 KB) and degree-2 polynomial interpolation. The algorithm is implemented using C language.

   (b) I implement a single-precision (32-bit) square root accelerator using C language, which is based on Newton's method.

   (c) By treating square root results generated by the GNU math.h library as a golden benchmark, our LUT-based and Newton's accelerators are compared with the benchmark via exhaustive testing, to generate error distribution report. It shows both LUT-based and Newton's accelerators have 1 ULP maximum error.

   (d) The LegUp HLS is applied to interpret the square root accelerators specified in C software to Verilog implementations.
   (e) The Quartus and ModelSim are used to synthesize the Verilog implementations into hardware targeting Intel/Altera Cycle V 45nm FPGA.

   I finished tasks (a), (b), (c) only by myself. Prof. Jason H. Anderson helped on task (d). The LegUp HLS tool [5] is provided by Prof. Anderson's team at University of Toronto.

3. **Design and implementation of the RISC-V soft processor**. The detailed contributions are summarized as follows:
   (a) I read and understand the specification of the 32-bit RISC-V integer instruction set (RV32I), which mainly focuses on the function description of the instructions.
   (b) I implement a RV32I multi-cycle processor using C language, which fully realizes the logical and structural functions of the 39 user instructions.
   (c) I test each instruction in the RV32I instruction set to make sure it executes correctly and adheres to the RISC-V specification.
   (d) Custom programs are also created by using RV32I instructions for further testing, which include:
      i. Manually created testing programs in RV32I assembly code.
      ii. Testing programs generated from GCC toolflow.
   (e) The LegUp HLS is applied to interpret the RV32I implementation specified in C software to a multi-cycle processor in Verilog.
   (f) The Quartus and ModelSim are used to synthesize the processor implementation specified in Verilog into hardware targeting Intel/Altera Cycle V 28nm FPGA.

I finished tasks (a), (b), (c) and the part (i) of task (d) only by myself. Prof. Jason H. Anderson helped on the part (ii) of task (d) and task (e). The LegUp HLS tool [5] is provided by Prof. Anderson's team at University of Toronto.

1.1.2 Algorithm Improvements Compared to Master Thesis

The LUT-based algorithms used to implement reciprocal and square root accelerators are developed based on the algorithm of implementing the logarithm in my master thesis [48]. Since the algorithm used by the logarithm is universal, it could be applied to implement an entire library of single-precision, floating-point elementary functions. Compared to the algorithm presented in my master thesis, there are four significant innovations in our new LUT-based algorithm:

1. The LUT compression technique has been improved, so that the LUT is 6-8× smaller without harming accuracy.
2. Worst accuracy of the accelerator has been improved to 1 ULP from 3 ULP.
3. The computation of the degree-2 polynomial interpolation has been reduced to two multiplications and two additions from three multiplications and two additions.
4. The internal computation format changes from ≈70 bits to 32 bits.

1.2 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 begins with a background on the frequently-used elementary transcendental functions and their applications, IEEE-754 single-precision floating-point representation, ULP (unit in the last place), LUT-based function evaluation method, FPGA technology, and digital circuit design methodology (RTL, HLS). Chapters 3 and 4 elaborate on the design and implementation of the reciprocal and square root hardware accelerators, respectively. The algorithm design, accuracy, speed and circuit area of the proposed accelerators are addressed. Chapter 5 presents the design and implementation of the 32-bit RISC-V soft processor. First, we give a brief introduction about the RISC-V ISA. Then, the architecture of the processor, and the testing programs are proposed. In evaluation, we compare our processor with two open source RISC-V implementations in terms of speed and area on FPGA. Finally, conclusions and suggestions for future work are provided in Chapter 6. Seven manually created testing programs for the RISC-V processor are listed in the Appendix.

Chapter 2
Background

In this chapter, we provide background on elementary transcendental functions, FPGA technology, and digital circuit design methodologies. In Sections 2.1 and 2.2, definitions and applications of elementary transcendental functions are presented. A complete list of frequently-used functions is shown in Table 2–1. In Sections 2.3 and 2.4, we introduce the IEEE-754 floating-point standard and ULP (unit in the last place) metric, respectively. In Section 2.5, common approaches which have been extensively used to implement elementary transcendental functions are presented. We further explain the pros and cons of such approaches. Then, we propose several important properties to measure the performance of such functions in section 2.6, and discuss the relationships and various trade-offs made between those properties in section 2.7.

The concept of modern FPGA technology is presented in Section 2.8, as FPGAs are the hardware platform we use for several accelerator implementations. We introduce the major features and general composition of the FPGA. Moreover, the architecture of an FPGA and its role in cloud computing are noted. In Section 2.9, we review the evolution of the methods used to design and verify hardware. That leads us to take a look at the various abstraction levels of FPGA programming, which mainly include register-transfer level (RTL) and high-level synthesis (HLS) design. We further elaborate on the HLS technique, which considerably improves the productivity of hardware design. The concept of heterogeneous computing is discussed in Section 2.10. In this context, computation units in various architectures work collaboratively to accelerate a broader range of applications.

2.1 What Are Elementary Transcendental Functions?

In mathematics, a transcendental function is a function which cannot be expressed in terms of a finite composition of the algebraic operations of addition, subtraction, multiplication, division, raising to a power and root extraction [33, 92, 77, 98]. Examples of transcendental functions include, but are not limited to, logarithmic ($\log x$), exponential (e^x) and trigonometric ($\sin x$) functions. An elementary function is a one-variable function that is composed of a finite number of algebraic, logarithmic, exponential or, trigonometric functions through combinations using addition, subtraction, multiplication or division [25, 92, 77, 98]. Examples of elementary functions are power (x^2), reciprocal ($\frac{1}{x}$), square root ($\sqrt{x}$), and so on. To help readers get a general idea, the frequently-used elementary transcendental functions (or elementary functions, for short) are shown in Table 2–1. In this thesis, we implement two frequently-used elementary functions into hardware accelerators, which are reciprocal and square root.

Table 2–1: Frequently-used Elementary Transcendental Functions

powers	x, x^2, x^3
reciprocal	$\frac{1}{x}$
roots	$\sqrt{x}, \sqrt[3]{x}$
exponential	e^x
logarithm	$\log x$
trigonometric functions	$sin(x), cos(x), tan(x)$, etc.
inverse trigonometric functions	$arcsin(x), arccos(x), arctan(x)$, etc.
hyperbolic functions	$sinh(x), cosh(x)$, etc.

2.2 Why Are Elementary Functions Important?

Modern computer systems that support floating-point computations usually require floating-point mathematical libraries (e.g. `math.h`). The libraries include a large collection of elementary functions, such as exponents, logarithms, powers, and roots. Numerous scientific applications rely on the evaluation of elementary functions, so there are both `software` [7, 8] and `hardware` [2] implementations of these functions. The applications of logarithmic, reciprocal, square root and exponential functions are summarized below:

- **The logarithmic function** has been used to compute log-likelihood in Gaussian mixture models for multimedia applications. For example, a machine learning-based speech recognition system, the ICSI speaker engine [114], spends 80% of its run time on computing log-likelihood. Furthermore, machine learning algorithms compute logarithm to simplify their evaluation of loss functions. For example, HMMs (hidden Markov models) compute logarithm to transform products of loss functions into sums to reduce cost and retain accuracy. In addition, logarithm and exponent are also frequently-used by web search and data analytics applications in data centres [61, 79].
- **The reciprocal and square root functions** have been extensively used in image and digital single processing applications. In particular, matrix decomposition, such as QR and LU algorithms, rely on evaluations of reciprocal and square root [91]. Moreover, they are also important for MIMO (Multiple-Input, Multiple Output) wireless communication systems [91]. In addition, square root is heavily used in solving stochastic problems, probabilistic calculations, and numerical approximation [47].
- **The exponential function** is a primary function used by scientists. Most biological systems (e.g. population growth/decline) and investment strategies can be modelled using exponential functions [35]. In machine learning applications, the exponential function has been used to evaluate

the sigmoid activation function of neural networks, thanks to the non-linear and differentiable properties of exponents [51]. Furthermore, the exponential function is used for maximum entropy classifiers since it can ensure a global minimum for the error functions [88].

2.3 IEEE-754 Floating-Point Standard

There were numerous floating-point formats at the start of the computer era. The IEEE-754 floating-point standard was widely adopted in 1985. It describes floating-point representations, arithmetic, rounding rules, exception handling, etc. In [52], the 2008 revision of the standard, specifications of frequently-used elementary functions are included. Figure 2–1 illustrates how single- (32-bits) and double- (64-bits) precision floating-point numbers are represented by the IEEE-754 2008 standard.

A floating-point number contains three parts: which are the sign, exponent and mantissa. Single-precision has a 1-bit sign, 8-bit exponent and 23-bit mantissa. There is a radix point between the 22*nd* and 23*rd* bit. A hidden bit before the radix point can be either bit-0 or bit-1, depending on the exponent. Similarly, double-precision has a 1-bit sign, 11-bit exponent and 52-bit mantissa. The radix point stands between the 51*st* and 52*nd* bit. In [52], the IEEE-754 2008 standard also added 128-bit and 256-bit representations, which are called quadruple and octuple-precision, respectively. However, they are not widely supported by hardware yet.

According to the IEEE-754 standard, the decimal value of a single-precision floating-point number *fp* can be represented by the sign and the product of the factor $2^{e_{7-0}-127}$ and the mantissa $1.m_{22-0}$, as shown by equation 2.1:

$$fp = (-1)^s \times 2^{e_{7-0}-127} \times 1.m_{22-0} \tag{2.1}$$

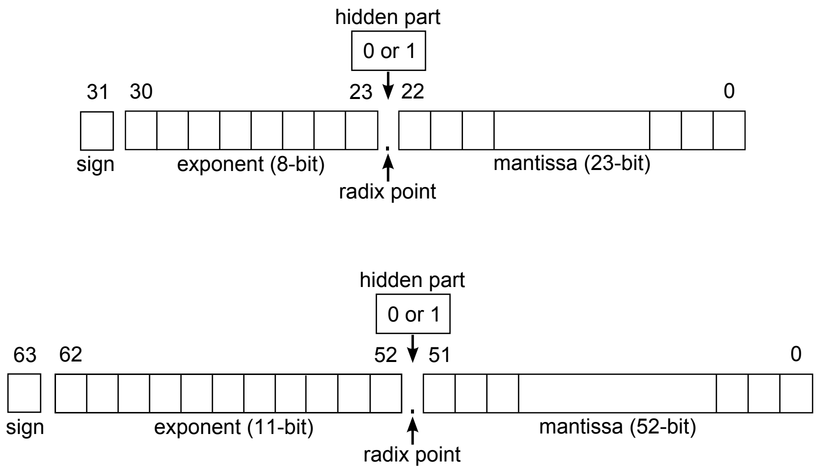

Figure 2–1: IEEE-754 single and double-precision floating-point formats [48]

In this case, sign s is either positive or negative. The value that *fp* represents can fall into one of the five categories presented in Table 2–2:

- **zero**: If the exponent and mantissa are $0x00$ and $0x0000$, respectively, then *fp* represents $zero$.
- **subnormal**: If the exponent is $0x00$, but the mantissa is not $0x00$, then the exponent represents -126, and the hidden bit before radix point is bit-0, so *fp* represents $\pm 2^{-126} \times 0.m$ (values very close to $zero$).
- **normalized**: If the biased exponent e is within the interval ($0x00$, *0xff*), then the real exponent is $e - 127$ (need to subtract the bias of 127), and the hidden bit before the radix point is bit-1, so *fp* represents $\pm 2^{e-127} * 1.m$.
- **infinity**: If the exponent and mantissa are *0xff* and $0x0000$, respectively, then *fp* represents $Infinity$.
- **not-a-number**: If the exponent is *0xff*, and the mantissa is not $0x0000$, then *fp* represents Not-a-Number.

Table 2–2: Represented Values for Single-Precision Numbers

Category	Sign	Exponent	Mantissa	Value
zero	0 or 1	e=0x00	m=0x0000	± 0.0
subnormal	0 or 1	e=0x00	m≠0x0000	$\pm 2^{-126} \times 0.m$
normalized	0 or 1	0x00<e<*0xff*	random	$\pm 2^{e-127} \times 1.m$
infinity	0 or 1	e=*0xff*	m=0x0000	±inf
not-a-number	0 or 1	e=*0xff*	m≠0x0000	±NaN

In Table 2–2, we observe that the arithmetic of double-precision representation is more complicated than that of single-precision for the following reasons: 1) double-precision has higher precision, it uses 52-bits mantissa, so double-precision functions need wider bit-width to perform internal computations to retain the precision; 2) double-precision consists of more floating-point values on its domain. For example, it has 2^{63} positive floating-point numbers, which is 2^{32} times more than that of single-precision. Hence, the workload of doing exhaustive testing on double-precision functions is much heavier; 3) the primitive LUT of double-precision $\log x$ (without any errors) is approximately 32,768 TB. A table of this size can hardly fit into the disk of current computer systems. After realizing the complexity of the double-precision arithmetic, we prefer to begin with the single-precision implementation of elementary functions. In fact, a large number of double-precision math function libraries in software use single-precision computing engines as their kernels.

2.4 Unit in the Last Place (ULP)

In numerical analysis, unit in the last place (ULP) [67] is widely used as a metric to measure error for floating-point arithmetic. It represents the space between floating-point numbers as defined by the IEEE-754 standard. For example, 1 ULP represents the space between two adjacent floating-point numbers. Table 2–3 shows two 1 ULP testing cases for reciprocal and square root, respectively. In the first case, x_1 represents a single-precision input of reciprocal, GNUrecip(x_1) represents the reciprocal evaluation by the GNU `math.h` library, and MYrecip(x_1) represents our hardware implementation of reciprocal. In the second case, x_2 represents a single-precision input of square root, GNUsqrt(x_2) represents the square root evaluation by the GNU `math.h` library, and MYsqrt(x_2) represents our hardware implementation of square root. In this case, MYrecip(x_1) has a 1 ULP error with respect to GNUrecip(x_1) since they have the same exponent and the difference of the mantissas is 0.0000...01. Similarly, MYsqrt(x_2) also has a 1 ULP error with respect to GNUsqrt(x_2) when applying the same rule. When we say that the maximum error of either a reciprocal or square root is 1 ULP in later sections, it implies that our interpolated evaluations have at most 1 ULP error with respect to evaluations from the GNU `math.h` library. Moreover, as will be demonstrated, for a large percentage of inputs, our hardware accelerators exhibit no error.

Table 2–3: Cases with 1 ULP Error

Case 1	Sign	Exponent	Mantissa
x_1	0	00000001	00000010000100111101000
GNUrecip(x_1)	0	11111100	11111011111000001111000
MYrecip(x_1)	0	11111100	11111011111000001110111
Case 2	Sign	Exponent	Mantissa
x_2	0	00000001	00001001000101100011101
GNUsqrt(x_2)	0	01000000	00000100100000001111100
MYsqrt(x_2)	0	01000000	00000100100000001111101

2.5 Prevalent Approaches

Series expansion and/or look-up tables (LUTs) have been the prevalent approaches to realizing elementary functions in both `software` and `hardware` math libraries. In this subsection, we briefly introduce the two approaches to help readers get a general idea of how elementary functions have been evaluated. Then, we discuss the pros and cons of such approaches, and propose a combined approach for our functions evaluation.

Series expansion, such as Taylor series [92] ($\sum_{n=0}^{\infty} \frac{f^{(n)}(a)\,(x-a)^n}{n!}$), can produce highly accurate approximations of elementary functions without using excessive memory resources. The Maclaurin series expansion of the natural exponential function (e^x) can be expressed as the sum of a degree-n polynomial and a remainder, the so-called $R_n(x)$. $R_n(x)$ represents the error of the polynomial approximation with respect to the real value of e^x. With enough terms in the series, the Maclaurin approximation will be highly accurate, as the remainder $R_n(x)$ becomes extremely close to zero.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + R_n(x) \tag{2.2}$$

To produce an accurate approximation, a high-order polynomial is usually adopted. Such polynomial approximation may lead to lengthy evaluation time in both software and hardware, due to the many additions and multiplications required. A wide bit-width has sometimes been used to store the accumulated products generated from computing the polynomial. In addition, choosing a correct rounding scheme is important for retaining accuracy.

A look-up table (LUT) can also generate accurate evaluations of elementary functions. In this case, function values have been pre-computed and stored in a LUT. Hence, each tabulated value corresponds to an approximation. Evaluating a function is equivalent to searching for and retrieving certain entries from the LUT. In doing so, it considerably reduces run-time computations versus a series expansion approach. A LUT-based approach has the potential to outperform series expansion in terms of execution time; however, it may suffer from cache thrashing problems in a computer system. Here, "cache thrashing" refers to the frequent data swapping between the cache and the main memory. It often happens when the program, data and LUTs are competing for limited cache resources at run-time. Cache-thrashing effects may worsen when more than one elementary function is being evaluated at the same time. Generally speaking, the accuracy of the LUT-based approach is proportionate to the LUT size. The more entries a LUT has, the higher the accuracy that has been evaluated for an elementary function.

A conjunction of look-up table (LUT) and polynomial is applied to implement reciprocal and square root functions. In this case, only a fraction of the function values have been pre-computed and stored in the LUT. The missing function values are interpolated using relatively low-order polynomials. The combined method has the potential to reduce LUT size, as well as save run-time computations. To explain the method, we briefly introduce the algorithm of LUT along with degree-2 polynomial interpolation used by the logarithm accelerator presented in my master thesis [48]. Since the algorithm of implementing logarithm is universal, it could be used to implement an entire library of single-precision, floating-point elementary functions.

A full-precision LUT for $\log(x)$ is 8 GB since it has $2^8 \times 2^{23} = 2G$ entries, and each entry uses four bytes. Using the ICSILog algorithm [114], the LUT is shrunk to 32 MB. The ICSILog algorithm takes advantage of logarithmic product and power rules, so $log(+fp)$ is transformed into:

$$\begin{aligned} \log(+fp) &= \log(2^{e_{7-0}-127} \times 1.m_{22-0}) && (2.3) \\ &= \log(2^{e_{7-0}-127}) + \log(1.m_{22-0}) && (2.4) \\ &= e_{7-0} - 127 + \log(1.m_{22-0}) && (2.5) \end{aligned}$$

Eqns. 2.3 to 2.5 transform the evaluation of $\log(+fp)$ to $\log(1.m_{22-0})$. At the same time, they shrink the domain of $\log(+fp)$ on $(0, +\infty)$ to $\log(1.m_{22-0})$ on [1,2). A LUT for $\log(1.m_{22-0})$ has 2^{23} entries, and each entry uses four bytes, making it 32 MB in total.

To further shrink the 32 MB full-precision LUT of $\log(x)$ for better performance, a degree-2 polynomial interpolation is applied. We keep only part of the entries in the 32 MB LUT and use a parabola to interpolate the "missing" entries. In this case, not all 23-bits mantissa for $log(1.m_{22-0})$ have been used as an index to access the new smaller LUT. Table 2–4 demonstrates an 8 KB LUT, which uses the highest 11-bits of the mantissa for table retrieval. In this scenario, there are only 2^{11} entries, so it is significantly smaller than the 32 MB LUT, which takes all 23-bit of mantissa as an index. Based on the Table 2–4, another new LUT to permit a degree-2 polynomial interpolation for $\log(1.m_{22-0})$ is created, as shown by Table 2–5.

Table 2–4: 8 KB LUT with high 11-bits as index [49]

Entry	Index (11-bits)	Content (32-bits)
E_0	$\underbrace{00000...00000}_{11}$	$\log(1.\underbrace{00000...00000}_{11}\underbrace{00000...00000}_{12})$
E_1	00000...00001	log(1.00000...0000100000...00000)
E_2	00000...00010	log(1.00000...0001000000...00000)
...	...	...
E_{2045}	11111...11101	log(1.11111...1110100000...00000)
E_{2046}	11111...11110	log(1.11111...1111000000...00000)
E_{2047}	11111...11111	log(1.11111...1111100000...00000)

To use a degree-2 polynomial ($y \approx ax^2+bx+c$) to interpolate $\log(1.m_{22-0})$, x refers to the value of $1.m_{22-0}$ which is the input of $\log(1.m_{22-0})$, y refers to the interpolated value of $\log(1.m_{22-0})$, and a, b and c are the corresponding coefficients. In Table 2–5, each entry stores three coefficients (a, b, c). To evaluate the coefficients for each entry, assume we have an input x whose index is larger than entry E_0, but smaller than entry E_1 ($E_0 < index_x < E_1$) in Table 2–4. In this case, we construct a parabola, and coefficients of the parabola have been stored into entry E_0 as a_0, b_0, c_0 in Table 2–5. Here, each coefficient uses four bytes, so each entry of Table 2–5 uses 12 bytes. To interpolate $\log(1.m_{22-0})$, one needs to use the highest 11 bits of the mantissa

$(1.m_{22-12})$ as an index to retrieve the LUT values for three coefficients (a, b, c), and then use the remaining of the mantissa $(1.m_{11-0})$ to perform three multiplications and two additions ($y \approx ax^2 + bx + c$) to obtain the interpolated value for $\log(1.m_{22-0})$. For interested readers, please refer to the section 2.4 "Interpolation Algorithm" and subsection 2.4.3 "Parabolic Interpolation" for more details about parabolic interpolation in my master thesis [48].

Table 2–5: 24 KB LUT for degree-2 polynomial interpolation [49]

Entry	Index (11-bits)	Content (96-bits)
E_0	$\underbrace{00000...00000}_{11}$	a_0 (32-bits), b_0 (32-bits), c_0 (32-bits)
E_1	00000...00001	a_1, b_1, c_1
E_2	00000...00010	a_2, b_2, c_2
...	...	...
E_{2045}	11111...11101	a_{2045}, b_{2045}, c_{2045}
E_{2046}	11111...11110	a_{2046}, b_{2046}, c_{2046}
E_{2047}	11111...11111	a_{2047}, b_{2047}, c_{2047}

2.6 Important Properties

To design high-performance implementation of elementary functions, there are numerous properties that we need to pay attention to. In this section, three important properties are presented:

1. Speed (latency, throughput)
2. Accuracy/Error (ULP)
3. Area (LUT size, degree of polynomial)

Speed: The speed of evaluating elementary functions (in software/hardware math libraries) is usually measured using latency (ns) or throughput (MHz). For elementary functions implemented using software, the latency or the number of CPU cycles is used to represent the time spent on a single function evaluation. As many elementary functions are implemented into hardware accelerators with pipeline architecture, the throughput of pipeline is applied to measure how many function evaluations are completed in one second.

Accuracy/Error: The IEEE-754 floating-point standard [52] defines single-/double-precision representations, which enables highly accurate elementary functions implementation. Using the standard, accuracy or error (interchangeable) is measured by the ULP (unit in the last place) [64, 93]. Here, 1 ULP error represents the gap between the two adjacent floating-point numbers. The Intel/Altera math library [12] generates function evaluations with 1 ULP error. In general, the accuracy is closely related to the algorithm of function evaluation, say, a higher degree polynomial and/or a larger LUT usually leads to more accurate results.

Area: The area of an elementary function accelerator in hardware refers to either the silicon area cost in an ASIC, or the amount of resources consumed in an FPGA. An look-up table is synthesized into RAM blocks, while the computation of a polynomial is synthesized into adaptive logic modules (ALMs) and DSP slices on an FPGA. Each DSP slice consists of multipliers and adders of various bit-widths. For our LUT-based accelerators, to reduce circuit area, we must try to shrink the table size as much as possible, as it is desirable to use as little memory as possible, both in an ASIC or FPGA implementation.

2.7 Discussion

In this section, we explain the impact of applying deep pipelining technique. As well, we present a discussion on the relationships, and various trade-offs made between the three properties: speed, accuracy and area.

Impact of deep pipelining: As many elementary function accelerators are implemented using pipeline architecture, one would apply a deep pipeline to further increase throughput. To do so, each pipeline stage has been shrunk aggressively to have extremely small delay, which is equal to the delay a few logic gates. This makes the number of pipeline stages increase, and the inclusion of flip-flops for the additional pipeline stages may increase circuit area. As the throughput becomes higher, the deep pipeline may exhibit larger circuit area and longer latency.

Performance trade-offs: Various trade-offs can be made to the elementary function accelerators in accordance with the requirements of different applications. For applications that are more tolerant of inaccurate evaluations (e.g. deep learning [66, 42]), one can trade accuracy for fewer area and/or faster speed. For our LUT-based reciprocal accelerators (see Chapter 3), accuracy drops from 1 to 3 ULP as the LUT is reduced to half the size (see Table 3–7). Likewise, after using degree-1 polynomial for interpolation, accuracy drops from 1 to 127 ULP. The changes in algorithms help to reduce the area, at the expense of higher error.

Roughly speaking, none of the three properties (speed, accuracy, area) is a “free lunch” in terms of the overall performance. To produce accurate evaluations, one may use a large LUT and/or high-order polynomial, but usually result in an area increase. Similarly, a faster implementation may also lead to an area increase, as more computing resources (i.e. DSPs) have been used to exploit the underlying parallelism in the algorithms. As we know, FPGAs have capacity restrictions for their resources (i.e. ALMs, RAMs, DSPs). It may be desirable, in certain applications, to allow some errors in function evaluation in order to reduce resource usage. For this purpose, we propose reduced-precision implementations for our iterative and LUT-based reciprocal accelerators presented in Chapter 3.

2.8 The Rise of the FPGA

The Field-Programmable Gate Array (FPGA) has entered the mainstream of computing mainly due to the following features [96, 75]: 1) reconfiguration capability, 2) high-performance (e.g. throughput/watt) and 3) energy efficiency. In principle, a circuit implemented on an ASIC [59] will perform at a higher speed and energy efficiency than an FPGA. However, the functions of an ASIC cannot be modified after its manufacture. Whereas, it is possible to change the function of an FPGA by reprogramming it. This reconfigurable feature of the FPGA offers a flexibility similar to a CPU. While a CPU typically operates at a higher frequency than an FPGA design, the spatial parallelism offered by an FPGA allows the performance gap to be recovered in many cases, e.g. [110, 83, 56, 60].

Modern FPGAs consist of logic elements (containing look-up tables (LUTs)), random access memory (RAM) blocks, digital signal processing (DSP) blocks and so on. DSP blocks are mostly used for arithmetic operations. The Intel/Altera Cyclone V FPGA, which we use in this work, contains columns of DSP blocks in the 2D FPGA fabric, with each DSP block containing multipliers, cascaded adders, accumulators, etc, as shown in Table 2–6. Table 2–7 shows the multipliers that a variable-precision DSP block supports. In this case, the DSP block can be configured as one 27×27, two 18×18 or three 9×9 independent multipliers for computation of various precisions. Users are able to create custom hardware by configuring the various internal blocks and interconnect of an FPGA. Any functional change to the hardware requires re-programming the FPGA.

When the computing resources of an FPGA have been properly configured/programmed, multiple instances of computational units can be realized

Table 2–6: A Variable-Precision DSP Block for Cyclone V Devices [10]

Support 9×9, 18×18 and 27×27 bits precision multiplication
Two 64-bit accumulators
A hard preadder of 18- and 27-bit modes
Cascaded output adders

Table 2–7: Variable-Precision DSP Configurations for Cyclone V Devices [10]

Applications	Multiplier Size (Bit)
Low precision fixed point computings	Three 9 x 9
Medium precision fixed point computings	Two 18 x 18
General DSP usage	Two 18 x 18 with accumulate
High precision fixed- or floating-point computings	One 27 x 27 with accumulate

to exploit the parallelism within applications. Conversely, on a CPU, computations are typically compiled into a long sequence of instructions on a CPU, executed in a relatively sequential order. Each instruction passes through a data path which generally includes instruction fetch, decode, register file access, execution and write-back. The generic nature of a CPU makes it easy to program. With an FPGA, on the other hand, since parallel hardware tailored to the application can be applied, superior energy performance can be achieved relative to a CPU.

With the significant growth of computing capacity in FPGAs, many applications are built with FPGA-based custom accelerators. Cloud service providers, such as Amazon and Microsoft, now deploy FPGA instances in their cloud or data centre architectures [16, 20]. In the context of cloud computing, heterogeneous architectures which contain FPGAs and CPUs have been adopted to accelerate applications. Custom hardware accelerators, together with CPU systems, are capable of delivering both performance and energy benefits to a broad range of computation-intensive applications. According to Amazon's website [16], an FPGA-assisted processor system in the AWS cloud has the potential of achieving up to *30*× speedup for some bioinformatics, financial risk and data analysis applications.

2.9 Development of Hardware Design Methodology

As summarized in [11], methods to design and verify hardware have evolved across recent decades. We take the emergence of electronic design automation as a milestone, dividing hardware development into pre-EDA and post-EDA periods. In the 1970s or earlier pre-EDA days, most hardware was hand designed and verified with the human eye [11]. At that time, the function of integrated circuits (ICs) was relatively simple and human-manageable [11]. As the complexity of the ICs increased in the 1970s, the EDA techniques arose, providing automated tools to design, debug and simulate hardware [11]. In the following subsections, we first present various design abstraction levels offered by EDA in subsection 2.9.1. Then, we proceed to a brief introduction on RTL and HDLs in subsection 2.9.2. Finally, we elaborate on the HLS in subsection 2.9.3, which includes its features, upcoming challenges, and general working flow.

2.9.1 Design abstraction levels

EDA offers different abstraction levels for digital design, as discussed in [112]. As shown in Figure 2–2, the higher the level of abstraction, the fewer design details an engineer deals with in hardware. Transistor-level design is low-level design, where the designer specifies the functionality using transistors [112]. In logic-level design, gates are the main building blocks, with which combinational and sequential circuits are built [112]. For more complex digital designs, such as processor designs, the primary building blocks are multiplexers, counters, adders, registers, and the datapaths connecting them together [112]. Broadly speaking, when a design is required to specify the cycle-by-cycle functionality of the circuit, the design methodology is referred

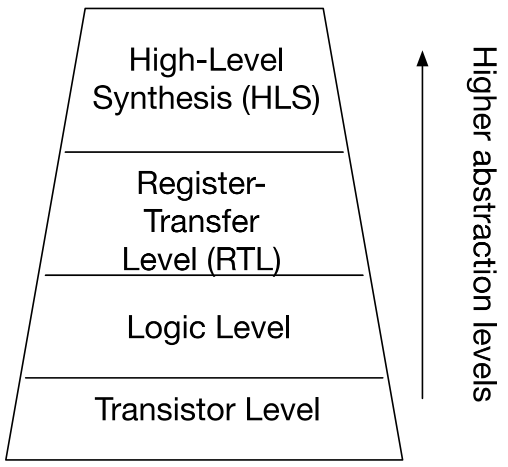
High-Level Synthesis (HLS)
Register-Transfer Level (RTL)
Logic Level
Transistor Level
Higher abstraction levels

Figure 2–2: Digital Design Abstraction Levels [6]

to as register-transfer level (RTL). High-level Synthesis (HLS) refers to the use of high-level programming languages (e.g. C/C++) to specify hardware circuit behaviour. Compared with RTL design, the HLS design methodology does not require one to specify the cycle-by-cycle circuit behaviour.

2.9.2 Register-Transfer Level (RTL) and Hardware Description Languages (HDLs)

In RTL methodology, engineers usually use (HDLs) to specify hardware circuit behaviour. HDLs have concurrent language constructs to resemble the parallel behaviour of computations in a hardware circuit. In [17], the process of implementing a circuit design on an FPGA includes: 1) writing the HDL code for the design, 2) compiling, simulating and synthesizing the circuit on a target FPGA, and 3) downloading the programming bitstream to the FPGA. The reconfigurable nature of FPGAs offers a flexibility which is comparable to that of the CPU [59]. The advent of EDA and the FPGA broadens the concept of what is conventionally considered as ”software” [111]. Using HDLs (e.g. VHDL, Verilog) to program an FPGA dramatically improves efficiency; however, there remain challenges with this approach:

- **Hard to debug.** Bugs in HDL code are notoriously difficult to track down and correct, owning to the low-level of abstraction, even for experienced hardware engineers.
- **Hard to modify.** HDL programming requires careful consideration of circuit details, such as timing constraints, control flow and datapaths. Thus, modifying/updating HDL code is complex and error prone.
- **Hard to learn.** There are many more software developers than hardware engineers. As such, hardware expertise is comparatively rare versus software expertise.

2.9.3 High-Level Synthesis (HLS)

In order to overcome the aforementioned challenges, HLS design methodologies have gained prominence to reduce the cost and time of hardware design. HLS lifts the level of abstraction from HDLs to software programming languages [45]. An HLS tool takes a C/C++ specification as input and then translates it into HDL (e.g. Verilog) code. Generally, HLS techniques significantly improve the productivity of FPGA programming, and reduce design-to-market time for FPGA products [45].

The development of HLS tools can be viewed in four generations [84], after two generations of failures (starting in the 1980s), the third-generation HLS tools began to see success in the early 2000s. The main reasons that led to the success of this generation are:

- **Shift to the right input languages.** Programming languages (i.e. C/C++) familiar to software developers were adopted to design hardware circuits [84].
- **Quality of the generated HDL was improved.** Advanced HLS tools allow for the generation of improved-quality HDL from software programming languages. Results from previous HLS tools had been of inferior quality (e.g. large circuit area, poor speed) [84].

In recent years, HLS has entered the mainstream of FPGA programming. Much effort has been made by FPGA vendors (e.g. Xilinx VivadoHLS [22]) and university startups (e.g. LegUp Computing [45]) to develop their HLS tools for commercial or academic use [94].

As discussed in [45], HLS remains an active research area for both industry and academia. A key challenge with current HLS tools is on generating more highly optimized hardware from a software specification rather than just functionally correct hardware [45]. Here, "more highly optimized hardware" refers to a low-area cost and/or high-speed/frequency circuits [45]. To produce more highly optimized hardware, a certain programming style is used for the C/C++ input. For example, one needs to manually remove the control flow and replace it with predication or explicitly indicate to the compiler the specific part of the hardware that needs to be pipelined. In doing so, it is possible to obtain orders-of-magnitude performance improvements in certain cases [85]. We elaborate on HLS-specific programming style in Section 4.11.

Recent HLS tools are built based on the LLVM open-source compiler framework [80]. From [94], we summarize the process of generating an HDL program from an HLS C/C++ input below:

- Parsing: The C/C++ program is parsed and translated into an intermediate representation (IR) for the compiler.
- IR optimization: The compiler further optimizes the IR before sending it to the HLS back-end, performing optimizations such as dead code elimination and loop transformations [70].
- Allocation: The compiler receives and recognizes constraints, such as the allowed width or number of multipliers available on the FPGA chip.
- Scheduling: The compiler assigns the computations in the IR into time-steps, which correspond to states of a finite-state machine (FSM). After this step, the IR is ”timed”.
- Binding: The compiler maps the timed IR onto hardware units.
- HDL generation: C/C++-specified HDL is generated.

2.10 Heterogeneous Computing

A heterogeneous computing system incorporates computational cores of various types (e.g. CPU, GPU, FPGA). In such a system, performance gains mainly come from assigning computations to the cores with the most suitable architecture for acceleration [76]. For example, CPUs are good at dealing with sequential applications (e.g. the operating system), while GPUs and FPGAs are more efficient at accelerating parallel applications with a large dataset (e.g. image processing). It is inefficient to have CPUs execute parallel tasks, as single instruction multiple data (SIMD)-styled instructions may translate into dozens of assembly instructions, which take hundreds of CPU cycles [122]. Likewise, if sequential tasks are assigned to GPUs or FPGAs, then the computing power may not only be wasted, but also cause additional energy consumption [122]. As we know, most machines on the market (i.e. laptops, tablets, smartphones) have a heterogeneous computing architecture. For example, the iPhone XR model is embedded with six CPUs (two performance cores and four efficiency cores), four GPUs, eight cores for the neural engine to accelerate machine learning applications and one image signal processor [23]. In addition, Amazon [16], Microsoft [20] and IBM [1] adopt heterogeneous architectures in their cloud computing platforms.

The FPGA is becoming prominent in heterogeneous computing thanks to its design flexibility and high performance. FPGA-based accelerators can be treated as a co-processor attached to a host CPU. In this scenario, the host CPU serves as a control unit, executing largely sequential code. An application-specific accelerator, on the FPGA, would be invoked by the CPU.

Chapter 3
Reciprocal Accelerators

3.1 Publication

Main content of this chapter comes from the following published manuscript:

[50] **Jing Chen**, Xue Liu and Jason H. Anderson, "Software-Specified FPGA Accelerators for Elementary Functions", the 2018 International Conference on Field-Programmable Technology (FPT'18), **full paper**.

3.2 Organization

In this chapter, we propose the algorithm designs and implementations of single-precision floating-point reciprocal hardware accelerators. Section 3.3 presents a brief introduction about this chapter. Next, we introduce prior work on the reciprocal accelerator in Section 3.4. In Section 3.5, we elaborate on the range reduction, which leverages a math formulation derivation to simplify reciprocal evaluation. In Sections 3.6 and 3.7, two algorithms are addressed, which are trial subtraction, and LUT along with degree-2 polynomial interpolation, respectively. In Section 3.8, accuracy is examined through exhaustive testing. An error distribution report is given as well. We compare our LUT-based accelerator with that of the Intel/Altera FPGA vendor in terms of speed and area usage in Section 3.9. In Section 3.10, we give a performance comparison between ours and the state-of-the-art reciprocal implementations. To obtain better performance (i.e. area, speed), we also generate implementations with reduced precision in Section 3.11. Finally, we demonstrate the HLS C programming style in Section 3.12, and summarize our work in Section 3.13.

3.3 Introduction

Reciprocals are frequently used in image and digital signal processing applications [91]. We implement single-precision, floating-point reciprocal accelerators. The algorithms of the proposed accelerators are designed using C language, and synthesized into hardware targeting on Intel/Altera 45 nm FPGA by the LegUp HLS [5], Quartus and ModelSim. For IEEE-754 single-precision non-subnormal inputs, the proposed accelerators are able to produce results with 1 ULP maximum, and 0.3-0.5 ULP average errors, respectively. The error distribution is obtained by conducting exhaustive testing on the domain of reciprocal, and generated results are then compared with reciprocal of the GNU math.h library.

The reciprocal accelerators are designed with two algorithms, which are trial subtraction (iterative) and LUT-based (non-iterative). For LUT-based algorithm, we use a LUT of around 1 KB and degree-2 polynomial interpolation. The novelties of the LUT-based accelerator are: 1) a small LUT, 2) a low degree of polynomial, and 3) highly accurate results. Since the LUT-based algorithm is universal, it could be applied to implement an entire library of single-precision elementary functions into high-performance hardware accelerators.

In evaluation, we compare our accelerators with the state-of-the-art implementations in terms of algorithm design, accuracy, testing method, throughput, architecture and platform. In particular, the Intel/Altera reciprocal IP core is treated as a "golden" baseline. In comparison with the baseline, our LUT-based reciprocal accelerators are considerably better in area usage, but slightly worse in maximum throughput on Cyclone V 45 nm FPGA. Because the proposed accelerators are specified by C language, it is easy to generate reduced-precision accelerators. In this case, accuracy is traded for faster speed and/or less area usage.

This work addresses two questions: 1) Can we find better algorithm for elementary functions design, which produces more accurate results but use a smaller LUT along with a lower order of polynomial?; 2) Can HLS efficiently interpret C specification to Verilog code for elementary functions implementation?

3.4 Related Work

We created a table to summarize the performance of previous reciprocal implementations. In Table 3–1, there are two columns, which are "($\times$, +)" and "Width". In this case, "($\times$, +)" represents the number of multiplications and additions that each reciprocal implementation contains, respectively. In addition, "Width" refers to the number of bits that each multiplication and addition has, respectively. For example, the first row of Table 3–1 tells readers that Cockburn et al. implements the reciprocal function using two multiplications and additions, respectively. Both the multiplication and addition are evaluated using a bitwidth of 30.

The methods used to implement reciprocal can be categorized into non-iterative and iterative approaches. Non-iterative approaches usually leverage polynomial and/or LUT interpolation. Iterative approaches, such as digit-recurrence, Newton-Raphson, Goldschmidt algorithms, use several iterations of multiplications and additions to generate results.

3.4.1 Non-Iterative Algorithms

Cockburn et al. [37] present a unified architecture for six key elementary functions in single-precision floating-point representation, which are $\frac{1}{x}$, $\sqrt{x}$, $\frac{1}{\sqrt{x}}$, $\log x$, $\ln x$, 2^x. The inputs of the six functions are limited to the interval [1, 2).

Table 3–1: Comparison of the proposed and previous reciprocal implementations

Authors	Year	LUT	Accuracy[1]		$(\times, +)$	Width	Exhaustive Testing[2]	Throughput (MHz)	Pipeline?	Methodology
			ULP	Error bound						
Cockburn et al.[3]	2010	832 B	1	n/a	(2, 2)	(30, 30)	partial	328	yes	FPGA
Cockburn et al.[4]	2010	7680 B	1	n/a	(2, 0)	(26, 0)	partial	301	yes	FPGA
Vestias et al.	2011	4352 B	n/a	$2^{-55} \times 1.100...1$	(5, 3)	(18, 18)	?[5]	294.1	yes	FPGA
Jose et al.	2014	4608 B	1	n/a	(2, 2)	(24, 48)	?	435	yes	FPGA
Cao et al.	2015	280 B	n/a	$2^{-25} \times 1.110...0$	(3, 3)	(28, ?)	?	?	no	ASIC
Libessart et al.	2017	n/a	?	?	(6, 3)	(34, 34)	?	294.1	yes	FPGA
Moroz et al.	2018	n/a	1	n/a	(5, 3)	(32, 32)	?	?	no	software
Intel IP	2018	?	1	n/a	?	?	yes	254-310	yes	FPGA
Our work[6]	2018	≈1024 B	1	n/a	(2, 2)	(32, 64)	yes	197-253	yes	FPGA

[1] Accuracy is given in either ULP or an upper bound (in single-precision floating-point format) for each work presented in this table.
[2] Accuracy is obtained through exhaustive or partial testing. There are $2^{32} \approx 4$ billion testing cases for single-precision $\frac{1}{x}$ in exhaustive testing. Partial testing uses random numbers to verify accuracy. The exhaustive testing approach allows us to make a stronger statement on the accuracy for $\frac{1}{x}$ using low-order polynomial.
[3] Reciprocal implemented using non-iterative approach.
[4] Reciprocal implemented using iterative approach.
[5] "?" represents unknown.
[6] Our LUT-based reciprocal accelerator.

The reciprocal function is implemented by using both non-iterative and iterative algorithms. For non-iterative implementation, it leverages a degree-2 polynomial and LUT interpolation. The LUT used for storing coefficients is equal to $2^7 \times 52$ bits ≈ 832 bytes. Using Horner's rule, it requires a total of two multiplications and two additions for the polynomial interpolation. With exhaustive testing on the input interval [1, 2), it offers 1 ULP maximum error. However, the area and maximum frequency of the proposed reciprocal accelerator is not explicitly shown in their paper. For iterative implementation, it takes advantage of the Newton-Raphson method with one iteration only. By doing so, a LUT of $2^{12} \times 15 = 61{,}440$ bits is provided for the initial approximation. Each iteration requires two multiplications and one bit inversion. The iterative reciprocal implementation is synthesized to the Xilinx Virtex-4 FPGA; it consumes 155 configurable SLICEs, four block memories and four DSP48 slices. In addition, one single reciprocal computation requires 10 clock cycles (33 ns latency) operating at 301 MHz. The maximum error is assumed to be correctly rounded to 1 ULP.

Jose et al. [71] present a single-precision floating-point arithmetic unit which supports reciprocal evaluation. The proposed unit leverages a degree-2 polynomial and LUT interpolation. A LUT of approximately 36 Kbits is used for the coefficients of the interpolation. The design is fully-pipelined and processes one reciprocal evaluation per clock cycle. It operates at 435 MHz on the Xilinx Virtex-7 FPGA. One single reciprocal evaluation takes nine clock cycles (20.69 ns latency). In addition, the unit consumes 302 LUTs, 327 flip-flops and two DSP slices on the target FPGA. The global error of the reciprocal is smaller than 2^{-28}.

Cao et al. [46] propose a hardware implementation of a single-precision (32-bit) floating-point reciprocal operator. It uses a degree-3 polynomial and a LUT interpolation algorithm. Compared to the degree-2 polynomial interpolation [95], it significantly decreases the LUT size. According to their paper, the maximum error and LUT size are 5.45×10^{-8} and 2,240 bits, respectively. The maximum latency of the proposed operator is measured in terms of the delay of the full adder, t. The reciprocal operator has a maximum latency of 12 t.

FPGA vendors provide highly optimized floating-point accelerators for basic arithmetic operations and elementary functions. Hence, users can instantiate the required floating-point cores in their design by treating them as black boxes. The Intel/Altera reciprocal intellectual property (IP) operates at 254-310 MHz and has a latency of 47.2-51.6 ns on the Intel/Altera Cyclone V FPGA. As a pipelined design, it accepts a new single-precision floating-point input at every clock cycle. The maximum error of the IP is 1 ULP for the inputs in the domain. Since the Intel/Altera IP is proprietary, we infer that it uses a LUT approximation from the large amount of memory usage in their FPGA synthesis report.

3.4.2 Iterative Algorithms

Vestias et al. [113] implement a decimal divider based on the Newton-Raphson iterative approach. The initial approximation is evaluated via a minimax polynomial. Each iteration has two multiplications and one addition. Two iterations are required to obtain a maximum error of 0.42×10^{-17} for a 16-bit division. The 16-bit divider is synthesized to a Virtex 4 SX35-12 FPGA: it consumes 1,478 SLICEs, 2,091 LUTs, 1,820 flip-flops and seven DSPs. In addition, a 34 Kb LUT is created to store the coefficients of the minimax polynomial. The maximum frequency of the divider is 294.1 MHz, it requires 112 clock cycles to complete (380.82 ns latency).

Libessart et al. [82] propose a fixed-point reciprocal operator. The Newton-Raphson FPGA implementation needs 3-5 iterations to compute the division. Each iteration consists of two multiplications and one subtraction. The pipelined architecture of the 16-bit fixed-point reciprocal is synthesized to the Xilinx Virtex-4 SX35 FPGA. Experiments show that it requires 25 clock cycles for one single reciprocal operation operating at 294.1 MHz frequency. The design consumes 372 LUTs, 568 flip-flops and seven DSPs.

Moroz et al. [91] present a single-precision floating-point reciprocal computation unit in software. The proposed unit is implemented using two modified Newton-Raphson iterations with a magic constant as the initial approximation. As a result, the precision of the unit is 23.8 bits (roughly 1 ULP). The reciprocal unit is tested on the ESP-WROOM-32 system, which contains two low-power Xtensa 32-bit microprocessors. The two-iteration algorithm takes 255.902 ns to finish on the ESP-WROOM-32 system.

He et al. [69] leverage Goldschmidt's algorithm to design a double-precision (64-bit) floating-point divider. A reciprocal LUT is created to calculate the initial approximation for the subsequent iterations. Similar to the Newton-Raphson algorithm, Goldschmidt requires two multiplications and one addition at each iteration. However, the multiplications can be computed independently in parallel. Experiments show that the 64-bit divider takes 12 clock cycles and operates at a frequency of 0.5 GHz.

3.4.3 Clarification Regarding Accuracy

There are $2^{32} \approx 4$ billion numbers defined by the single-precision (32-bit) floating-point representation. Single-precision elementary functions are those whose inputs and outputs are both in single-precision representation. To approximate a single-precision elementary function only via a LUT, a primitive table which stores all 32-bit approximations and is deemed to have full precision (zero error) is required. However, the full precision table would have 4 billion entries, which is neither practical to be implemented in software nor hardware. To address this problem, polynomial and/or LUT interpolations have been adopted to significantly reduce LUT size and restore accuracy. In order to evaluate the accuracy of elementary functions, two metrics are extensively used: 1) ULP and 2) an upper bound. ULP is discussed in Section 2.4, and is considered to be a straightforward way to measure accuracy. Hence, we only elaborate on the accuracy measured by the upper-bound metric and contrast it with the ULP metric.

Assume that the reciprocal function f(x) = $\frac{1}{x}$ has a single-precision input $x_1 = 2^{110} \times 1.0...0$, where there are 23 zeros in its mantissa. Hence, the result is f(x_1) = $2^{-110} \times 1.0...0$. Consider a second single-precision input x_2, and assume that f(x_2) = $2^{-110} \times 1.1...1$, where there are 23 ones in its mantissa. We express the error of f(x_2) with respect to f(x_1) as $|f(x_1) - f(x_2)| = 2^{-111}$ $\times$ 1.1...10, where only the LSB is 0. The error is approximately 7.703×10^{-34} in decimal, which is a very small value. However, if measuring the error under ULP, then the error is equal to $2^{23} - 1$ ULP. As a result, a small error bound in decimal may imply a very large error under ULP model.

3.5 Range Reduction

In the trial subtraction method, evaluation of a floating-point reciprocal is similar to that of floating-point division. In this case, division can be transformed to reciprocal by setting the dividend/numerator to floating-point value 1.0. In the IEEE-754 standard, 1.0 in single-precision format is represented as

$(-1)^0 \times 2^0 \times 1.0_{22-0}$. Thus, the reciprocal of *fp* can be represented as:

$$\frac{1}{fp} = \frac{(-1)^0 \times 2^0 \times 1.0_{22-0}}{(-1)^s \times 2^{e_{7-0}-127} \times 1.m_{22-0}} \tag{3.1}$$

$$= (-1)^s \times 2^{127-e_{7-0}} \times \frac{1.0_{22-0}}{1.m_{22-0}} \tag{3.2}$$

As shown in Eqn. 3.1, the sign of reciprocal $\frac{1}{fp}$ is the same as that of *fp*. The exponent of reciprocal $\frac{1}{fp}$ is the negation to that of *fp*. The mantissa of reciprocal $\frac{1}{fp}$ is equal to performing 23-bits fixed-point division between mantissas of value 1.0 and *fp*. Hence, we only need to calculate term $\frac{1.0_{22-0}}{1.m_{22-0}}$ rather than term $\frac{(-1)^0 \times 2^0 \times 1.0_{22-0}}{(-1)^s \times 2^{e_{7-0}-127} \times 1.m_{22-0}}$. The transformation of Eqn. 3.1 to 3.2 shrinks the domain of computing reciprocal $\frac{1}{fp}$. To comply with floating-point accelerators from FPGA vendors (e.g. Xilinx, Intel/Altera) and the FloPoCo [54], both reciprocals do not support subnormal inputs; instead, subnormal inputs are flushed to zero.

3.6 Iterative Implementation: Trial Subtraction

Trial subtraction is a digit-recurrence algorithm which leads to long evaluation time. For example, 25 iterations/cycles are required to compute the term $\frac{1.0_{22-0}}{1.m_{22-0}}$ in Eqn. 3.2 to obtain 1 ULP maximum error. A case study of trial subtraction is given below.

Case Study

Assume we would like to evaluate the reciprocal of 2.5 ($\frac{1}{2.5}$) in single-precision representation. Here, the binary floating-point value 2.5 is represented by Eqn. 3.3:

$$(2.5) = 0, 10000000, 01000000000000000000000 \tag{3.3}$$

$$sign = 0 \tag{3.4}$$

$$exponent = 10000000 = (128 - 127)_{decimal} = (1)_{decimal} \tag{3.5}$$

$$mantissa = 1.01000000000000000000000 \tag{3.6}$$

In the above, we use commas to separate sign, exponent and mantissa in Eqn. 3.3. Here, the exponent represents decimal value 1 after subtracting bias 127. Also, there is a "leading-1" for the mantissa. The steps used to compute the reciprocal in trial subtraction are as follows:

1) **Compute sign and exponent.** The sign of $\frac{1}{2.5}$ is the same as that of input 2.5, while the exponent of $\frac{1}{2.5}$ is the negation to that of input 2.5. Eqn. 3.7 shows the biased exponent of $\frac{1}{2.5}$.

$$\frac{1}{2.5}(exponent) = (-1)_{decimal} + (127)_{decimal} = (01111101)_{binary} \tag{3.7}$$

2) **Compute mantissa.** Apply the trial subtraction algorithm to compute the term $\frac{1.0_{22-0}}{1.m_{22-0}}$. To demonstrate the process step by step, X represents 1.0_{22-0} (mantissa of floating-point 1.0), and Y represents $1.m_{22-0}$ (mantissa of floating point 2.5).

$$X = 1, 00000000000000000000000 \tag{3.8}$$

$$Y = 1, 01000000000000000000000 \tag{3.9}$$

$$X\_HLS = 00000000100000000000000000000000 \tag{3.10}$$

$$Y\_HLS = 00000000101000000000000000000000 \tag{3.11}$$

The "leading-1s" in both X and Y represents the hidden bit in front of the radix point of the mantissa. In HLS C software, a 32-bit unsigned integer type is used to represent X and Y as shown by Eqns. 3.10 and 3.11 . In this case, 25 iterations/cycles are needed to obtain 1 ULP error for the final value of reciprocal. In the first iteration/cycle, we perform subtraction $X - Y$ as a 32-bit unsigned integer. As the result is negative, X is shifted one bit left, and the least significant bit (LSB) of quotient Q is set to “bit-0”.

$$(X << 1) : 0000000100000000000000000000000\mathbf{0} \tag{3.12}$$

$$Y : 00000000101000000000000000000000 \tag{3.13}$$

$$Q : 0000000000000000000000000000000\mathbf{0} \tag{3.14}$$

In the second iteration/cycle, we perform trial subtraction again on $(X << 1)$ and Y to obtain a positive result. We replace $X << 1$ with $(X << 1) - Y$, then shift the quotient Q one bit left, and finally set the LSB of Q to “bit-1”.

$$(X << 1) - Y : 00000000011000000000000000000000 \tag{3.15}$$

$$Y : 00000000101000000000000000000000 \tag{3.16}$$

$$Q : 000000000000000000000000000000\mathbf{01} \tag{3.17}$$

After 25 iterations/cycles of trial subtraction, we have the following result for quotient Q:

$$Q : 00000001100110011001100110011001 \tag{3.18}$$

In our case, the “leading-0” appears at the 24th bit of the quotient Q. This facilitates exponent adjustment in the next step.

3) **Exponent adjustment.** The reciprocal of $\frac{1}{2.5}$ is a normal number defined by the IEEE-754 standard, which implies that there is a “leading-1” at the 23rd bit of the mantissa. Since the biased exponent of $\frac{1}{2.5}$ (Eqn. 3.7) falls into the interval of [-125, +126], quotient Q is shifted one bit to scale as the mantissa, making the “leading-1” appear at the 23rd bit. After that, the biased exponent of $\frac{1}{2.5}$ is subtracted by 1 accordingly.

$$\frac{1}{2.5}(exponent) = (125)_{decimal} = (01111101)_{binary} \tag{3.19}$$

$$(Q >> 1) : 00000000110011001100110011001100\mathbf{1} \tag{3.20}$$

4) **Rounding.** After the quotient Q is scaled to the mantissa in Eqn. 3.20, the LSB of Q is removed after rounding. To restore accuracy, we tried three rounding schemes: 1) round to zero (truncation); 2) round to nearest, ties away from zero; and 3) round to nearest, ties to even. Both 2) and 3) are intended to give results with fewer errors than 1) in theory, but may produce larger circuit area when implemented into hardware. In our case, rounding schemes 2) and 3) produce the same error (1 ULP) as 1) through exhaustive testing. Therefore, rounding scheme 1) has been adopted due to the smaller hardware overhead. To obtain results of 0 ULP (no error) in the trial subtraction algorithm, one needs to have approximately 50 iterations/cycles to compute the quotient Q in Eqn. 3.18 along with rounding scheme 2).

$$Q\ \ after\ \ rounding : 00000000110011001100110011001100 \tag{3.21}$$

5) **Result Reconstruction.** We concatenate the mantissa with the biased exponent and sign to form the final result for $\frac{1}{2.5}$ shown by Eqn. 3.22. For inputs of reciprocals, when their biased exponent is either 253 or 254, some outputs become subnormals (extremely small values close to zero) according to the IEEE-754 standard.

$$\frac{1}{2.5}(result) : 0, 01111101, 10011001100110011001100 \tag{3.22}$$

6) **Exception handling.** For reciprocal inputs whose values are subnormal, (± zero), (± infinity) or NaN defined by the IEEE-754 standard, there is an exception handling demonstrated by Table 3–2. For example, if input is + zero (positive zero), then the corresponding output is + infinity (positive infinity). In the IEEE-754 standard, the 32-bit encoding for + infinity is 0x7f800000 in hexadecimal representation. To comply with the reciprocal implementation of the Intel/Altera math library, subnormal inputs are flushed

to zero. Hence, the reciprocal of a subnormal is +/- infinity. In addition, to comply with results from the GNU `C math.h` library, when inputs of the reciprocal function are NaNs, outputs are *fp* || 0x00400000.

Table 3–2: Exception Processing for Reciprocal [50]

Input	Output	Encoding
subnormal	+,- Infinity	0x7f800000/0xff800000
+,- zero	+,- Infinity	0x7f800000/0xff800000
+,- infinity	+,- Zero	0x00000000/0x80000000
NaN	NaN	*fp* || 0x00400000

3.7 Non-Iterative Implementation: Lookup-Table (LUT)

A LUT along with polynomial interpolation is an alternative way to implement elementary functions. Compared to the iterative method, the LUT-based approach offers different resource-usage trade-offs. It requires less evaluation time and a smaller circuit area in hardware. In the LUT-based approach, function values are pre-computed and stored in a LUT, so that function evaluation is equivalent to table retrieval. In this case, the LUT size is critical, since a large table leads to cache thrashing and extra area cost in software and hardware. To shrink the LUT size, we do not store all function values in the LUT. Missing values are restored through polynomial interpolation.

Recall that we need to compute the term $\frac{1.0_{22-0}}{1.m_{22-0}}$ for reciprocal in Eqn. 3.2, whose domain is [1:2) and range is $(\frac{1}{2}:\ 1]$. To compute $\frac{1.0_{22-0}}{1.m_{22-0}}$, a LUT is created and divided into 128 intervals as $log_2(128) = 7$. Within any interval, a degree-2 polynomial (y = a $\cdot x^2$ + b $\cdot x$ + c) is used to interpolate those missing values. Here, a, b and c are scalar coefficients which are stored in the LUT, and their values vary for intervals. Since $log_2(128) = 7$, the seven most significant bits (MSBs) of the mantissa (i.e. m_{22-16}) are used to address the LUT, while the remaining 16-bits (i.e. m_{15-0}) are treated as x for degree-2 polynomial interpolation. In order to have better performance and a lower area cost, we need to keep the LUT and degree of polynomial as small as possible.

To compute the coefficients for degree-2 polynomial interpolation, we use the Chebyshev polynomial approximation [109] through the `mpmath` Python library. The `chebyfit` function in the library computes coefficients for each interval given a polynomial degree of a function (i.e. reciprocal). To further reduce the number of multiplications for polynomial interpolation, we have $\mathrm{y} = \mathrm{a} \cdot x^2 + \mathrm{b} \cdot x + \mathrm{c}$ as $\mathrm{y} = x \cdot (\mathrm{a} \cdot x + \mathrm{b}) + \mathrm{c}$ by applying Horner's rule. Exception handling of the LUT-based approach is the same as that of the iterative method.

3.8 Error Study

In this section, we analyze the accuracy of the reciprocal accelerators through exhaustive testing. To better show the accuracy, error distributions are given to trial subtraction and LUT approaches, respectively. Apart from the maximum and average errors, we also calculate the percentage of evaluations that have 0 or 1 ULP error.

3.8.1 Exhaustive testing

We perform exhaustive testing on the domain of reciprocal to obtain an overall error distribution. As subnormal numbers and zero are excluded, there are approximately 4×10^9 input cases for reciprocal. Previous work [36] produces an error report by testing a certain number of random inputs (i.e. 2×10^7 input cases) on the domain of the logarithm function. However, it is still possible that inputs other than the tested cases may produce a larger error. Thus, an error report that is not generated by exhaustive testing may not be able to find the maximum error in the domain of a function. On the other hand, our error distribution guarantees that every valid input of reciprocal complies with the maximum and average errors we stated in Table 3–3. In addition, reciprocal evaluations generated by the GNU `math.h` library are treated as a golden benchmark.

3.8.2 Error Distribution

In Table 3–3, only *normal* inputs (exponent varies from 1 to 254) are shown. For other inputs, such as subnormal or zero, the results can be found in exception handling Table 3–2. The first half of Table 3–3 demonstrates the error distribution of reciprocal in the trial subtraction algorithm. The first column represents the biased exponent: we split *normal* inputs into 254 intervals by the biased exponent. The second and third columns represent the maximum and average errors for each interval, respectively. The last two columns represent number of 0 or 1 ULP cases at each interval. Percentages in these two columns denote the percentage of evaluations which have either 0 or 1 ULP error, for a total of 8,388,608 testing cases. We combine intervals 1 to 252 into one entry in Table 3–3 because they have the same accuracy. To simplify, the input sign is omitted, so each interval represents both positive and negative numbers. From this data, we conclude the maximum error of reciprocal is 1 ULP, while the average error is 0.5 ULP because the number of 0 or 1 ULP cases each accounts for approximately 50.0% of total testing cases.

Similarly, the second half of Table 3–3 demonstrates the error distribution of the LUT-based reciprocal. In this case, we exclude inputs with biased exponents of 253 and 254 because the evaluation of reciprocal falls into the subnormal numbers. In the LUT-based approach, although the maximum error is still 1 ULP, the average error has been improved to 0.31 ULP, and the percentage of evaluations which have 0 ULP is 68.7%. The reciprocal from Intel/Altera is guaranteed to have 1 ULP error at most.

Table 3–3: Error distribution for reciprocal [50]

Trial Subtraction Reciprocal				
Exp	Max Error (ULP)	Avg Error (ULP)	0 ULP (50.01%)	1 ULP (49.99%)
1 to 252	1	0.50	4,194,910 cases	4,193,698 cases
253	1	0.50	4,194,555 cases	4,194,053 cases
254	1	0.50	4,194,647 cases	4,193,961 cases
LUT-Based Reciprocal				
Exp	Max Error (ULP)	Avg Error (ULP)	0 ULP (68.7%)	1 ULP (31.3%)
1 to 252	1	0.31	5,767,024 cases	2,621,584 cases

3.9 Experimental Study

In this section, we demonstrate the performance of the LUT-based and trial subtraction reciprocal accelerators in terms of speed and area usage on Intel/Altera 45nm FPGA. To show the efficiency of our designs, we carefully compare the performance of our LUT-based accelerator with that of the Intel/Altera IP cores, targeting the same FPGA device (Cyclone V).

LegUp HLS [5] allows us to generate pipelined reciprocal accelerators, whose maximum throughput can be specified by user based on various application requirements. We experimented with five different clock period constraints: 20, 15, 10, 5 and 2 ns. The Verilog circuit interpreted by the HLS is influenced by such constraints: as the clock period decreases, the accelerators are supposed to have a finer-grained pipeline architecture. In Table 3–4, we present our accelerators in three configurations: 1) the accelerator with the fastest maximum throughput, 2) the accelerator with the smallest circuit area, and 3) the accelerator in the midway. To clarify, the midway accelerator usually has lower maximum throughput and larger circuit area than that of 1) and 2), respectively.

For the Intel/Altera reciprocal IP core generator, it is also possible to specify a target maximum throughput. Here, we make their accelerators target 150, 200 and 250 MHz. Although the algorithm used to implement the Intel/Altera IP core is not disclosed, it is highly likely that they use polynomial interpolation/approximation and a LUT-based approach because memory blocks are being used. We also note that cores from Intel/Altera are based on RTL design (i.e. hand-designed using HDL and/or a circuit diagram), while our accelerators are designed using a higher level of abstraction (i.e. `C` language). Both RTL and HLS accelerators accept one new input to the pipeline architecture per clock cycle.

3.9.1 LUT-Based Reciprocal Accelerator

In Table 3–4, we compare our accelerators with Intel/Altera IP cores. All accelerators are synthesized into hardware using Quartus version 16.1 and targeted to the Intel/Altera Cyclone V 45nm FPGA. For performance, we show maximum throughput and pipeline depth. For circuit area, we show adaptive logic modules (ALMs), memory bits (Mbits), DSPs and effective ALMs (eALM). ALMs in Cyclone V comprise a fracturable six-input LUT with two extra inputs. An ALM can implement any six-input function, any two four-input functions, and several other combinations. DSP units in Cyclone V can implement three narrow 8×8 multiplies, two 18×18 multiplies, or one wide 27×27 multiply [10]. To explain, ALMs represent the number of soft logic elements used by the circuit, Mbits represent the memory bits consumed by the LUT, and DSP slices offer adders and multipliers of various widths. Here, eALM represents the total number of soft logic elements used by the circuit. In this case, the area consumed by the DSPs and memories is translated into effective ALMs. Based on data presented in the literature [100, 116], each

Table 3–4: Intel vs. our LUT-based reciprocal accelerators [50]

Intel/Altera FP_FUNCTIONS						
	Throughput	**Latency**	**ALMs**	**Mbits**	**DSPs**	**eALMs**
Smallest	254.9	12	163	30,208	3	348.67
Midway	280.5	14	192	30,208	3	377.67
Fastest	310.1	22	273	30,208	6	548.67
Our Work (handling exceptions)						
	Throughput	**Latency**	**ALMs**	**Mbits**	**DSPs**	**eALMs**
Smallest	197.16	9	150	10,112	2	273.78
Midway	195.47	10	157	10,112	2	280.78
Fastest	253.68	15	203	10,112	2	326.78
Our Work (no exception handling)						
	Throughput	**Latency**	**ALMs**	**Mbits**	**DSPs**	**eALMs**
Smallest	184.71	8	136	10,112	2	259.78
Midway	183.89	9	139	10,112	2	262.78
Fastest	222.92	12	168	10,112	2	291.78

M10K memory block is equivalent to 31.89 ALMs and each DSP unit is equivalent to 30 ALMs. Both ours and Intel/Altera accelerators achieve 1 ULP maximum error. We perform exhaustive testing only on the C specification of our accelerator, while HLS-interpreted Verilog circuits are simulated with ModelSim to verify functional and timing correctness.

The top part of Table 3–4 shows the results for the Intel/Altera IP cores, and the two bottom parts show the results of our LUT-based accelerators. For the fastest accelerator, we observe that the Intel/Altera reciprocal accelerator operates at 310.1MHz, has 22 cycles of pipeline depth, and consumes 273 ALMs, 30.2K Mbits and six DSP slices, leading to the total circuit area of approximately 548.67 eALMs. Our fastest reciprocal accelerator with exception handling operates at 254 MHz, has fifteen cycles of pipeline depth and, consumes 203 ALMs, 10.1K Mbits and two DSP slices, so the total circuit area is approximately 326.78 eALMs. Similarly, for the smallest accelerator, we observe that Intel/Altera reciprocal accelerator operates at 254.9 MHz, has 12 cycles of pipeline depth and, consumes 163 ALMs, 30.2K Mbits and three DSP slices, so the total circuit area is approximately 348.67 eALMs. Our smallest reciprocal accelerator with exception handling operates at 197.16MHz, has nine cycles of pipeline depth and, consumes 150 ALMs, 10.1K Mbits and two DSP slices, so the total circuit area is approximately 273.78 eALMs. In both cases, Intel/Altera offers a slightly better maximum throughput, but our accelerator has a shorter pipeline depth and considerably less circuit area in ALMs, Mbits and DSPs. Such advantages may be beneficial in real-time applications. Our midway reciprocal accelerator has larger circuit area and lower maximum frequency than the smallest and fastest accelerators, respectively. Table 3–4 also presents our accelerators which exclude exception handling. We observe that the accelerators without exception handling offer a slightly lower maximum frequency, but have a shorter pipeline depth and a lower circuit area in ALMs than ones with exception handling.

Figure 3–1 illustrates maximum throughput and eALMs (circuit area) for Intel/Altera and our reciprocal accelerators (with and without exception handling). In Figure 3–1, maximum throughput is shown on the vertical axis, and eALMs are shown on the horizontal axis. Blue dots represent Intel/Altera implementations, while orange and grey dots represent our implementations with and without exception handling, respectively. We observe the same results as those of Table 3–4, i.e. that Intel/Altera implementations operate at slightly higher throughput, and our implementations have considerably less circuit area. The Cyclone V FPGA is considered to be low-end, so many applications that run on such a device do not need to operate at a frequency which exceeds 250 MHz.

We believe that the Intel/Altera reciprocal accelerators presented in Table 3–4 are developed by a large IP team. Furthermore, they are highly optimized for the target FPGA platform, so that they can fully take advantage

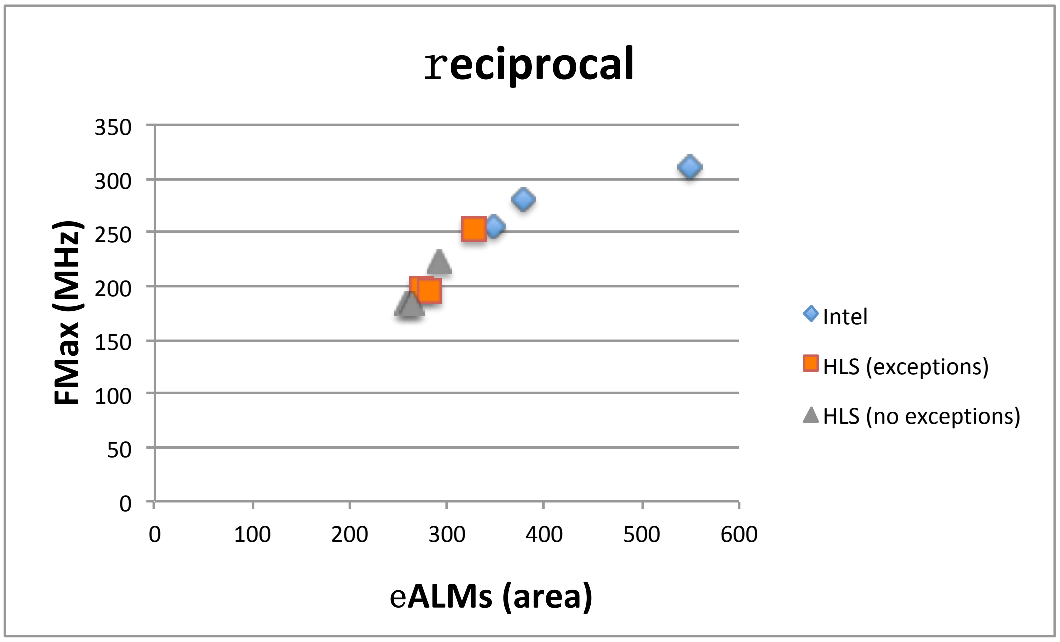

Figure 3–1: Throughput vs. area (eALMs) trade-offs for reciprocal accelerators. [50]

of the underlying resources and features of the FPGA architecture. The performance of our accelerators are competitive with that of the Intel/Altera IP cores. We observe that our accelerators are better in pipeline depth and circuit area, but are slightly worse in maximum frequency. The reasons that we are able to design the competitive reciprocal accelerator are: 1) better algorithm: we use a small LUT and low-degree polynomial to generate highly accurate (1 ULP) results for reciprocal, which leads to a significant reduction to circuit area (especially memory usage), and 2) HLS tool: LegUp HLS efficiently interprets the algorithm design of reciprocal (specified in C language) into Verilog code, and then map it to the target FPGA.

3.9.2 Iterative Reciprocal Accelerator

Table 3–5 demonstrates our reciprocal accelerators in the trial subtraction method. The left columns show results with exception handling, while the right columns show results without exception handling. As above, we present three configurations, which are accelerator with the smallest circuit

Table 3–5: Iterative implementations of reciprocal, with and without exception handling [50]

	With exceptions			**Without exceptions**		
	Throughput	**Latency**	**ALMs**	**Throughput**	**Latency**	**ALMs**
Smallest	126.58	15	842	121.24	14	819
Midway	213.77	28	1,027	214.68	27	976
Fastest	234.14	81	2,294	236.18	78	2,184

area, accelerator with fastest throughput, and accelerator in the midway. We observe that there are neither DSP slices nor memory bits consumed in the iterative implementations. Thus, iterative implementations are entirely built with ALMs (soft logic). For implementations with exception handling, we observe that throughput ranges from 126.58 to 234.14 MHz, pipeline depth ranges from 15 to 81 cycles, and ALMs range from 842 to 2,294. For implementations without exception handling, we observe that throughput basically remains the same, but there are reductions in pipeline depth and ALM consumption. In comparison with the LUT-based implementations shown in Table 3–4, iterative implementations have longer pipeline latency and more ALM usage (approximately six to 13 times larger). However, we believe that iterative implementations are feasible alternatives when the target application requires neither DSP nor memory (i.e. low-power embedded applications), or the target FPGA platform contains neither DSP slices nor memory blocks (e.g. FPGAs from Lattice Semiconductor [4]).

3.10 Performance Comparison

In this section, we compare our LUT-based with the state-of-the-art reciprocal implementations mentioned in section 3.4. The last one row in Table 3–1 refers to our LUT-based accelerator. We observe that our LUT is smaller than most implementations. Cockburn et al. propose a LUT-based reciprocal accelerator, which uses 832 B LUT and has a maximum error of 1 ULP. However, the error is obtained only through exhaustive testing on interval [1, 2). Cao et al. also leverage a smaller LUT, but they take advantage of degree-3 polynomial interpolation, which requires more usages of multiply and add. The throughput of their work is measured by delay of a full adder. Unfortunately, we are not sure about the accuracy of their work. In fact, the testing method for most implementations is not explicitly mentioned in the respective papers. Both Intel IP and our proposed accelerators achieve 1 ULP maximum error through exhaustive testing. The two error bounds which are given in decimal are translated into single-precision floating-point format. It is worth reminding the reader that a small error bound in decimal may imply a very large error under ULP (see Section 3.4.3).

It is difficult to give a fair and quantitative comparison among various implementations of reciprocal in terms of area usage and throughput, as they are targeted to different platforms. As a proxy for area complexity, we report the number and width of multiplications and additions performed. In this case, our proposed accelerator has lower multiplication and addition usage than the majority. It has two 32-bit multiplications, one 32-bit addition and one 64-bit addition. The throughput of our accelerator is not remarkable in comparison with other implementations, as Cyclone V is considered to be a low-end low-cost FPGA. However, the pipeline architecture adopted by our accelerator accepts a new input of reciprocal at every clock cycle. Hence, we say the initiation interval (II) of the pipeline is equal to 1. However, it may not be true that other pipeline accelerators in Table 3–1 also have II = 1.

Finally, it is worth reinforcing that the algorithms of our reciprocal accelerators are designed by `C` language, rather than in manually designed RTL hardware. Our accelerators, therefore, are straightforward to change and modify, and the C specification, combined with HLS, allows a range of area/performance trade-offs to be explored rapidly. It also validates LegUp HLS [5] can interpret C specification of elementary function into Verilog code in an efficient way.

3.11 Reduced-Precision Variants

A large number of applications, such as signal or image processing, do not require absolutely accurate evaluations. In the field of approximate computing, researchers trade the accuracy of designs for faster speed or area reduction. In [72, 121], accuracy-adjustable adders are proposed for applications with varying accuracy. Since the algorithms of our accelerators are designed by C language, it is easy for them to be modified for implementations with lower accuracy. This is useful for applications which are critical in resource usage, but more tolerant of inaccurate evaluations. In this scenario, Table 3–6 shows the results of reduced-precision implementations for iterative reciprocal accelerators. To simplify, we present only two reduced-precision variants, which offer 32 and 1024 ULP maximum error. Reduced-precision variants are created by reducing the number of iterations used in trial subtraction. The top half of Table 3–6 gives results in which fast speed is a priority. In this case, clock period constraints have been shrunk aggressively to better pipeline our designs. The bottom half gives results in which small circuit area is a priority. Compared to Table 3–5, we observe that throughput increases while pipeline latency and area decrease as precision is reduced in Table 3–6. Notice that the throughput-objective accelerators have longer pipeline latency and are built with more soft logic units than area-objective accelerators. Part of the reason is that the throughput-objective implementations have finer-grained pipeline architecture.

Table 3–7 shows the results for reduced-precision implementations of LUT-based reciprocal accelerators. The top half of the table shows results

Table 3–6: Reduced-precision variants of iterative reciprocal [50]

	Throughput	**Latency**	**ALMs**	**Accuracy**
Throughput objective	243.55	61	1688	32 ULP
	251.51	46	1197	1024 ULP
Area objective	126.73	11	637	32 ULP
	129.05	9	473	1024 ULP
Our work	213.77	28	1,027	1 ULP

Table 3–7: Reduced precision variants of LUT-based reciprocal [50].

	Throughput	**Latency**	**ALMs**	**Mbits**	**DSPs**	**Accuracy**
Degree-1 poly	193.61	5	78	6656	1	127 ULP
	194.51	6	82	6656	1	127 ULP
	196.58	7	91	6656	1	127 ULP
Half-size table	222.12	7	115	5056	2	3 ULP
	211.19	8	124	5056	2	3 ULP
	221.68	9	127	5056	2	3 ULP
Intel/Altera	310.1	22	273	30,208	6	1 ULP
Our work	253.68	15	203	10,112	2	1 ULP

of accelerators which use the same LUT, but degree-1 polynomial for interpolation. The bottom half gives results of accelerators which use the same degree of polynomial, but a LUT that is half the size. In addition, the top and bottom implementations have a maximum error of 127 ULP and 3 ULP, respectively. We observe that degree-1 polynomial implementations operate at roughly 200 MHz and, use 78-91 ALMs, 6,656 memory bits and one DSP. The half-size-LUT implementations operate at roughly 220 MHz and use 115-127 ALMs, 5,056 memory bits and use two DSPs. Compared with the 1 ULP precision results, LUT-based reduced-precision implementations achieve a slightly lower frequency, but use significantly fewer ALMs and almost half-size memory usage. Also, we note that degree-1 polynomial (y = a $\cdot$ x + b) implementations only use one DSP slice, as one multiplication and addition are required in this case. It is easy to modify the C specification to create reduced-precision implementations to accommodate speed- or resource- sensitive applications.

3.12 HLS C Implementation

In this section, we show the HLS C programming style by presenting a number of coding examples to demonstrate how it is slightly different from traditional C programming. The HLS C specifications of floating-point reciprocal accelerators are implemented by using fixed-point data types and arithmetic. For example, 32-bit unsigned integer types are used for inputs of the accelerators. We perform bit-mask operations to extract the sign, exponent and mantissa. Thus, 8-bit, 16-bit and 32-bit unsigned integer types have been used accordingly. All floating-point computations are then implemented as fixed-point arithmetic.

- **Top design.** Floating-point accelerators are instantiated as C functions. For example, the reciprocal accelerator in the trial subtraction method is instantiated as `trial_recip()`. The following code snippet shows the structure of the top design. Two queues are created: one is for reading inputs to `trial_recip()`, and the other is for writing outputs produced by `trial_recip()` to verify functional correctness. The `while` loop statement constantly checks the input queue, and writes results to the output queue.

```
// define two queues for input and output, respectively
void fpCore(FIFO* inputQ, FIFO* outputQ){
 ...
// constantly read an input from input queue
 while (1) {
   // read an single-precision (32-bits) input
   uint32_t input = fifo_read(inputQ);
   // feed the input to square root accelerator,
   // and then get an output as result
   uint32_t output = trial_recip(input);
   // write the output to output queue
```

```
      fifo_write(outputQ, output);
    }
  }
```

- **Bit-mask operation.** Given a single-precision floating-point number SP, the following code snippet shows how to extract the sign (1 bit), exponent (8-bits) and mantissa (23-bits) from SP. In this case, we include the `stdint.h` library, which defines new data types that explicitly specify bitwidth and the signed/unsigned attribute, such as `uint32_t`, `uint16_t` and `uint8_t`. By doing this, we can precisely control the bitwidth for each variable in bit-mask operations. Otherwise, the HLS tool sometimes assigns a data type with unexpected bitwidth to those variables.

```
// <stdint.h> library defines new data types, such as
// uint32_t, to replace unsigned int
#include <stdint.h>
// sp is a single-precision (32-bits) floating-point input
uint8_t sign = sp >> 31;
uint16_t exp = (sp & 0x7f800000) >> 23;
uint32_t mantissa = sp & 0x007fffff;
```

- **Replacing conditional statements and logical operators.** We manually replace all `if-then-else` statements with the alternative expression `<cond>?<val1>:<val2>`. In doing so, the HLS tool can generate more efficient Verilog code from the `C` specification without inserting unnecessary control flow. The HLS tool automatically removes redundant control flow, but we sometimes manually optimize complicated conditional statements for better performance. Likewise, all logical operators `(||, &&)` are replaced with bitwise operators `(|, &)`. If this is not done, the HLS tool occasionally mis-inserts control flow to places where the logical operators appear. Commented code in the following code snippet shows `if-then-else` statements and logical operators in a traditional C program, while the code below demonstrates an HLS-style C program after making the corresponding replacements. Note that in versions later than LegUp HLS version 5.5, manual replacements to `if-then-else` statements and logical operators are no longer required, as the HLS tool has been improved to perform such changes internally and automatically.

```
 /*
if (!c1 && !c2)
    xn = xn + 0x01800000;
else
    xn = xn;
*/
xn = (!c1 & !c2)? (xn + 0x01800000) : xn;
```

3.13 Summary

In this chapter, we presented single-precision floating-point reciprocal hardware accelerators. The accelerators are designed with two algorithms: 1) iterative (trial subtraction), and 2) LUT along with degree-2 polynomial interpolation. The algorithms are implemented by C language, and synthesized into hardware using the LegUp HLS, Quartus and Modelsim. Through exhaustive testing, we find that the maximum errors of both the iterative and LUT-based accelerators are 1 ULP, which is equivalent to Intel/Altera IP cores. In evaluation, we compare the performance of our LUT-based reciprocal accelerators with that of Intel/Altera IP cores, which are both targeted to the Cyclone V 45nm FPGA. Results show that Intel/Altera IP cores win slightly in throughput, but our accelerators win considerably in terms of pipeline latency, circuit area, especially memory usage. For iterative accelerators, although the performance is not as good as the LUT-based ones, they are still attractive to embedded applications targeted at low-end FPGAs since no DSP units are used. In addition, we also propose accelerators with reduced-precision options for applications with more emphasis on speed and area than precision. In this case, we trade accuracy for higher throughput and a fewer circuit area. In general, we consider the performance of our reciprocal accelerators to be promising, since Intel/Altera IP cores are developed by a large team and are undoubtedly highly optimized toward their own FPGAs.

Chapter 4
Square Root Accelerators

4.1 Publication

Main content of this chapter comes from the following published manuscript:

[50] **Jing Chen**, Xue Liu and Jason Anderson, "Software-Specified FPGA Accelerators for Elementary Functions", the 2018 International Conference on Field-Programmable Technology (FPT'18), **full paper**.

4.2 Organization

In this chapter, we propose the algorithm designs and implementations of single-precision (32-bit) floating-point square root hardware accelerators. Section 4.3 presents a brief introduction about this chapter. We review prior work on square root accelerators in Section 4.4. In Section 4.5, we elaborate on the range reduction, which leverages a math formulation derivation to simplify the square root evaluation. In Sections 4.6 and 4.7, two algorithms are addressed, which are Newton Raphson's method, and LUT along with degree-2 polynomial interpolation, respectively. In Section 4.8, accuracy is examined through exhaustive testing. An error distribution report is given as well. We compare our LUT-based accelerator with Intel/Altera square root IP core in terms of speed and area usage in Section 4.9. In Section 4.10, we give a performance comparison between ours and the state-of-the-art square root implementations. Finally, we demonstrate the HLS `C` programming style in section 4.11, and summarize our work in Section 4.12.

4.3 Introduction

Square roots are frequently used in digital signal processing applications and wireless communication system [47, 91]. The proposed square root accelerators are designed with two algorithms, which are Newton's method (iterative) and LUT-based approach (non-iterative). For LUT-based algorithm, we use a LUT of around 1 KB and degree-2 polynomial interpolation. Since the LUT-based algorithm is universal, it could be applied to implement an entire library of single-precision elementary functions into high-performance hardware accelerators. Based on exhaustive testing on the domain of square root, the accelerators are able to produce results with 1 ULP maximum and 0.25-0.27 average errors, respectively, compared with square root from GNU `math.h` library. The algorithms are implemented by C language, and synthesized into hardware targeting on Intel/Altera 45 nm FPGA by the LegUp HLS [5], Quartus and ModelSim. Similar to the reciprocal accelerators, we compare ours with the state-of-the-art implementations in terms of algorithm design, accuracy, testing method, throughput, architecture and platform.

4.4 Related Work

Table 4–1 gives readers a good understanding of previous square root implementations. In this case, the accuracy is given either in ULP or by an error bound in decimal. However, we translate error bounds from decimal to single-precision floating-point format. Recall that a small error bound in decimal may imply a very large error under ULP (see section 3.4.3). There are numerous algorithms for implementing floating-point square root. Generally, they can be categorized as non-iterative and iterative algorithms.

4.4.1 Non-Iterative Algorithms

This usually leverages polynomial and/or LUT approximation for square root evaluation. Kwon and Draper [78] present a pipelined, multiply/divide/square root fused floating-point computation unit by using Taylor series expansion and LUT approximation. In this case, a degree-3 Taylor series and 448 B LUT are adopted. Since the fused unit was synthesized to 90 nm CMOS fabrication technology, the area usage of the fused unit is reported in μm^2 (as an ASIC) with a clock frequency of under 500 MHz. We observe that the area of the multiply/divide fused unit is 37722.8 μm^2 while the area of the multiply/divide/square root unit is 45339.0 μm^2. There is an approximately 20% increase in area for the multiply/divide/square root unit when extended from the multiply/divide unit. Unfortunately, the accuracy for the square root unit alone is not mentioned in the paper.

Dinechin et al. [53] implement square root based on polynomial approximation only. Both inputs and outputs of are single-precision (32-bit) floating-point numbers. The square root has been correctly rounded to 0.5 ULP accuracy and uses degree-2 polynomial approximation. It operates at 237 MHz and takes 12 clock cycles (50.6 ns latency) to complete a single square root computation. The implementation is incorporated into the open-source FloPoCo framework [2].

Intel/Altera square root intellectual property (IP) operates at 263-310 MHz and takes 30.4-51.6 ns latency on the Intel/Altera Cyclone V FPGA. As a pipelined design, it accepts a new single-precision floating-point input at every clock cycle. The maximum error of the IP is 1 ULP for the inputs in the domain. Since the Intel/Altera IP is proprietary, we infer that it uses LUT approximation from the large amount of memory usage in their FPGA synthesis report.

Savas et al. [102] propose a single-precision floating-point square root unit. The hardware implementation uses two degree-2 polynomial and three LUTs, which take 528 bytes. The total number of arithmetic operations for LUT approximation and polynomial interpolation is five integer multiplications, seven integer additions and one shift operation. The square root unit is tested

Table 4–1: Comparison of our proposed and previous square root implementations

Authors	Year	LUT	Accuracy[1]		$(\times, +)$	Width	Exhaustive Testing?[2]	Throughput (MHz)	Pipeline?	Methodology
			ULP	Error bound						
Kown, Draper	2008	448 B	?[3]	?	(14, 5)	?	?	500	yes	ASIC
Dinechin et al.	2010	n/a	0.5	n/a	(2, 2)	(17, ?)	?	19.75	no	FPGA
Suresh et al.	2013	n/a	2048	n/a	(0, 78)	(0, 52)	?	6.6	no	FPGA
Hasnat et al.	2017	n/a	2048	n/a	(8, 3)	?	partial	194.122	yes	FPGA
Chandu, Maradi	2017	n/a	?	?	?	?	?	25	no	FPGA
Li et al.	2017	?	n/a	$2^{-20} \times 1.000...1$	?	?	partial	97.5	no	FPGA
Satpute	2018	n/a	?	?	?	?	?	50	no	FPGA
Intel IP	2018	?	1	n/a	?	?	yes	263-310	yes	FPGA
Savas et al.	2019	528 B	1	n/a	(5, 7)	?	partial	240	yes	FPGA
Our work[4]	2018	$\approx$ 1024 B	1	n/a	(2, 2)	(32, 64)	yes	104-244.92	yes	FPGA

[1] Accuracy is given in either ULP or an upper bound (in binary floating-point format) for each work presented in this table.

[2] Accuracy is obtained through exhaustive or partial testing. There are $2^{31} \approx 2$ billions testing cases for single-precision $\sqrt{x}$ in exhaustive testing. Partial testing uses random numbers to verify accuracy. The exhaustive testing approach allows us to make a stronger statement on the accuracy for $\sqrt{x}$ using low-order polynomial.

[3] "?" represents unknown.

[4] Our LUT-based square root accelerator.

with 2^{24} (16,777,216) single-precision floating-point inputs to obtain less than 1 ULP error. Furthermore, the hardware implementation is verified by implementation on the Xilinx UltraScale FPGA. Results show that the five-stage pipelined design operates at 240 MHz and has a latency of 20.8 ns to complete a single square root evaluation.

4.4.2 Iterative Algorithms

These approximate the square root via a number of calculation iterations. Each iteration is intended to give a more accurate result toward the final square root evaluation. For iterative approaches (e.g. Newton's method), finding a good starting point can effectively reduce the number of iterations required to generate a result of certain accuracy for the square root. The non-restoring method approximates the square root through a series of steps. The method initially computes a starting point $x_0 = 2^{n-1}$ for a 2n-bit number x. Then, it approximates the real value of $\sqrt{x}$ through $n - 1$ steps [43]. This method has been widely adopted to implement square root due to its simplicity. Satpute et al. [101] propose a single-precision (32-bit) floating-point square root implementation using the non-restoring method. It operates at 50 million samples per second (MSPS), and uses 84 LUTs and 141 SLICEs on the Xilinx Spartan 3E FPGA. However, the accuracy of the square root evaluation is not mentioned in their paper. Moreover, Suresh et al. [105] apply a non-restoring algorithm to their square root computing unit, which achieves an accuracy rate of 12 out of 23-bits in the mantissa (2048 ULP). The unit operates at 203.25 MHz and takes 31 clock cycles to complete. In addition, the unit consumes 819 LUTs, 276 SLICEs and 531 registers on the Xilinx Virtex-6 FPGA.

Hasnat et al. [68] present a fast implementation of single-precision (32-bit) floating-point square root. The design was coded in VHDL and leverages a modified Quake's algorithm [103, 44]. Here, the Quake's algorithm refers to a modified Newton's method, which targets to a 3D video game so-called Quake. It adopts magic numbers to make good guess for the starting point. The work is synthesized to the Xilinx Virtex 5 XC5VLX85T-3FF1136 FPGA. Experiments show that the design operates at 194.122 MHz and takes 12 cycles to get 12-bit accuracy (maximum error is 2048 ULP) for input ranges in [1, 2). In addition, the area consumed is 199 LUTs, 24 registers.

Chandu and Maradi [47] present an efficient square root computation unit based on the Vedic algorithm [106]. The Vedic algorithm is an iterative algorithm to find square root based on fast manual calculation. Neither input nor output comply with the IEEE-754 floating-point standard. In this case, they used a 32-bit (16-bit + 16-bit) floating-point input, and a 16-bit (8-bit + 8-bit) floating-point output. The design was coded in Verilog HDL, and synthesized to the Xilinx XST SPARTAN 6 family XC6SLX25T FPGA. The accuracy of the unit was not discussed. In addition, the square root unit consumes 360 LUTs and, operates at approximately 25 MHz.

The Coordinate Rotation Digital Computer (CORDIC) algorithm [89], invented by Jack E. Volder, has been extensively used in fields such as signal and image processing. CORDIC is an iterative algorithm that simply leverages shift-add operations for the computation of trigonometric functions, division, square root, etc. Although the latency of evaluating CORDIC may be long, the simplicity of the algorithm facilitates hardware implementation.

Xilinx square root intellectual property (IP) [14, 15, 102] is implemented using the CORDIC algorithm. After synthesizing to the Xilinx Virtex Ultra-scale FPGA, the IP consumes 448 LUTs and 786 flip-flops. In addition, the IP operates at 568 MHz and has a latency of 44.0 ns to complete a single square root operation. The IP is implemented in a 25-stage pipeline and accepts a new input every clock cycle.

Li et al. [81] implement a floating-point square root based on the CORDIC algorithm. The unit has 12 iterative layers for SAR (Synthetic Aperture Radar) imaging processing. Since the square root unit targets small numerical values, 200 random values with a range of [0, 100] are used to verify accuracy. Results show that all errors do not exceed 10^{-6}. The design was coded in VHDL and synthesized on the Xilinx XC7VX690T FPGA. The proposed unit consumes 1279 LUTs, 953 registers and requires four clock cycles to complete the computation, while operating at 390 MHz.

In addition to the aforementioned works, Mopuri et al. [90] discuss a CORDIC-based method to implement square root, where inputs are complex numbers. Bagala et al. [40] propose a square root algorithm based on a binomial series for 16-bit binary numbers.

4.5 Range Reduction

Similar to reciprocal, we also perform range reduction for square root. For a floating-point input *fp*, we have the equation of $\sqrt{fp}$ as follows:

$$\sqrt{fp} = \sqrt{(-1)^s \times 2^{e_{7-0}-127} \times 1.m_{22-0}} \tag{4.1}$$

To obtain valid results for square root, we assume that the sign of the *fp* is positive ($(-1)^s = 1$); then, we have:

$$\begin{aligned} \sqrt{+fp} &= \sqrt{2^{e_{7-0}-127}} \times \sqrt{1.m_{22-0}} && (4.2)\\ &= 2^{(e_{7-0}-127)/2} \times \sqrt{1.m_{22-0}} && (4.3)\\ or &= 2^{(e_{7-0}-127-1)/2} \times \sqrt{2} \times \sqrt{1.m_{22-0}} && (4.4) \end{aligned}$$

We have two equations 4.3 and 4.4 for the results of square root, depending on whether the biased exponent $(e_{7-0} - 127)$ is even or odd. When $(e_{7-0} - 127)$ is an even number, then it is easy to compute $\sqrt{2^{e_{7-0}-127}}$ as $2^{(e_{7-0}-127)/2}$ in Eqn. 4.3. However, when $(e_{7-0} - 127)$ is an odd number, then $(e_{7-0} - 127 - 1)$ is an even number. Thus, we compute $\sqrt{2^{e_{7-0}-127}}$ as $2^{(e_{7-0}-127-1)/2}$, and multiply $\sqrt{1.m_{22-0}}$ by $\sqrt{2}$ in Eqn. 4.4. Looking at Eqns. 4.3 and 4.4, we realize that our major work is to compute $\sqrt{1.m_{22-0}}$, whose range is $[1 : \sqrt{2})$. Eqn. 4.3 or 4.4 shrinks the domain of square root to $1.m_{22-0}$. To comply with floating-point accelerators from FPGA vendors (e.g. Xilinx, Intel/Altera) and the FloPoCo open-source library [54], our square root also does not support subnormal inputs – such inputs are flushed to zero.

4.6 Iterative Implementation: Newton-Raphson Method

Newton-Raphson method (Newton's method) is an iterative approach to find roots of a continuous function $y = f(x)$; that is, the method finds the locations, $x_a, x_b, \ldots,$ for which y evaluates to 0. Newton's method begins by making an initial guess, x_0, for a root. Then, at the guess location, $f(x_0)$ and the first-order derivative $f'(x_0)$ are evaluated. A new/better guess, x_1, is then computed as:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{4.5}$$

x_1 is the intersection of the line which is tangent to $f(x_0)$ and the x-axis.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.6}$$

Repeating the process in Eqn. 4.6 yields a successively better/equal approximation after each iteration. To evaluate square root $\sqrt{fp}$, we apply Newton's method to the function $f(x) = x^2$ - fp, $(fp \geqslant 0)$. Observe that, for this function, $f(x) = 0$ when $x = \sqrt{fp}$. Thus, the Newton's iterative equation is as follows:

$$x_{n+1} = \frac{1}{2}(x_n + \frac{fp}{x_n}) \tag{4.7}$$

For the starting point x_0, we use the following rough approximation from [24]:

$$\begin{aligned} log_2(fp) &= log_2(2^{e_{7-0}-127} \times 1.m_{22-0}) && (4.8)\\ &= log_2(2^{e_{7-0}-127}) + log_2(1.m_{22-0}) && (4.9)\\ &\approx log_2(2^{e_{7-0}-127}) && (4.10)\end{aligned}$$

It is easy to obtain $fp \approx 2^{e_{7-0}-127}$ in Eqn. 4.10. Since $\sqrt{fp} \approx 2^{e_{7-0}-127/2}$, we need to subtract the exponent of fp by 127, then divide it by two and finally adjust the value back to the biased exponent. Based on Eqn. 4.7, each iteration requires one division, one addition and one shift operation.

4.6.1 Case Study

In this section, a concrete example is given to evaluate the square root of 22.5 using Newton's method. We have Newton's iterative equation as follows:

$$x_{n+1} = \frac{1}{2}(x_n + \frac{22.5}{x_n}) \tag{4.11}$$

To approximate the starting point x_0, we use the methodology stated in Eqns. 4.8, 4.9 and, 4.10 from [24]. Here, we give x_0 in binary floating-point and decimal representations:

$$\begin{aligned} x_0 &= 0, 10000001, 00110100000000000000000 \quad (binary) && (4.12)\\ &= 4.8125 \quad (decimal) && (4.13)\end{aligned}$$

We know that x_0 is a rough approximation to $\sqrt{22.5}$. To obtain a better approximation, we apply Eqn. 4.11 to compute x_1:

$$x_1 = \frac{1}{2}(x_0 + \frac{22.5}{x_0}) \tag{4.14}$$

$$= 0, 10000001, 00101111100111000100001 \quad (binary) \tag{4.15}$$

$$= 4.743912 \quad (decimal) \tag{4.16}$$

After the first iteration of Newton's approximation, we further apply Eqn. 4.11 to compute x_2 and x_3.

$$x_2 = \frac{1}{2}(x_1 + \frac{22.5}{x_1}) \tag{4.17}$$

$$= 0, 10000001, 00101111100101000010001 \quad (binary) \tag{4.18}$$

$$= 4.7434163 \quad (decimal) \tag{4.19}$$

$$x_3 = \frac{1}{2}(x_2 + \frac{22.5}{x_2}) \tag{4.20}$$

$$= 0, 10000001, 00101111100101000010001 \quad (binary) \tag{4.21}$$

$$= 4.7434163 \quad (decimal) \tag{4.22}$$

We observe that Newton's approximation converges on x_2 after two iterations of computation. In this case, x_2 has 0 ULP error compared with the square root of 22.5 generated by the GNU `math.h` library. However, three iterations are required to achieve a 1 ULP error approximation for all square root inputs (as will be demonstrated below in the accuracy analysis section). For an input value which is subnormal, (± zero, ± infinity or NaN), we apply the exceptions listed in Table 4–2. For example, if an input is a negative number, then the corresponding output is NaN (Not-a-Number). In the IEEE-754 standard, the 32-bit encoding for NaN is 0xffc00000 in hexadecimal. However, to match the results from the GNU C `math.h` library, when inputs of the square root function are NaNs, the output is *fp* || 0x00400000.

Table 4–2: Exception handling for square root. [50]

Input	Output	Encoding
+subnormal	+zero	0x00000000
+zero	+zero	0x00000000
+infinity	+infinity	0x7f800000
negative	NaN	0xffc00000
NaN	NaN	*fp* || 0x00400000

4.7 Non-Iterative Implementation: Lookup Table (LUT)

In addition to Newton's iterative approximation, we also implement square root by using a LUT along with polynomial interpolation. Here, we borrow the techniques of implementing a LUT-based reciprocal, as described in the previous chapter. Compared with iterative implementation, the LUT-based square root is faster in evaluation time and smaller in circuit area.

Recall that for square root, as shown in both Eqns. 4.3 and 4.4, we need to compute $\sqrt{1.m_{22-0}}$, whose domain is [1:2) and range is [1:$\sqrt{2}$). Similar to the LUT-based reciprocal, to compute $\sqrt{1.m_{22-0}}$, a LUT is created and divided into 64 intervals. Within any interval, a degree-2 polynomial ($y = a \cdot x^2 + b \cdot x + c$) is used to interpolate the missing values. Here, a, b, and c are scalar coefficients, and they are different for each of the intervals. We use the 6 MSBs of mantissa (i.e. m_{22-17}) to address the LUT for coefficients (a, b, c), while the remaining 17-bits of the mantissa (i.e. m_{16-0}) are treated as x for the degree-2 polynomial interpolation.

Recall from Eqn. 4.4 that, there is an additional multiplication to $\sqrt{1.m_{22-0}}$ by $\sqrt{2}$, when the biased exponent is odd. In this case, we tried two kinds of approximation to $\sqrt{2} \cdot \sqrt{1.m_{22-0}}$ for precision: 1) $\sqrt{2} \cdot (a \cdot x^2 + b \cdot x + c)$, and 2) $(\sqrt{2} \cdot \mathrm{a}) \cdot x^2 + (\sqrt{2} \cdot \mathrm{b}) \cdot \mathrm{x} + (\sqrt{2} \cdot \mathrm{c})$. However, the latter approximation yields higher precision. The former approximation is implemented as an additional multiply with $\sqrt{2}$ to the result of polynomial interpolation. The latter approximation multiplies each coefficient of polynomial interpolation by $\sqrt{2}$. Thus, it actually uses two sets of coefficients, depending on whether the biased exponent is an even or odd number. The latter approximation achieves higher precision because it avoids rounding errors caused by an additional multiply, but it suffers from the cost of doubling the LUT size by storing two sets of coefficients.

Similar to the LUT-based reciprocal, we use Chebyshev polynomial approximation to compute coefficients (a, b, c) for each interval. In addition, we use Horner's rule to reduce one multiplication for degree-2 polynomial interpolation from $y \approx a \cdot x^2 + b \cdot x + c$ to $y \approx ax \cdot (x+b) + c$. Exception handling of the LUT-based approach is the same as that of Newton's iterative approximation.

4.8 Error Study

In this section, we analyze the accuracy of the square root accelerators through exhaustive testing. Similar to the error study of the reciprocal accelerators, error distributions are presented in terms of maximum/average error and the percentage of evaluations which have 0 or 1 ULP error.

4.8.1 Exhaustive testing

We also perform exhaustive testing on the domain of square root to obtain an overall error distribution. Our square root accelerator does not support subnormals, and there are roughly 16,777,214 subnormal numbers according to the IEEE-754 standard. Thus, valid inputs of square root are approximately 2×10^9 cases, which exclude subnormal and negative numbers. Similar to reciprocal, we treat square root evaluations generated by the GNU `math.h` library as a golden benchmark, and we compare our Newton's and LUT-based approaches with the benchmark, to generate an overall error distribution.

4.8.2 Error Distribution

Table 4–3 shows the error distribution of the square root implementations. To simplify, we only show results of *normal* inputs according to the IEEE-754 standard. For Newton's iterative approximation, we observe that 75% of inputs produce no error (0 ULP), while the remaining 25% of inputs produce 1 ULP error. In this case, maximum and average error are 1 ULP and 0.25 ULP, respectively. The LUT-based implementation shows two results depending on whether the biased exponent is odd or even. If it is odd, then 73% of inputs produce no error (0 ULP), and 27% of inputs produce 1 ULP. We observe that the maximum and average errors are 1 ULP and 0.27 ULP, respectively. Similarly, if the biased exponent is even, maximum and average errors are 1 ULP and 0.24 ULP, respectively. Finally, in terms of average error, Newton's iterative approximation sometimes has a slightly higher average error than that of LUT-based implementations: 0.25 ULP vs. 0.24-0.27 ULP.

Table 4–3: Error distribution for square root [50]

Newton's Square Root				
Exp	Max Error	Avg Error	0 ULP	1 ULP
all	1 ULP	0.25 ULP	75%	25%
LUT-Based Square Root				
Exp	Max Error	Avg Error	0 ULP	1 ULP
odd	1 ULP	0.27 ULP	73%	27%
even	1 ULP	0.24 ULP	76%	24%

4.9 Experimental Study

We compare our square root accelerators with Intel/Altera IP cores, in terms of speed and circuit area. Similar to the reciprocal accelerators shown in Table 3–4, three configurations of square root accelerators are produced: fastest core, smallest core and core in-between. All cores are targeted to the Intel/Altera Cyclone V 45 nm FPGA. Also, they are synthesized into hardware using the LegUp HLS, Quartus and ModelSim. We present the performance of

Table 4–4: Intel vs. Our LUT-based square root accelerators [50]

	Intel/Altera FP_FUNCTIONS					
	Throughput	**Latency**	**ALMs**	**Mbits**	**DSPs**	**eALMs**
Smallest	263.1	8	83	15,872	2	238.67
Midway	277.9	9	100	15,872	2	255.67
Fastest	310.1	16	178	15,872	3	363.67
	Our work (handling exceptions)					
	Throughput	**Latency**	**ALMs**	**Mbits**	**DSPs**	**eALMs**
Smallest	104	6	94	10,624	2	217.78
Midway	149.84	6	112	10,624	2	235.78
Fastest	244.92	10	162	10,624	2	285.78
	Our work (no exception handling)					
	Throughput	**Latency**	**ALMs**	**Mbits**	**DSPs**	**eALMs**
Smallest	98.75	5	89	10,624	2	212.78
Midway	163.61	7	117	10,624	2	240.78
Fastest	191.39	8	135	10,624	2	258.78

square root cores in terms of maximum throughput, latency (pipeline depth), ALMs, memory bits (Mbits), DSPs and equivalent ALMs (eALMs).

4.9.1 LUT-Based Square Root Accelerator

Table 4–4 compares our LUT-based accelerators with Intel/Altera IP cores. For the fastest cores, we observe that the Intel/Altera IP core operates at 310.1 MHz, has 16 cycles of pipeline depth, and consumes 178 ALMs, 15.8K Mbits and three DSPs, for a total circuit area of approximately 363.67 eALMs. Our fastest accelerator with exception handling operates at 244.92 MHz, has 10 cycles of pipeline depth and, consumes 162 ALMs, 10.6K Mbits and two DSPs, for a total circuit area of approximately 285.78 eALMs. Similarly, for the smallest core, we observe that the Intel/Altera IP core operates at 263.1 MHz, has eight cycles of pipeline depth and, consumes 83 ALMs, 15.8K Mbits and two DSPs, for a total circuit area of approximately 238.67 eALMs. Our smallest accelerator with exception handling operates at 104 MHz, has six cycles of pipeline depth and, consumes 94 ALMs, 10.6K Mbits and two DSPs, for a total circuit area of approximately 217.78 eALMs. In both cases,

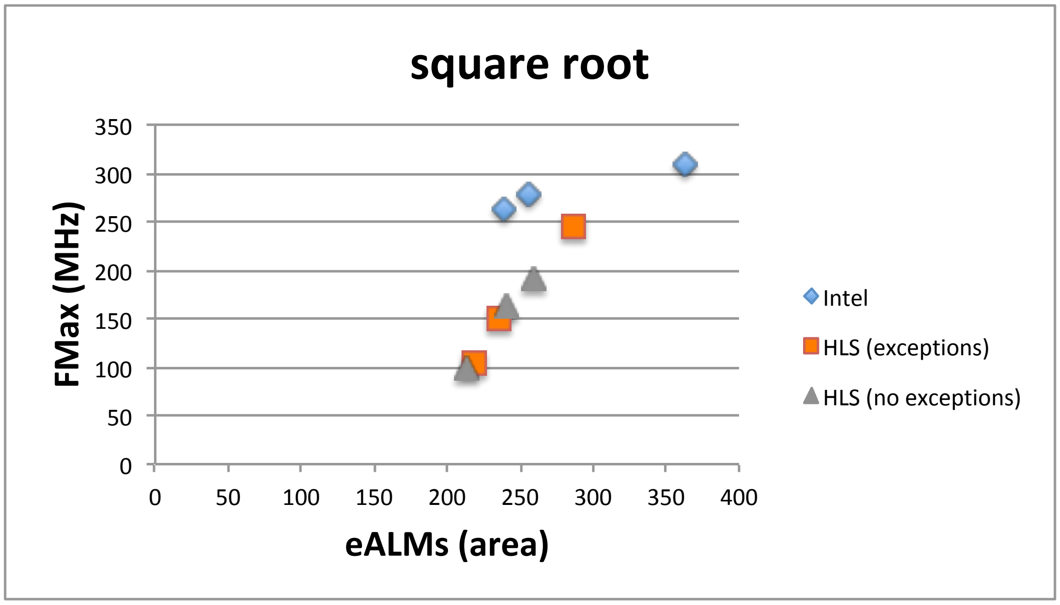

Figure 4–1: Throughput vs. area (eALMs) trade-offs for square root accelerators [50]

the Intel/Altera IP cores again provide a higher throughput, and also consume fewer ALMs in some cases. However, our accelerators offer a reduced pipeline depth and, fewer memory bits, DSPs and eALMs. Table 4–4 also presents the accelerators without exception handling. We observe that there are reductions in pipeline depth and ALMs after exception handling has been removed.

Figure 4–1 illustrates the performance of Intel/Altera IP cores and our accelerators (with and without exception handling). Again, the blue dots represent Intel/Altera implementations, while orange and grey dots represent our implementations with and without exception handling, respectively. In this case, Intel/Altera implementations operate at a higher throughput, and also have less circuit area in some cases. On the other hand, our accelerators operate at a lower throughput, and have less circuit area in most cases.

4.9.2 Iterative Square Root Accelerator

Table 4–5 shows the results of square root implementations using Newton's iterative approximation. There are two groups: the left-hand group shows the results with exception handling, while the right-hand group shows

Table 4–5: Newton's iterative implementations of square root – with and without exception handling [50]

	With exceptions			**Without exceptions**		
	Throughput	**Latency**	**ALMs**	**Throughput**	**Latency**	**ALMs**
Smallest	106.1	29	1,776	104.82	28	1,716
Midway	167.34	57	2,231	159.72	55	2,015
Fastest	216.54	169	5,409	217.49	165	4,749

the results without exception handling. For square root, with exception handling, we observe that the throughput is 106-216 MHz, pipeline depth is 29-169 cycles, and ALMs are 1,776-5,409. Without exception handling, we observe that maximum throughput is 104-217 MHz, pipeline depth is 28-165 cycles, and ALMs are 1,716-4,749. After exception handling is removed, there are reductions in pipeline depth and ALM usage.

As expected, iterative implementations of square root operate at a lower throughput, have longer pipeline depth, and use considerably more ALMs. However, iterative implementations are entirely built on soft logic (ALMs), and require neither Mbits nor DSP units. For applications for which ALM usage is critical, LUT-based implementations are superior, owing to the fact that ALM usage in Newton's iterative approximation is nine to 24 times that of the LUT-based approach. However, from the perspective of Mbit/DSP usage, Newton's iterative approximation is preferred because it does not require memory blocks or DSP units.

4.10 Performance Comparison

Table 4–1 demonstrates performance of our proposed and previously mentioned square root accelerators. The last row in the table refers to our LUT-based accelerators. In Table 4–1, we observe that most implementations have n/a in the column of LUT. This is because for an iterative implementation, is not necessary to have LUT for evaluation. Kown and Draper uses a smaller

LUT than us, but they require roughly many more multiplications and additions vs. our approach. In addition, accuracy is not explicitly mentioned in their paper. Similarly, Savas et al. also present a smaller LUT, but requires more multiplications and additions. Both Intel IP and our proposed accelerators achieve 1 ULP maximum error through exhaustive testing. We note that Dinechin et al. apparently offers better accuracy, however, it is not clear if they also perform exhaustive testing to verify the accuracy. Similarly, Savas et al. claim an error of 1 ULP through partial testing.

To give a fair and quantitative comparison among various implementations of square root, evaluation is quantified by presenting the number and width of multiplication and addition operations. In this case, our LUT-based implementation uses fewer multiplications and additions than the majority of the prior work. It has two 32-bit multiplications, one 32-bit addition and one 64-bit addition. The accelerator proposed by Suresh et al. has 78 52-bits additions without using any multiplications. The maximum throughput of our LUT-based accelerator is higher than all FPGA-based implementations except Intel/Altera IP cores. Kown and Draper's work has highest throughput in Table 4–1. However, their accelerator is based on an ASIC instead of an FPGA.

It is worth to note that non-pipelined implementations generally have lower throughput than pipelined ones. For example, the accelerator presented by Li et al. operates at a frequency of 390 MHz and requires four clock cycles to complete one single computation of square root. In this case, the throughput is approximately 97.5 MHz although the accelerator operates at a fairly high frequency. In our case, the LUT-based accelerator adopts pipelined architecture, and the pipeline accepts a new input of square root at every clock cycle. Hence, the initiation interval (II) of our proposed accelerator is equal to one (II = 1). However, it may not be true that other pipeline accelerators in the table also have II = 1.

4.11 HLS C Implementation

In this section, we present a number of HLS C code snippets for the square root accelerators. For performance optimization, we replace floating-point division with fixed-point subtraction in Newton's iterative implementation, which helps to reduce ALM usage dramatically. Likewise, avoiding the use of conditional statements and floating-point addition also results in large ALM usage reductions. In addition, modifications have been made to the interpreted Verilog code from HLS to reduce DSP usage on target FPGA.

- **Loop unrolling.** Recall that we use Newton's approximation to implement the iterative square root accelerator. In LegUp HLS version 5.5, we need to manually unroll all loop statements, and remove all the corresponding control flow as well. The following code snippet shows an unrolled loop for three iterations of Newton's approximation. In versions later than LegUp HLS 5.5, a manual loop unroll is no longer required, as the tool has been enhanced to perform unrolling automatically.

```
/*
for(i = 1; i < = 3; i++)
{
  div = divide(a, xn); // a/xn
  xn = add(xn, div); // xn + a/xn
  xn = xn - 0x00800000; // xn/2
}
*/
// 1st iteration
div = divide(a, xn);
xn = add(xn, div);
xn = xn - 0x00800000;
```

```
// 2nd iteration
div = divide(a, xn);
xn = add(xn, div);
xn = xn - 0x00800000;
// 3rd iteration
div = divide(a, xn);
xn = add(xn, div);
xn = xn - 0x00800000;
```

The commented code shows the loop implementation for Newton's approximation in traditional `C`. To simplify, we skip the process of finding the starting point for `xn`. `xn` is implemented as a 32-bit unsigned integer, but treated as 32-bit floating-point number. The `divide` and `add` functions implement floating-point division and addition in fixed-point arithmetic. Here, we use `xn - 0x00800000` to replace `xn/2`, since fixed-point subtraction is less expensive than floating-point division. By doing so, we observe an approximately 200- to 300- ALM usage reduction in the HLS synthesis report.

- **Reducing ALM usage.** Recall that we manually replace the `if-then-else` statements with the expression `<cond>?<val1>:<val2>`. LegUp HLS translates a `<cond>?<val1>:<val2>` statement into a multiplexer, which consumes soft logic (ALMs). To reduce ALM usage, we need to avoid the use of conditional statements as much as possible. Note that in versions later than LegUp HLS 5.5, manual replacement of `if-then-else` is no longer required, as the HLS tool has been improved to perform such changes internally and automatically. The following code snippet shows how to do floating-point addition by using fixed-point arithmetic in an area-wise manner for Newton's approximation. For floating-point addition, $fp_1 + fp_2$, one needs to complete in four steps: 1) compare the magnitudes of fp_1 and fp_2, then scale the number with less magnitude, 2) add the mantissas of fp_1 and fp_2, 3) adjust the result according to the IEEE-754 standard, and 4) perform rounding. In our case, we implement floating-point addition by assuming the exponent of fp_1 is greater or equal to that of fp_2. In doing so, the control flow used to do a comparison between fp_1 and fp_2 in the first step is eliminated, which results in a significant reduction in ALM usage.

```
// a, b are both single-precision (32-bit) floating-point
    numbers
uint32_t add(uint32_t a, uint32_t b)
{
  ...
  // a_exp, b_exp represent exponents of a, b respectively
  diff_exp = a_exp - b_exp;
  result_exp = a_exp;
  // a_mantissa, b_mantissa represent mantissas of a,b
     respectively
  b_mantissa = b_mantissa >> diff_exp;
  result_mantissa = a_mantissa + b_mantissa;
  ...
}
```

- **Rounding.** After the second multiplication of degree-2 polynomial interpolation ($y \approx (a \cdot x + b) \cdot x + c$), we shift the product $value = (a \cdot x + b) \cdot x$ right by 23-bits. The following code snippet shows the crude rounding scheme we adopt. Before doing the right shift, we need to look at the 22nd bit of the product $value = (a \cdot x + b) \cdot x$. After that, we need to perform the 23-bit shift right. If the 22nd bit is bit-1, then the shifted product is incremented. Otherwise, the shifted product remains the same. We use a similar rounding scheme for the final addition with coefficient c for degree-2 polynomial interpolation.

```
// check if 22nd bit of product is bit-0 or bit-1
unsigned char roundUp = (value >> 22) & 0x1;
// shift the product right by 23-bit
value = value >> 23;
// if the 22nd bit of product is bit-1, then round up to (
   value + 1); otherwise, round down to (value);
value = (roundUp) ? value + 1 : value;
```

- **Reducing DSP usage.** For Intel/Altera Cyclone V FPGA, one DSP slice can implement a 27×27-bit multiplication at most on the Cyclone V FPGA we target. Thus, multiplication with a wider bitwidth requires more than one DSP slice. This may lead to more ALM usage due to additional soft logic being needed to accumulate intermediate products. For DSP usage considerations, we need to keep operands of multiplication below 27-bits. Recall that degree-2 polynomial interpolation, $y \approx (a \cdot x + b) \cdot x + c$, is used for LUT-based implementations. Coefficient a is represented using 27 bits because it does not have a large variance in its exponent. Variable x is the 16 or 17 least significant bits (LSBs) of the mantissa. Hence, the first multiplication, $a \cdot x$, easily meets the 27-bit requirement. Likewise, the second multiplication is also kept within the 27-bit boundary.

  Owing to the fact that we use 32-bit unsigned types in C language, 0s appear in all MSBs for operands of multiplication. We observe that the HLS tool translates operands of multiplication into 32-bit wide signals in Verilog. Similarly, Quartus does not recognize those leading 0s in operands of multiplication, leading to additional DSP usage after translating into hardware on the target FPGA. To this end, we manually modify the HLS-interpreted Verilog code to narrow down the bitwidth used for operands, making each multiplication in the degree-2 polynomial interpolation use only one DSP slice.

It is well known that `C` code must be structured in a specific style to produce a high-quality circuit in Verilog via HLS. Regarding the manual changes to the interpreted Verilog, it was a few lines change to adjust the bitwidths in a file with thousands of lines of HLS-interpreted Verilog code. Such minor tweaks to code are normal for the state-of-the-art HLS.

4.12 Summary

In this chapter, we present single-precision floating-point square root hardware accelerators. Two algorithms for computing square root are designed: 1) iterative (Newton's method), and 2) LUT along with degree-2 polynomial interpolation. Similar to the reciprocal hardware accelerators, the square root accelerators are synthesized into hardware targeting Intel/Altera 45 nm FPGA using the Legup HLS, Quartus and Modelsim. We treat square root results from the GNU C `math.h` as a baseline; our accelerators achieve 1 ULP maximum error in comparison with the baseline through exhaustive testing. In evaluation, we compare the performance of our LUT-based accelerators with the state-of-the-art implementations, particular Intel/Altera IP cores. Results show that our accelerators win in pipeline latency and circuit area, especially memory and DSP usage. However, Intel/Altera IP cores win slightly in maximum throughput.

Chapter 5
RISC-V Soft Processor

5.1 Publication

Main content of this chapter comes from the following submitted manuscript:

Jing Chen, Jason H. Anderson, "High-Level Synthesis of a Lightweight FPGA-Based RISC-V Processor", the 30th International Conference on Field-Programmable Logic and Applications (FPL'2020), **under review**.

5.2 Introduction and Organization

In this chapter, we propose a 32-bit RISC-V multi-cycle processor implementation based on FPGA, which is a full realization of a 32-bit integer base instruction set (RV32I) and has 39 user instructions. The processor design is implemented using C language, and synthesized into hardware targeting Intel/Altera 28 nm FPGA using the LegUp HLS, Quartus and Modelsim. Custom testing programs are developed to verify if each instruction adheres to the RISC-V specification. In addition, we compare the proposed processor with two open source RISC-V implementations in terms of maximum frequency, circuit area, etc.

In Section 5.3, we present the motivation of adopting the RISC-V ISA. In Section 5.4, we give an brief introduction to the RISC-V ISA, which includes background, software ecosystem and ISA features. Previous soft processor implementations are discussed in Section 5.5. The architecture of the proposed RV32I multi-cycle processor is elaborated in Section 5.6. We show representative HLS-C code snippets in Section 5.7 to demonstrate several optimizations

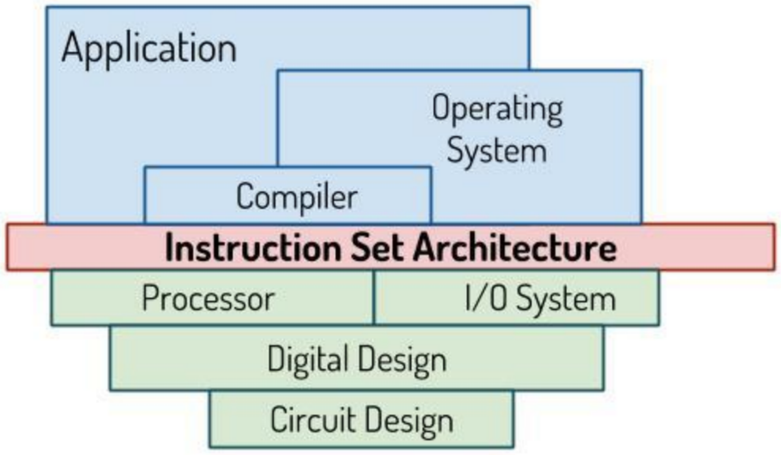

Figure 5–1: ISA serves as the interface between software and hardware [62]

made to the RV32I processor. Custom testing programs are described in Section 5.8. Lastly, we discuss the performance (i.e. area, frequency) of the RV32I multi-cycle processor in Section 5.9. We give a performance comparison between ours and two open source RISC-V implementations in Section 5.10, and conclude this chapter in Section 5.11.

5.3 Motivation

An ISA (Instruction Set Architecture) is a software/hardware interface for the computer system as shown in Figure 5–1. It comprises an entire set of assembly commands that order the CPU to realize specific functions. The ISA defines almost everything for building the CPU architecture, such as the number and function of registers, arithmetic/logical operations, subroutine calling and stack manipulation conventions, etc. The various implementations of an ISA may differ in throughput, area and power consumption.

RISC-V is an open-source ISA that arose from research at the University of California, Berkeley, commencing in 2010. RISC-V has garnered significant attention in recent years, and the ISA has a rapidly expanded software ecosystem (e.g. compilers, debuggers, etc.). For this research, we opted for the

RISC-V [32] ISA for the following reasons: 1) open-source (unlike Intel and ARM ISAs, which are proprietary), 2) extensible, 3) efficient in all domains of computing, including server and mobile/embedded [32], 4) efficient for all implementation technologies [32], including ASIC and FPGA.

5.4 Introduction of RISC-V ISA

RISC-V [32] is an open and free RISC ISA for education, research and commercial purposes. The RISC-V Foundation, a non-profit organization, was founded in 2015 and now has more than 500 members (including IBM, Google, and Nvidia) globally. Early funding for the development of the RISC-V project came from DARPA (The Defense Advanced Research Projects Agency). The RISC-V Foundation aims to build a collaborative community to promote RISC-V ISA and drive the development of the related software/hardware ecosystem. Unlike other ISAs, which target certain types of applications, the RISC-V ISA is developed to accommodate all domains of computing, such as server, mobile and embedded applications. For example, the NVIDIA Falcon processor [13] operates at a frequency of approximately 3 GHz. It achieved a *2*× performance improvement and area reduction after adopting a 64-bit RISC-V ISA. Likewise, one can obtain high-performance, energy-efficient realizations from the RISC-V ISA regardless of micro-architecture (i.e. out-of-order) or implementation technology adopted (i.e. FPGA, ASIC, custom hardware).

5.4.1 Features of the RISC-V ISA

The RISC-V ISA is extensible and configurable. Most existing ISAs have a fixed number of instructions. Adding new instructions to an ISA may degrade a processor's execution performance because patches to the ISA need to be made. In the extreme, adding new instructions to an ISA may require a complete redesign, if there are not enough operation codes for the future instructions. RISC-V, on the other hand, has been designed in way that is extensible to new/future instructions.

RISC-V can be configured as base plus extended instruction sets. Here, extension sets include integer multiply/divide (M), atomic (A), single-precision (F), double-precision (D) and quad-precision (Q) floating-point, etc. In addition to the complete extension sets, there are sufficient operation codes reserved for future instructions. Moreover, the RISC-V ISA can be configured as a 32-bit, 64-bit or 128-bit architecture, depending on the target computing domain. One can therefore think of the RISC-V ISA as a "family" of processors, with various datapath widths and functional capabilities.

The ecosystem of RISC-V ISA is growing rapidly. The ecosystem of an ISA refers to both the software and hardware which allow system developers to leverage new devices or platforms easily [115]. For software ecosystem, an ISA defines a series of software development tools, such as ISA simulator, compilers, debuggers, operating systems. The software ecosystem of RISC-V includes, but is not limited to, the GCC/LLVM compiler, Linux OS and QEMU as a simulator [32]. There is ongoing continued development in all of these areas: compiler/IDE/library, operating system, simulator/toolchain, etc [32].

5.4.2 Overview of RISC-V ISA

Most traditional ISAs have not been designed for instruction expansion. Hence, they may suffer from inefficient instruction decoding and execution as their instruction set grows. The RISC-V ISA overcomes this issue by proposing the idea of an extensive instruction set.

RISC-V is configured as base plus extension instruction sets. [62] The base instruction sets (RV32I/RV64I/RV128I) realize the minimum implementation of a RISC processor. A base set handles arithmetic/logical operations, branch, jump and link, memory read/write, etc. The extension instruction sets (M/A/C/F/D/Q) provide special instructions for complicated functions. For example, the extension "M" consists of instructions for doing integer multiplication and division. To configure a RISC-V ISA, one needs to choose one base and zero or more extension instruction sets.

Base instruction sets can be configured as a 32-bit, 64-bit or 128-bit system. By doing so, a system is able to accommodate computing in various domains.

1. **RV32I:** 32-bit base integer instruction set, which is sufficient to support compiler and operating system. RV32I has 47 instructions in total, which include 39 user and eight system instructions.
2. **RV64I/RV128I:** 64-bit/128-bit base integer instruction sets, which are developed on the base of RV32I. RV64I/RV128I provide more integer registers and a wider address space (i.e. 64-bit). RV128I was invented for warehouse-scale computers, which contain more than 1PB of DRAM.

Extension instruction sets support single, double, and quad floating-point arithmetic. Those extensions aim to accelerate custom applications, such as image/audio processing, deep learning, etc [62].

1. **F extension:** a single-precision (32-bit) floating-point instruction set. It defines basic operations for floating-point numbers, such as status check, load/store, value comparison, rounding, etc. As well, it provides instructions that perform addition, subtraction, multiplication and division for floating-point numbers.

2. **D/Q extensions:** double (64-bit)/quad (128-bit)-precision floating-point instruction sets. Instructions from D/Q extensions are similar to those of the F extension, except that the operands/outputs are double/quad-precision numbers.

5.5 Related Work

There are numerous soft-processor implementations on FPGAs, as shown in Table 5–1, which can generally be categorized as vendor-specific and non-vendor-specific ones [117]. FPGA-vendor-specific cores, such as NiosII [28] and MicroBlaze [19], have low resource usage and unremarkable operating frequency due to the adoption of simple fixed-pipeline architectures. The frequency of the NiosII can be tuned to 240 MHz on the Intel/Altera Stratix IV FPGA [117]. Since most vendor-specific soft processors are not open source, it prevents researchers from doing in-depth investigations of the architecture to further explore resource/speed optimizations and accelerator integration. The LEON3/LEON4 [26], non-vendor-specific cores, are open source but were originally designed for an ASIC implementation. Although LEON3/LEON4 can be implemented on an FPGA, they are not well suited to the underlying FPGA architecture [87].

Given the flexibility of the RISC-V ISA, there is an increasing number of FPGA implementations for the RISC-V processor. As mentioned above, the RISC-V consists of various extensions, such as multiply and divide (M), atomic (A), compression (C), single-precision (F), double-precision (D) and quad-precision (Q) floating-point. In the following sections, we briefly introduce some popular RISC-V soft processor implementations in terms of the supported instruction sets, architecture, operating frequency, etc.

Table 5–1: Summary of previous 32-bit soft processor implementations

Vendor-Specific Cores					
Cores	Frequency (MHz)	FPGA?	Vendor	Word Length	Open Source?
NiosII	240	Yes	Intel	32-bit	No
MicroBlaze	232-724	Yes	Xilinx	32-bit	No
Non-Vendor-Specific Cores					
Cores	Frequency (MHz)	FPGA?	Pipeline?	RISC-V?	Open Source?
LEON3/4	125	Yes	7-stage	No	Yes
Rocket	25-100	Zynq	5-stage	I/M/A/F/D	Yes
GRVI	375	Kintex UltraScale	Yes	I	No
OCRA	75	Zynq	4-/5-stage	I/M	Yes
PicoRV32	172	Zynq	4-/5-stage	I/M/C	Yes
FWRISCV	89	Cyclone V	No	I	Yes
VexRISCV	170-233	Artix 7	Yes	I/M/C/A	Yes
Taiga	232-236	Arria 10	Yes	I/M/A	Yes
BRISCV	unknown	Yes	5-/7-stage	I/F	Yes

RISC-V Rocket processor. The Rocket chip [38] is an open-source, general-purpose processor generator which is written in the Chisel [39] hardware construction language. The Rocket chip features two processor generators: one is an in-order core generator (the Rocket), while the other is an out-of-order core generator (the BOOM). The Rocket core supports RV32G and RV64G, while the BOOM core only supports RV64G. Here, "G" refers to the RISC-V ISA extensions (I, M, A, F, D). In addition, the Rocket core has a five-stage pipeline architecture. Both the Rocket and BOOM cores support virtual memory and non-blocking caches. The Rocket core can be deployed to the Xilinx Zynq-7000 series FPGAs, achieving clock frequencies of 25-100 MHz for various configurations on the FPGA. Three mechanisms are available to integrate custom accelerators into the Rocket or BOOM cores: 1) integrate the accelerators directly into the five-stage pipeline by extending the RISC-V ISA; 2) incorporate the accelerators into a co-processor and communicate with the RISC-V core via a RoCC interface; and 3) implement the accelerators as independent cores and connect them to the memory system.

GRVI/Phalanx. Gray [65] implemented an FPGA-efficient RISC-V RV32I soft processor, the GRVI RV32I. The RV32I is the 32-bit RISC-V integer instruction set. The design goal of the GRVI RV32I processor was to minimize the area usage and maximize the operating frequency. Evaluation shows that the GRVI RV32I consumes 320 LUTs and runs at up to 375 MHz on the Xilinx Kintex UltraScale FPGA. In addition, the processor has 0.7 MIPS/LUT. Here, MIPS refers to millions of instructions per second. Since the design of the GRVI core is not open source, we cannot do an in-depth investigation on the fixed-pipeline architecture. Phalanx is a framework which consists of processor and accelerator clusters. The clusters communicate via the Hoplite NOC [73, 74] in the FPGA. A Phalanx which contains 400 GRVI RV32I cores was implemented on the Xilinx Kintex UltraScale KU404 FPGA. In this case, there were 10 $\times$ 5 clusters of eight GRVI RV32I cores in the Phalanx. In total, the peak throughput of the 400 cores reaches 100,000 MIPS (million instruction per second).

ORCA [29] is a platform-independent open-source RISC-V implementation. It can be configured to either RV32I or RV32IM. ORCA has a fixed-pipeline architecture and supports a data cache and local memory. It can be configured as either a four- or five-stage pipeline processor. A five-stage ORCA processor operates at 75 MHz on the Xilinx ZYNQ, or 218 MHz on the Intel Arria 10 FPGA.

PicoRV32 was proposed by Wolf et al. [30]. It is open source and implements the RISC-V RV32IMC instruction sets. PicoRV32 supports the an AXI4 interface [9] and local memory. It has been observed that the resource usage of PicoRV32 is between the four-stage and five-stage ORCA cores. In addition, PicoRV32 runs at 172 MHz on the Xilinx ZYNQ, or 299 MHz on the Intel Arria 10 FPGA.

Featherweight (FWRISCV, for short) [18] is a multi-cycle RISC-V implementation. The design of FWRISCV is optimized for FPGAs. FWRISCV supports the RV32I instruction set and various exceptions specified by the RISC-V ISA. An experiment shows that FWRISCV consumes 616 ALMs, 241 registers and 4,096 memory bits when targeted to the Intel/Altera Cyclone V FPGA. Its maximum frequency is approximately 89 MHz. In addition, it is reported that FWRISCV achieves 0.15 DMIPS/MHz. Here, DMIPS/MHz refers to how many complete Dhrystone runs can be done per second. Dhrystone is a well-known integer benchmark, so DMIPS/MHz is used to measure the computing capacity for integer programs.

VexRISCV [21] is an open-source, 32-bit RISC-V soft processor. The VexRISCV ISA implements the RV32IMCA instruction set. The Atomic extension ("A") provides atomic load and store instructions that allow multiple RISC-V threads running with shared-memory and enforces a memory consistency model [32, 63]. As such, VexRISCV supports caches and a MMU (Memory Management Unit). In addition, VexRISCV can be configured for various implementations. The smallest core (RV32I) runs at 233 MHz, and consumes 494 LUTs and 505 flip-flops on the Xilinx Artix 7 FPGA. The full core (RV32IMA, Linux OS) runs at 170 MHz and consumes 2530 LUTs and 2013 flip-flops on the same FPGA board. When instruction level parallelism (ILP) is considered, the smallest core has 0.52 DMIPS/MHz, and the full core has 1.21 DMIPS/MHz.

Taiga is a 32-bit RISC-V FPGA soft processor [86, 87], supporting the RISC-V Integer and the Multiply/Divide and Atomic extensions (RV32IMA).

Taiga is designed to support shared-memory systems, TLBs (translation lookaside buffers), caches and MMUs (memory management units), which are necessary to build a Linux OS upon. In addition, Taiga features a variable-length pipeline design, wherein the ALU (arithmetic logic unit) consists of multiple parallel variable-latency computing units. This architecture not only supports out-of-order execution, but also facilitates a more tightly-coupled integration of accelerators. Taiga's performance was evaluated on the Xilinx ZYNQ X7CZ020 and Intel Arria 10 (GX115) FPGAs. It was observed that Taiga's resource usage (with different configurations) is equivalent to OCRA and PicoRV32. Furthermore, Taiga's operating frequency is higher OCRA's, but lower than PicoRV32's, which runs at 109-120 MHz on the ZYNQ, or 232-236 MHz on the Arria 10 FPGA, respectively. However, Taiga's IPC (instructions per cycle) is at least two times higher than that of PicoRV32. When Taiga is integrated with one-to-four accelerator cores, it shows a 3-6% drop in the operating frequency. One advantage of a processor with a variable-length pipeline is the ability to integrate multiple accelerators without significantly changing the frequency.

BRISCV. The Boston University RISC-V Processor Set (BRISC-V) [31, 34, 41] is a parameterized design for RISC-V ISA processors. The design of BRISC-V is implemented as Verilog HDL modules such that the processor architecture can be changed by different parameter settings. As such, BRISC-V can be configured as various RISC-V cores, for example, an RV32I single-cycle core, an RV32I five-stage pipeline core, an RV32I seven-stage pipeline core and an RV32IF out-of-order core. Those cores support parameterized cache subsystems, which means the size, associativity and levels of the cache can be configured by parameters as well. We have chosen the ORCA, PicoRV32 and FWRISCV cores for comparison since they can generate RV32I cores with both simple and complex architectures.

5.6 Architecture of RV32I Processor

In this section, the architecture of the RV32I multi-cycle processor is elaborated. We first introduce the user register file, highlighting the specific function of each register. Then, we present the configuration of the memory system that adopts a Harvard architecture. Finally, we describe the RV32I instructions in terms of their format and functions.

5.6.1 User Register File

As shown in Figure 5–2, the RV32I processor contains 32 general purpose user registers, and each register has a bitwidth of 32. The functions of some registers are described below. Registers that are not mentioned serve as general purpose registers.

x0 (constant zero): The content of register `x0` is a constant `zero`. Register `x0` is mainly used for conditional operand comparison of branch instructions (e.g. BNE, BEQ).

x1, x5 (returning address): Register `x1` keeps the returning address of jump and link instructions (e.g. JAL, JALR) when a procedure call is invoked. Register `x5` serves as an alternative link register as well.

x2 (stack pointer): Register `x2` serves as a stack pointer, and the value of `x2` decreases as the stack grows. Here, stack and heap together contain 256 words.

x10-17 (saved registers): When a procedure call is invoked, registers `x10-x17` are used to transfer parameters and returning values of the procedure. Contents of registers `x10-x17` are pushed into stack before the procedure begins, and then restored after the procedure is terminated.

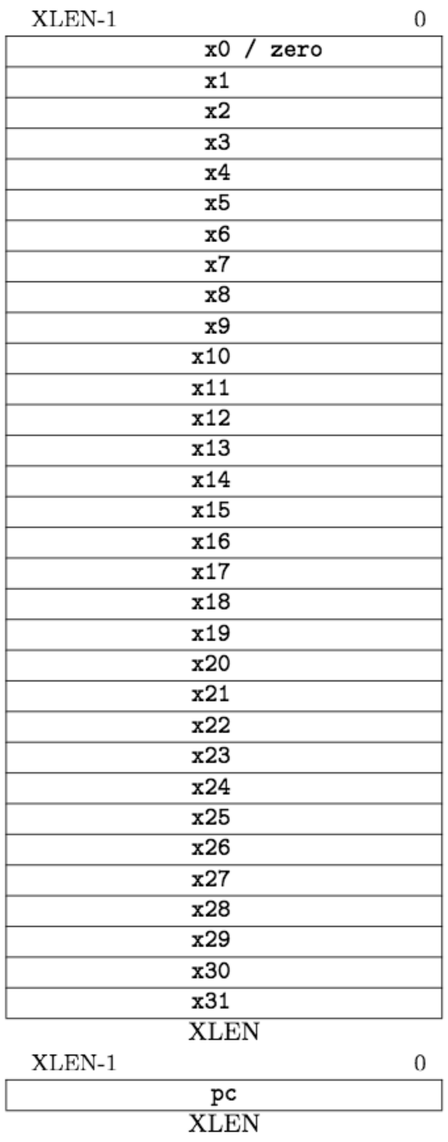

Figure 5–2: RV32I general-purpose user registers [32]

x6-8, x28-31 (temporary variables): Registers x6-8 and x28-31 can be used to hold temporary variables/results of internal computation within a procedure call.

pc (program counter): `pc` is considered as the 33rd register; however, it is transparent to users. The function of `pc` is to hold the address of the instruction currently being executed.

5.6.2 Harvard architecture

A Harvard architecture uses separate memory to store program and data, with each memory having its own data bus for read/write operations. Figure 5–3 illustrates the program and data memories for our RV32I processor. In order to simplify the HLS synthesis and simulation process, program and data memories are both configured as 512-words $\times$ 32-bits. To do so, program memory is instantiated as array `imem[512]` in the HLS `C` specification. Similarly, data memory is instantiated as an array `dmem[512]`. However, data memory is logically divided into two parts: 1) 256-words $\times$ 32-bit static data section; and 2) 256-words $\times$ 32-bit dynamic data section. In such cases, global variables and structures reside in the static data section. Stack and heap hold temporal variables, structures and addresses within a procedure call.

5.6.3 Instruction Formats/Functions

The RV32I processor comprises six instruction formats (R/I/S/B/U/J) as shown by Figure 5–4. We observe that the instruction formats of RV32I are simple and clean. For example, all instruction operation codes (`"opcode"` for short) are of fixed-length (7 bits) and placed at fixed-location `ins[6..0]`. Also, the placements of all source and destination registers (i.e. `rs1`, `rs2`, `rd`) are in fixed-locations across all types of instructions. These are attractive features which allow for fast decoding and efficient execution of instructions

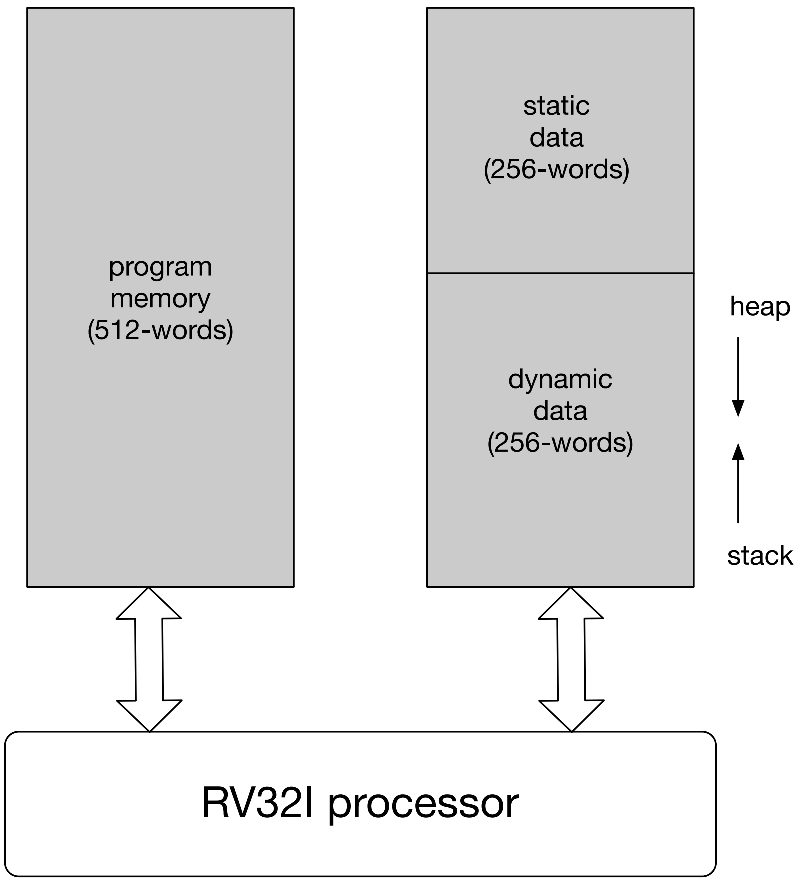

Figure 5–3: Harvard architecture for RV32I processor

for a processor. In the following sections, we elaborate on each of the six instruction types in terms of format and function.

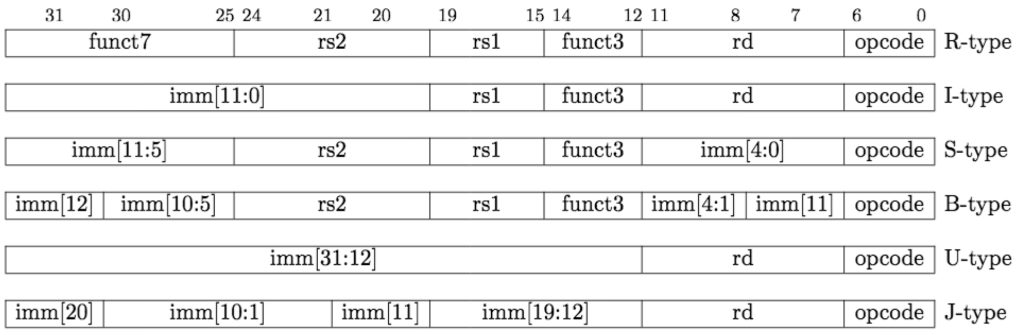
31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 5–4: RV32I instruction formats [32]

R-type instructions

The R-type instruction format is designated for three-register instructions, such as `ADD rd, rs2, rs1`, where `rs1` and `rs2` serve as resource registers and `rd` is the destination register. All R-type instructions have `opcode` "0010011", which has a bitwidth of 7 and is placed at LSBs (least significant bits)[6..0]. Segments `funct3` and `funct7` are used to further differentiate instructions after they are mapped into the R-type instruction category. Across all six types of instructions, `opcode` always resides at the LSBs of a word.

I-type instructions

Similar to R-type, the I-type instruction format is designated for two-register instructions, where `imm` represents a 32-bit immediate number/address which is sign-extended from a 12-bit immediate number (`imm[11:0]`). To achieve fast instruction decoding, the locations of source and destination registers for I-type instructions are the same as those of R-type. Generally, R- and I-type instructions cover all arithmetic and logical operations for the RV32I processor. In addition, there are five kinds of load instructions under the I-type format, which are LB (load a byte), LH (load a half-word), LW (load a word), LBU (load a byte as unsigned), and LHU (load a half-word as unsigned).

In such cases, to address a word in data memory, base addressing `(base + offset)` is adopted. Source register `rs1` holds the base address, and the offset is the 32-bit immediate address. Hence, destination register `rd` is loaded from the word addressed by `(rs1 + sign-extend(imm[11:0]))` in data memory. By doing so, the addressing range is $\pm 2K$ words with regard to the base address stored in register `rs1`. It is worth noting that last two significant bits of an address specify which byte or half-word is loaded from a word. For example, if the last two bits are "00", then the rightmost byte of a word is selected. The difference between LB and LBU is that, the selected byte is sign-extended into a word for LB, but zero-extended into a word for LBU. Furthermore, an interrupt is invoked in case there is an address mis-align for load instructions.

S-type instructions

The S-type instruction format is only designated for store instructions, such as SB (store a byte), SH (store a half-word) and SW (store a word). Similar to load instructions, base addressing is adopted. As a result, the byte/half-word/word stored in source register `rs2` is written to address `(rs1 + sign-extend(imm[11:0]))` in data memory. Similar to the load instructions, the last two bits of the address specify which byte or half-word of a word is written to. Furthermore, the addressing range is also $\pm 2K$ words, and an interrupt is invoked for address mis-align.

B-type instructions

The B-type instruction format is designated for branch instructions, such as BEQ (branch if equal), BNE (branch if not equal), etc. To do so, operands that need to be compared are placed into source registers rs1 and rs2, respectively. If a condition (i.e. rs1 == rs2) is satisfied, then a different instruction sequence is executed by using pc-relative addressing (pc + offset). Similar to base addressing, pc specifies the base address, and the offset is a 32-bit immediate address which is sign-extended from a 13-bit immediate (imm[12:0]), hence, the instruction in program memory imem[pc + sign-extend(imm[12:0])] is executed if condition is true. The range of pc-relative addressing is $\pm 4K$ words, and an interrupt is invoked in case there is an address mis-align for branch instructions.

U and J-type instructions

U and J-type instruction formats are designated for jump and link instructions (e.g. JAL, JALR) when procedure calls are invoked. The JAL instruction uses U-type format and pc-relative addressing (pc + offset), while the JALR instruction uses J-type format and base addressing (rs1 + offset). JAL and JALR make the processor execute a new program sequence beginning at address pc + sign-extend(imm[20:0]) and rs1 + sign-extend(imm[11:0]) in program memory, respectively. The addressing range for JAL ($\pm 1M$ words) is larger than that of JALR ($\pm 4K$ words). For both the JAL and the JALR instructions, if the destination register rd is equal to x1, then the returning address (pc + 4) is kept in register x1 by default. Otherwise, the returning address is simply discarded.

5.7 HLS C implementation

In this section, we present C code snippets of the RV32I multi-cycle processor. Similar to floating-point accelerators, fixed-point data types (i.e. uint32_t, uint16_t, uint8_t) and bit-mask operations are extensively used. In addition, the processor top design, initialization of register file and program/data memory are also presented. We demonstrate several methods to improve processor performance in the end.

Top design. Behaviours of the RV32I multi-cycle processor are described in the `main()` function. A `for-loop` construct is created to simulate that a processor continually fetches and executes an instruction stored in the program memory (`imem`) specified by the `pc`. We adopt the Harvard architecture to separate program and data memories (`dmem`). In our system, the program and data memory each have 512 words due to our testing programs do not need large memory space. Each iteration of the `for-loop` construct corresponds to the execution of one single instruction. This simulates how a multi-cycle processor executes instructions. One instruction execution starts upon completion of previous/current instructions. Hence, there is no interleaved execution of instructions like pipeline processor. To elaborate on the multi-cycle processor, after instruction fetch, an instruction is latched into the instruction register (IR), and then transferred into a function called `ins_dec_exe()` for further processing (i.e. decode/execute). As soon as the instruction is completed, `pc` is either incremented by 4 to address the next sequential instruction, or loaded with an address that points to a completely new program sequence. It is necessary for the FPGA synthesizer to know for how many iterations the `for-loop` construct will be executed before running the program. Hence, we first set the loop index as 512, a constant number. With this index, all aforementioned testing programs can execute correctly.

```
// top design of RV32I
int main (void){
  ...
   // instructions fetch, decode, execution
   loop_riscv:
   for ( i = 0; i < 512; i++){
      // instruction fetch, program memory contains 512 words
      ir = imem[pc >> 2];
      // instruction decode and execute
      ins_dec_exe(ir);
   }
 ...
}
```

Memories and register file initialization. Program/data memories and the register file are instantiated as one-dimensional arrays of various sizes. In the following code snippet, program memory is implemented as array `imem[512]`, which contains 512 words. The register file is implemented as array `reg32[32]`, which contains 32 general-purpose registers. In program memory, the first word (0x002081b3) represents instruction ADD x3, x2, x1. It adds content stored in two source registers (i.e. x1, x2), and puts the sum (x1 + x2) into a destination register (i.e. x3). After the ADD instruction is completed, the content of register `x3` becomes 0x00000001. The ”volatile” keyword `volatile` is added to arrays to prevent them from being wiped out by the optimization of LLVM `HLS-C` into the hardware circuit.

```
    // define program memory
    volatile uint32_t imem[512] = {
        0x002081b3, // ADD x3, x2, x1;
        0x40520333,
        0x008394b3,
        0x00b55633,
        0x40e6d7b3,
            ...
    }
    // define register file
    volatile uint32_t reg32[32] = {
        0x00000000, // x0 = 0
    0xffffffff, // x1 = -1
    0x00000002, // x2 = 2
    0x00000000, // x3 = 0
    0x00000000, // x4 = 0
      ...
}
```

Instruction decode and execute. The behaviours of instruction decode and execute are described in function `ins_decode_execute()`. To take ADD instruction as an example, three segments are extracted for decode, which are operation code `op_r`, funct3 `op1_0` and funct7 `op2_0`. A global structure variable, `sig1`, which contains instruction enable signals, is created. If the ADD instruction's enable signal (i.e. `sig1.add`) is set to high, add function `uint32_t add (uint32_t, uint32_t)` is invoked. In this case, all three segments need to comply with the encoding of the ADD instruction to set `sig1.add` to high. Since the ADD instruction requires three registers, two source registers (i.e. `reg32[rs1]`, `reg32[rs2]`) and one destination register (i.e. `reg32[rd]`) are specified. Hence, indexes of registers (i.e. `rs1, rs2, rd`) are extracted before invoking the function. When `sig1.add` is high, the contents of the source registers are passed to the `add` function as parameters. The sum of the source registers is written back to the destination register when the `add` function is completed.

```
// define instruction control signals
struct dec_sig {
 ...
 // if add is valid, then it executes ADD instruction
 uint32_t add;
 ...
} sig1;
```

```
// instruction decode and execute
void ins_decode_execute(uint32_t ins){
 // instruction decode
 uint32_t op_r, op1_0, op2_0;
 ...
 // operation code
 op_r = (ins & 0x0000007f) ^ 0x00000033;
 // funct3
 op1_0 = (ins & 0x00007000) ^ 0x00000000;
 // funct7
 op2_0 = (ins & 0xfe000000) ^ 0x00000000;
 ...
 // set enable signal of ADD instruction
 sig1.add = !op_r & !op1_0 & !op2_0;
 ...
 // extract content of two source and one destination registers
 uint32_t rs1, rs2, rd;
 ...
 // 1st source register
 rs1 = (ins >> 15) & 0x0000001f;
 // 2nd source register
 rs2 = (ins >> 20) & 0x0000001f;
 // destination register
 rd = (ins >> 7) & 0x0000001f;
 ...
 // call subroutine to execute ADD instruction
 if (sig1.add){
    reg32[rd] = add(reg32[rs1], reg32[rs2]);
 }
}
```

Remove redundant reads/writes to register files. Due to the constraint on the maximum number of reads/writes allowed for memory blocks on FPGAs, efforts have been made to reduce read/write operations to register files. The following code snippet shows before and after redundant read/write operations have been removed. In the original code, two functions are shown, which represent ADD and SUB instructions, respectively. In this case, two reads and one write are necessary for each instruction. Thus, six memory accesses (i.e. four reads and two writes) are made in total during hardware synthesis.

```
// original code
if (sig1.add){
   reg32[rd] = add(reg32[rs1], reg32[rs2]); // ADD
}
...
if (sig1.sub){
   reg32[rd] = sub(reg32[rs1], reg32[rs2]); // SUB
}
...
```

In optimized code, we observe that the values of source registers are loaded to variables (i.e. rs1c, rs2c) before the function starts, and then passed as parameters to the corresponding function. Likewise, the returning value is assigned to a variable (i.e. rdc) within the function, and the register file is updated after. By doing so, all three-register instructions only need to perform two reads and one write to the register file. Without using this mechanism, each three-register instruction requires two reads and one write.

```
// optimized code
rs1c = reg32[rs1]; // read 1st source register
rs2c = reg32[rs2]; // read 2nd source register
if (sig1.add){
   rdc = add(rs1c, rs2c);
}
...
if (sig1.sub){
    rdc = sub(rs1c, rs2c);
}
...
reg32[rd] = rdc; // write back to the destination register
...
```

Remove redundant reads/writes to data memory. Similar to the register file updates, the code snippet is modified to reduce redundant read-/write operations to data memory. In the following code snippet of function `ins_decode_execute()`, four subroutines represent instructions LB (load a byte), LH (load a half-word), SB (store a byte) and SH (store a half-word). To reduce read operations to data memory, a word is loaded from data memory to variable `mem_c` before calling a subroutine, and then passed as a parameter to the corresponding subroutine. By doing so, all memory read instructions only require one memory access. In the body of subroutine `lb()`, the rightmost byte of a word is sign-extended via bit-mask/shifting. Likewise, to reduce write operations to data memory, a byte/half-word/word after a certain extension is treated as a return value, and then written back to data memory outside a subroutine. By doing so, all memory write instructions only require one memory access.

```
union Data32 {
    uint32_t ui; // unsigned
    int32_t i; // signed
}

void ins_decode_execute(uint32_t ins) {
    ...
    // retrieve a word from data memory
    mem_c = dmem[addr];
    ...
```

```
   // call subroutine to load a byte
   if ( sig1.lb ) {
      tmp1 = lb(mem_c, addr);
   }
   // call subroutine to load a half-word
   if ( sig1.lh ) {
      tmp1 = lh(mem_c, addr);
   }
   ...
   // call subroutine to store a byte
   if ( sig1.sb ) {
      tmp2 = sb(addr, rs2c);
   }
   // call subroutine to store a half-word
   if ( sig1.sh ) {
      tmp2 = sh(addr, rs2c);
   }
   ...
   dmem[addr >> 2] = tmp2; // write back to data memory
}

// load byte instruction
uint32_t lb(uint32_t mem_c, uint32_t addr) {
   union Data32 temp;
   temp.ui = mem_c;

   // if load rightmost byte of the word
   if ( ( addr & 0x00000003 ) == 0x00000000 )
   {
      temp.ui = (temp.ui & 0x000000ff) << 24;
      temp.i = temp.1 >> 24;
   }
   ...
```

```
}
```

5.8 Testbenches for RV32I Processor

In this section, we first manually create seven custom testing programs in RV32I assembly code to verify: 1) if the soft-processor implementation adheres to RISC-V specifications, and 2) if the soft-processor generates correct results for those testing programs. After passing the seven testing programs, we then use GCC toolflow to generate four additional custom programs from C software to RV32I assembly code for further testing.

5.8.1 Manually Created Testing Programs

RV32I instructions can be summarized into four types, as shown by Table 5–2. In this table, R-type refers to all three-register arithmetic/logical instructions, such as `ADD x3, x2, x1`. I-type refers to all two-register arithmetic/logical instructions, such as `ADDI x2, x1, imm0`. Furthermore, LS-type represents all load/store instructions which perform read/write operations to data memory, such as `LB x4, imm1(x3)` or `SB x6, imm2(x5)`. Finally, B-type represents all branch, jump and link instructions, such as `BEQ x6, x7, Label`. We manually create one custom testing program in RV32I assembly code for each of R, I, LS-type instructions. There are four testing programs created for B-type instructions because they are more complicated in terms of program control. Each testing program uses instructions that fall into that type to verify the functional correctness of each instruction. To simplify, we only show the nested procedure call program (in RV32I assembly code) for testing the B-type instructions in this section. For interested readers, please refer to the Appendix Chapter for the complete seven testing program descriptions.

Table 5–2: Categories of RV32I ISA.

Category	Description	Instructions
R-type	3-register arithmetic/logical instructions	ADD, SUB, SLL, SLT SLTU, XOR, SRL, SRA OR, AND, NOP
I-type	2-register arithmetic/logical instructions	ADDI, SLLI, SLTI, SLTUI SLTUI, XORI, SRLI SRAI, ORI, ANDI
LS-type	load/store instructions	LB, LH, LW, LBU LHU, SB, SH, SW
B-type	branch/jump and link instructions	BEQ, BNE, BLT, BGE BLTU, BGEU, JAL, JALR LUI, AUIPC

Nested procedure call program for B-type instructions testing

The following testing program realizes a recursive algorithm, which is implemented as a nested procedure call in high-level programming language. The algorithm performs recursive addition on integer `n`, and produces a sum which is equal to `n + (n-1) + (n-2) + ...  + 2 + 1`. In this case, subroutine `iter_add` starts from address imem[10] in program memory, and parameter `n (n = 5)` is loaded into register `x10`. In addition, register `x1` is used to keep the returning addresses of a procedure call, and register `x2` serves as a stack pointer. In the body of subroutine `iter_add()`, values stored in registers `x1` and `x10` are recursively pushed back into stack until variable `n` is less than 1 (n < 1). By observing the stack in data memory, the first returning address is stored at dmem[510] as `0x00000068`, which refers to the returning address of subroutine `iter_add()`. Later on, the returning address of instruction `JAL r1, iter_add`, are pushed into stack one by one. After five iterations, stack growing terminates at `dmem[501]`, as variable `n` is less than 1 (n < 1). In this case, all the values of variable `n` and returning addresses are popped to registers recursively, and, finally, a sum which is equal to `n + (n-1) + (n-2) + ...  + 2 + 1` is produced at register `x10`.

```
/*
int iter_add (int n) {
   if(n < 1) return (1);
   else return(n + iter_add(n-1));
}
*/
// program memory
imem[10]: 0xff810113 // iter_add: ADDI r2, r2, 0xff8
imem[11]: 0x00112223 // SW r1, 0x4(r2);
imem[12]: 0x00a12023 // SW r10, 0x4(r2);
imem[13]: 0xfff50293 // ADDI r5, r10, 0xfff;
imem[14]: 0x0002d863 // BGE r5, r0, L1;
imem[15]: 0x00100513 // ADDI r10, r10, 0x001;
imem[16]: 0x00810113 // ADDI r2, r2, 0x008;
imem[17]: 0x00008067 // JALR r0, 0x000(r1);
imem[18]: 0xfff50513 // L1: ADDI r10, r10, 0xfff;
imem[19]: 0xfddff0ef // JAL r1, iter_add;
imem[20]: 0x00050313 // ADDI r6, r10, 0x000;
imem[21]: 0x00012503 // LW r10, 0x0(r2);
imem[22]: 0x00412083 // LW r1, 0x4(r2);
imem[23]: 0x00810113 // ADDI r2, r2, 0x008;
imem[24]: 0x00a30533 // ADD r10, r10, r6;
imem[25]: 0x00008067 // JALR r0, 0x0(r1);
// data memory (stack)
dmem[501]: 0x00000001 // n-4
dmem[502]: 0x00000050 // returning address imem[20]
dmem[503]: 0x00000002 // n-3
dmem[504]: 0x00000050 // returning address imem[20]
dmem[505]: 0x00000003 // n-2
dmem[506]: 0x00000050 // returning address imem[20]
dmem[507]: 0x00000004 // n-1
dmem[508]: 0x00000050 // returning address imem[20]
dmem[509]: 0x00000005 // n
```

```
dmem[510]: 0x00000068 // returning address imem[26]
```

5.8.2 Testing Programs Generated from GCC toolflow

In addition to the seven manually created RV32I assembly testing programs in the previous section, we also use RV32I *GCC* toolflow to compile testing programs from `C` software to RV32I machine code. Those testing programs include bubble sort applied to an in-memory array, a Fibonacci-sequence generator, factorial (using software multiply), and recursive vector addition. To simplify, we only show the recursive vector addition program (in C-software) in this section.

```c
volatile int list[] = {-1, 5, 8, -5, -8, 10, 12, 23,
    -45, -5, -6, -7, 45, 78, 0};

int recurAdd(int n) {
  int q = list[n];

  if (q == 0)
    return 0;
  return q + recurAdd(n+1);
}
```

5.9 Experimental Study of RV32I Processor

In this section, we show the performance of our RV32I multi-cycle processors. The HLS tool interprets the C specification to RTL hardware whose maximum clock frequency (FMax) can be specified by the user. In our case, we tune the clock period to six variants – 20, 15, 10, 5, 3 and 2 ns – to produce a variety of processor variants. As the clock period decreases, FMax increases as expected. In the following subsections, we demonstrate the performance of the processors in terms of maximum frequency (FMax), ALMs, number of cycles for instruction execution (Cycle No.), cycles per instruction (CPI), instruction per cycle (IPC) and clock period time (CP). Here, an ALM contains a six-input lookup-table and two flip-flops. In addition, the proposed processors are targeted to the Intel/Altera Cyclone V 28 nm FPGA. The LegUp HLS version 7.4, Quartus version 18.1 and ModelSim are used to interpret, synthesize and verify the functional and timing correctness of the processors.

5.9.1 RV32I Multi-Cycle Processor

We created various multi-cycle processors by specifying different clock cycle periods: 20, 15, 10, 5, 3 and 2 ns. In experiments, we find that an HLS-interpreted processor sometimes may have small variance in maximum frequency and circuit area when loaded with different testing programs. Hence, to clarify, our multi-cycle processors are all loaded with the recursive vector addition testing program shown in section 5.8.2 when measuring the performance. In Table 5–3, the smallest processor operates at 124.1 MHz, and consumes 795 ALMs. The total number of cycles required to execute the testing program is 7,062. Hence, each instruction needs 27.48 cycles, and 0.036 instructions are executed per cycle. Next, the fastest processor operates at 232.18 MHz, and consumes 1,018 ALMs. The total number of cycles required to execute the testing program is 11,824. Hence, each instruction needs 46.01 cycles, and 0.021 instructions are executed per cycle. We observe that the FMax is dramatically affected by decreasing the clock period time. As expected, the circuit area increases as the FMax varies from 124.1 to 232.18 MHz. The HLS tool automatically chops long combinational logic paths into shorter ones and inserts flip-flops as the FMax increases. Those additional flip-flops result in an increase to the circuit area. We observe that the multi-cycle processor generally needs dozens of clock cycles for each instruction. In

Table 5–3: Our RV32I multi-cycle processor

FMax (MHz)	**ALMs**	**Cycle No.**	**CPI**	**IPC**	**CP** (ns)
124.1	795	7,062	27.48	0.036	20
154.1	832	7,591	29.54	0.034	10
220.31	967	10,010	38.95	0.026	5
232.18	1,018	11,824	46.01	0.021	3

this case, one instruction execution starts upon completion of previous/current instructions. Hence, there is only one instruction executing at any one time in the processor, causing a lower issue rate for instructions. The FMax is set by propagation delay of the critical path in the processor datapath.

5.10 Performance Comparison

We have chosen two open-source RV32I RTL implementations for performance comparison: FWRISCV [18] and PicoRV32 [30]. Both FWRISCV and PicoRV32 are RV32I multi-cycle cores. Table 5–4 demonstrates our four multi-cycle cores and the two open source cores. Also, Figure 5–5 illustrates the area usage and FMax for the six processors. Here, CP_3 refers to the multi-cycle core whose clock period is 3 ns.

In Figure 5–5, the left axis represents the number of ALMs that processors require, while the right axis represents the FMax at which the processors operate. For the area usage, the four blue bars on the left are our proposed cores. The two blue bars on the right are the open source cores. We observe that the FWRISCV core uses the fewest ALMs, while the PicoRV32 takes approximately 1000 ALMs. The FMax of our proposed cores ranges from 124.1 MHz to 232.18 MHz. On the other hand, FWRISCV operates at 89.71 MHz, PicoRV32 operates at 183.35 MHz. As a result, we can see that our proposed cores are competitive with the open source RTL cores.

Table 5–4: Performance comparison between HLS-interpreted RV32I cores and RTL implementations of RV32I

	FMax (MHz)	**ALMs**	**IPC**
CP_20	124.1	795	0.036
CP_10	154.1	832	0.034
CP_5	220.31	967	0.026
CP_3	232.18	1,018	0.021
FWRISCV	89.71	616	unknown
PicoRV32	183.35	952	0.25-0.3

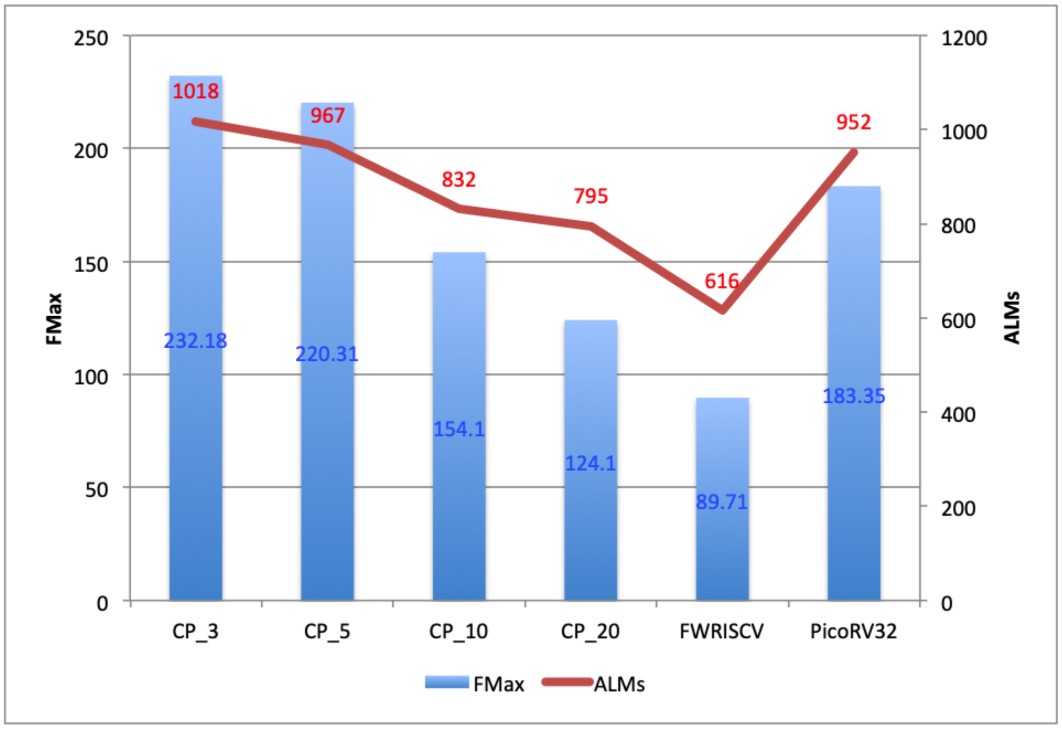

Figure 5–5: Area and FMax comparisons between HLS-generated RV32I cores and RTL implementations of RV32I

5.10.1 Instructions per Cycle (IPC)

According to [30], PicoRV32 requires 3 cycles for ALU instructions, 3 cycles for direct jumps, 4-14 cycles for shift operations, and 5-6 cycles for the remaining instructions. That is, at a minimum PicoRV32 requires 3 cycles/instruction. We expect that with a tightly-coupled memory, it would offer an IPC of between 0.25-0.3. Our multi-cycle processors, while offering high $Fmax$, have long latency, with different instructions requiring different numbers of cycles. We observed these take several dozen clock cycles, on average, to execute an instruction, making them inferior from the IPC perspective vs. the PicoRV32 processor.

5.11 Summary

In this chapter, we implement a RV32I multi-cycle processor by using C-language. The processor consists of 39 user instructions and realizes the minimum function of a 32-bit RISC-V integer processor. The `C` specification is synthesized into hardware circuits targeted to the low-cost 28 nm Intel/Altera Cyclone V FPGA using the LegUp HLS, Quartus and ModelSim. We verify the correctness of the processor with manually created testing programs in assembly code, and testing programs compiled from GCC toolflow (e.g. bubble sort, Fibonacci-sequence generator, factorial). From the same `C` specification, we obtain the cores with various performance/area trade-offs. The area/FMax of the cores is studied and compared with two open source RTL-implemented cores. Results show that our proposed cores are competitive with the RTL cores in terms of FMax and area, but are inferior from the IPC perspective.

Chapter 6
Conclusion and Future Work

Many scientific applications require maximum-throughput/ultra-low latency, high-precision and minimum energy consumption for floating-point mathematical operations (e.g. exponential, logarithm, trigonometry) [55]. On the other hand, some machine learning applications, such as deep learning, mainly concentrate on high-throughput/low latency and low energy consumption. In such cases, precision is traded for better performance per watt, by using customized floating-point precision (e.g. 16-bit half-precision). Hence, both high-performance and reduced-precision versions have been considered for elementary transcendental function accelerators in this dissertation.

We implement single-precision floating-point reciprocal and square root accelerators based on FPGA. The algorithms of the proposed accelerators are designed by C language, which include iterative (e.g. trial-subtraction, Newton's) and non-iterative (LUT-based) methods. Since the LUT-based algorithm is universal, it could be applied to implement an entire library of single-precision elementary functions into high-performance hardware accelerators. Based on exhaustive testing, our accelerators are able to produce results with 1 ULP maximum error, compared with single-precision cores from GNU math.h library. The LegUp HLS, Quartus and Modelsim are applied to synthesize the accelerators specified using C language into hardware circuit targeting Intel/Altera 45 nm FPGA. In evaluation, we compare ours and the state-of-the-art implementations. In particular, we treat the Intel/Altera IP cores as the "golden" baseline for performance comparison (i.e. area/speed). Results show that Intel/Altera IP cores win slightly on throughput, and our cores win considerably in circuit area (i.e. LUT size, number of DSPs) on Cyclone V 45 nm FPGA.

In addition to the elementary functions implementation, we propose a 32-bit RISC-V multi-cycle processor implementation based on FPGA. The processor is a full realization of the 32-bit integer base instruction set (RV32I) and consists of 39 user instructions. Custom testing programs are developed to verify the correctness of instructions execution and if the processor adheres to the RISC-V specification. The LegUp HLS, Quartus and Modelsim are applied to synthesize the C specification of the processor into hardware targeting Intel/Altera 28 nm FPGA. In evaluation, we compare our processor with two open source RISC-V implementations. Results show that our processor has equivalent performance in terms of area and speed, but the open source processor has higher IPC.

There are four possible directions for my future research:

1. **Evaluate energy efficiency.** We would like to evaluate energy efficiency for the proposed elementary function accelerators and RISC-V soft processor. In this case, performance will be given in throughput per watt (MHz/watt). Additional energy savings will be measured for reduced-precision implementations as well.
2. **An entire library of elementary functions implementation.** Since the LUT-based design methodology is generic, it could be applied to implement an entire library of single-precision floating-point elementary functions into high-performance hardware accelerators in an FPGA or ASIC.
3. **Pipeline RISC-V processor implementation.** We would like to implement a 32-bit RISC-V pipeline processor to further improve the parallelism of program execution. To do so, we will need to design

an efficient pipeline architecture, and address both data and control hazards.

4. **Future processor/accelerators system on FPGA.** We would like to integrate the proposed reciprocal and square root accelerators into the RISC-V processor by extending the ISA. The RISC-V ISA has extensions for single, double and quadruple-precision instructions. By doing so, the future processor/accelerators system is possible to bring significant performance benefits to a broad range of scientific applications.

Appendix

In this section, we elaborate on the seven manually created testing programs (in RV32I assembly code) for the proposed soft-processor. Contents of program memory, data memory and register files before and after a testing program is executed are listed as below.

6.1 Testing Program for R-type Instructions:

The following code snippet presents a testing program for R-type instructions. In this case, machine codes in hexadecimal representation and RV32I assembly are presented. The testing program starts from the first word in the program memory (`imem[0]`), and is executed in sequential order. Since all operands of R-type instructions are manipulated via registers, operands are manually loaded into the registers (i.e. `x1, x2, x4, x5`) to reduce traffic between the register file and data memory. In the code snippet, all R-type instructions are presented. In addition, the NOP instruction stands for "no operation". It is a pseudo-instruction that is translated to `ADD x0, x0, 0` after compilation.

```
// program memory
imem[0]: 0x002081b3 // ADD x3, x2, x1;
imem[1]: 0x40520333 // SUB x6, x5, x4;
imem[2]: 0x008394b3 // SLL x9, x8, x7;
imem[3]: 0x00b55633 // SRL x12, x11, x10;
imem[4]: 0x40e6d7b3 // SRA x15, x14, x13;
imem[5]: 0x01182933 // SLT x18, x17, x16;
imem[6]: 0x0129bab3 // SLTU x21, x20, x19;
imem[7]: 0x017b4c33 // XOR x24, x23, x22;
imem[8]: 0x01acedb3 // OR x27, x26, x25;
imem[9]: 0x01de7f33 // AND x30, x29, x28;
```

```
imem[10]: 0x00000013 // NOP (ADD, x0, x0, 0);
```

Contents of program memory, data memory and register files before and after the R-type testing program is executed are listed as below.

- **program memory.**

  0x002081b3, //0

  0x40520333, //1

  0x008394b3, //2

  0x00b55633, //3

  0x40e6d7b3, //4

  0x01182933, //5

  0x0129bab3, //6

  0x017b4c33, //7

  0x01acedb3, //8

  0x01de7f33, //9

- **data memory (before).** all zeros

- **register file (before).**

  0x00000000, //0

  0xffffffff, //1

  0x00000002, //2

  0x00000000, //3

  0x80000000, //4

  0x00000001, //5

  0x00000000, //6

  0x0000ffff, //7

  0x0000000f, //8

  0x00000000, //9

  0xffff0000, //10

0x0000000f, //11

0x00000000, //12

0xffff0000, //13

0x0000000f, //14

0x00000000, //15

0x80000001, //16

0x00000001, //17

0x00000000, //18

0x80000001, //19

0x00000001, //20

0x00000000, //21

0x55555555, //22

0xffffffff, //23

0x00000000, //24

0x55555555, //25

0xaaaaaaaa, //26

0x00000000, //27

0x55555555, //28

0xaaaaaaaa, //29

0x00000000, //30

0x00000000 //31

- **data memory (after).** all zeros
- **register file (after).**

0x00000000, //0

0xffffffff, //1

0x00000002, //2

0x00000001, //3

```
0x80000000, //4
0x00000001, //5
0x7fffffff, //6
0x0000ffff, //7
0x0000000f, //8
0x7fff8000, //9
0xffff0000, //10
0x0000000f, //11
0x0001fffe, //12
0xffff0000, //13
0x0000000f, //14
0xfffffffe, //15
0x80000001, //16
0x00000001, //17
0x00000001, //18
0x80000001, //19
0x00000001, //20
0x00000000, //21
0x55555555, //22
0xffffffff, //23
0xaaaaaaaa, //24
0x55555555, //25
0xaaaaaaaa, //26
0xffffffff, //27
0x55555555, //28
0xaaaaaaaa, //29
0x00000000, //30
```

0x00000000 //31

6.2 Testing Program for I-type Instructions:

Unlike R-type, one operand of I-type instructions comes from a 32-bit sign-extended immediate number rather than a source register. Hence, I-type instructions are considered to be more efficient because there is no need to load both operands from data memory to a register file. Similar to R-type, the testing program for I-type instructions starts from the first word of program memory (`imem[0]`), and is executed in sequential order. All I-type instructions are presented in the following code snippet, and operands are loaded into the register file manually.

```
// program memory
imem[0]: 0x7ff08113 // ADDI x2, x1, 0x7ff;
imem[1]: 0x80018213 // ADDI x4, x3, 0x800;
imem[2]: 0x5552c313 // XORI x6, x5, 0x555;
imem[3]: 0xaaa3c413 // XORI x8, x7, 0xaaa;
imem[4]: 0x5554e513 // ORI x10, x9, 0x555;
imem[5]: 0xaaa5e613 // ORI x12, x11, 0xaaa;
imem[6]: 0x5556f713 // ANDI x14, x13, 0x555;
imem[7]: 0xaaa7f813 // ANDI x16, x15, 0xaaa;
imem[8]: 0x8008a913 // SLTI x18, x17, 0x800;
imem[9]: 0x8009ba13 // SLTIU x20, x19, 0x800;
imem[10]: 0x00fa9b13 // SLLI x22, x21, 0x00f;
imem[11]: 0x00fbdc13 // SRLI x24, x23, 0x00f;
imem[12]: 0x40fcdd13 // SRAI x26, x25, 0x00f;
imem[13]: 0x40fdde13 // SRAI x28, x27, 0x00f;
imem[14]: 0x00000013 // NOP (ADD, x0, x0, 0);
```

Contents of program memory, data memory and register files before and after the I-type testing program is executed are listed as below.

- **program memory.**

  0x7ff08113, //0

  0x80018213, //1

  0x5552c313, //2

  0xaaa3c413, //3

  0x5554e513, //4

  0xaaa5e613, //5

  0x5556f713, //6

  0xaaa7f813, //7

  0x8008a913, //8

  0x8009ba13, //9

  0x00fa9b13, //10

  0x00fbdc13, //11

  0x40fcdd13, //12

  0x40fdde13, //13

- **data memory (before).** all zeros

- **register file (before).**

  0x00000000, //0

  0x00000001, //1

  0x00000000, //2

  0x00000800, //3

  0x00000000, //4

  0x00000aaa, //5

  0x00000000, //6

  0x00000555, //7

  0x00000000, //8

  0x00000555, //9

0x00000000, //10

0x00000555, //11

0x00000000, //12

0x00000000, //13

0x00000000, //14

0x00000fff, //15

0x00000000, //16

0x00000001, //17

0x00000000, //18

0x00000001, //19

0x00000000, //20

0xffffffff, //21

0x00000000, //22

0xffffffff, //23

0x00000000, //24

0x80000000, //25

0x00000000, //26

0x7fffffff, //27

0x00000000, //28

0x00000000, //29

0x00000000, //30

0x00000000 //31

- **data memory (after).** all zeros
- **register file (after).**

0x00000000, //0

0x00000001, //1

0x00000800, //2

```
0x00000800, //3
0x00000000, //4
0x00000aaa, //5
0x00000fff, //6
0x00000555, //7
0xffffffff, //8
0x00000555, //9
0x00000555, //10
0x00000555, //11
0xffffffff, //12
0x00000000, //13
0x00000000, //14
0x00000fff, //15
0x00000aaa, //16
0x00000001, //17
0x00000000, //18
0x00000001, //19
0x00000001, //20
0xffffffff, //21
0xffff8000, //22
0xffffffff, //23
0x0001ffff, //24
0x80000000, //25
0xffff0000, //26
0x7fffffff, //27
0x0000ffff, //28
0x00000000, //29
```

```
0x00000000, //30
0x00000000 //31
```

6.3 Testing Program for LS-type Instructions:

LS-type instructions handle read/write operations between data memory and the register file. To load data from data memory to the register file, there are LB (load a byte), LBU (load an unsigned byte), LH (load a half-word), LHU (load an unsigned half-word) and LW (load a word) instructions. Likewise, to store data from the register file into data memory, there are SB (store a byte), SH (store a half-word) and, SW (store a word) instructions. In such cases, base addressing `(base + offset)` is adopted to address data memory. In doing so, the base address is stored in a source register, and the offset comes from a 32-bit immediate address. Address alignment is enforced by RISC-V ISA. For example, the lowest bit of an address must be bit-0 if addressing a half-word, and the lower two bits of an address must be bit-00 if addressing a word. Otherwise, an interrupt is invoked to handle address misaligned cases. All LS-type instructions are presented in the following code snippet.

```
// program memory
imem[0]: 0x00108103 // LB x2, 0x001(r1);
imem[1]: 0x3ff18203 // LB x4, 0x3ff(r3);
imem[2]: 0x00229303 // LH x6, 0x002(r5);
imem[3]: 0x3fe39403 // LH x8, 0x3fe(r7);
imem[4]: 0x0004a503 // LW x10, 0x000(r9);
imem[5]: 0x0045a603 // LW x12, 0x004(r11);
imem[6]: 0x0046c703 // LBU x14, 0x004(r13);
imem[7]: 0x00c7c803 // LBU x16, 0x00c(r15);
imem[8]: 0x0048d903 // LHU x18, 0x004(r17);
imem[9]: 0x01c9da03 // LHU x20, 0x01c(r19);
```

```
imem[10]: 0x015b0023 // SB x21, 0x000(r22);
imem[11]: 0x017b00a3 // SB x23, 0x001(r22);
imem[12]: 0x018b0123 // SB x24, 0x002(r22);
imem[13]: 0x019b01a3 // SB x25, 0x003(r22);
imem[14]: 0x01ad9023 // SH x26, 0x000(r22);
imem[15]: 0x01ce9123 // SH x28, 0x002(r22);
imem[16]: 0x01efa223 // SW x30, 0x004(r22);
imem[17]: 0x00000013 // NOP (ADD, x0, x0, 0);
```

Contents of program memory, data memory and register files before and after the LS-type testing program is executed are listed as below.

- **program memory.**

  0x00108103, //0

  0x3ff18203, //1

  0x00229303, //2

  0x3fe39403, //3

  0x0004a503, //4

  0x0045a603, //5

  0x0046c703, //6

  0x00c7c803, //7

  0x0048d903, //8

  0x01c9da03, //9

  0x015b0023, //10

  0x017b00a3, //11

  0x018b0123, //12

  0x019b01a3, //13

  0x01ad9023, //14

  0x01ce9123, //15

  0x01efa223, //16

- **data memory (before).**

  0x13579bdf, // 0

  0xfdb97531, // 1

  0x2eca8642, // 255

  0x2468ace2, // 256

- **register file (before).**

  0x00000000, // 0

  0x00000000, // 1

  0x00000000, // 2

  0x00000000, // 3

  0x00000000, // 4

  0x00000000, // 5

  0x00000000, // 6

  0x00000000, // 7

  0x00000000, // 8

  0x00000000, // 9

  0x00000000, // 10

  0x000003f8, // 11

  0x00000000, // 12

  0x00000000, // 13

  0x00000000, // 14

  0x000003f0, // 15

  0x00000000, // 16

  0x00000000, // 17

  0x00000000, // 18

  0x000003e0, // 19

  0x00000000, // 20

0x12345678, // 21

0x00000600, // 22

0xfedcba98, // 23

0xf0e1d2c3, // 24

0xb4a59687, // 25

0x87ab1e4b, // 26

0x00000700, // 27

0x8b74aeb1, // 28

0x00000780, // 29

0xb847ea1b, // 30

0x000007c0 // 31

- **data memory (after).**

0x13579bdf, // 0

0xfdb97531, // 1

0x2eca8642, // 255

0x2468ace2, // 256

0xb4e1ba78, // 384

0x00001e4b, // 448

0x8b740000, // 480

0xb847ea1b, // 497

- **register file (after).**

0x00000000, // 0

0x00000000, // 1

0xffffff9b, // 2

0x00000000, // 3

0x0000002e, // 4

0x00000000, // 5

```
0x00001357, // 6
0x00000000, // 7
0x00002eca, // 8
0x00000000, // 9
0x13579bdf, // 10
0x000003f8, // 11
0x2eca8642, // 12
0x00000000, // 13
0x00000031, // 14
0x000003f0, // 15
0x00000042, // 16
0x00000000, // 17
0x00007531, // 18
0x000003e0, // 19
0x00008642, // 20
0x12345678, // 21
0x00000600, // 22
0xfedcba98, // 23
0xf0e1d2c3, // 24
0xb4a59687, // 25
0x87ab1e4b, // 26
0x00000700, // 27
0x8b74aeb1, // 28
0x00000780, // 29
0xb847ea1b, // 30
0x000007c0 // 31
```

6.4 Testing Program for B-type Instructions:

The B-type format comprises two kinds of instructions: 1) branch, and 2) jump link (return). These instructions change the order in which programs are executed. In high-level programming language, an `if` or `loop` statement is usually compiled into an assembly containing branch instructions. However, the procedure call construct is normally compiled into an assembly containing jump link instructions. One difference between branch and jump link (return) instructions is that, any branch instruction jump to a new address depends on the result of a comparison between a pair of operands. However, jump link (return) instructions jump directly to a new address without any conditions. The following code snippets show four testing programs for B-type instructions, each of which implements a program control construct in high-level programming language.

6.4.1 If-Then Program for Testing B-type Instructions

The following testing program realizes the function of an `if` statement in high-level programming language. In this case, variables `i`, `j`, `a`, `b` and `c` are manually loaded into registers `r22`, `r23`, `r19`, `r20` and `r21`. All conditional branch instructions adopt PC-relative addressing (`PC + offset`), and the addressing range is $\pm 4K$ words.

```
/*
if (i == j)
   c = a + b;
else
   c = a - b;
*/
// program memory
imem[0]: 0x017b1863 // BNE x22, x23, Else;
imem[1]: 0x014a89b3 // ADD x21, x20, x19;
imem[2]: 0x00000663 // BEQ x0, x0, Exit;
```

```
imem[3]: 0x415a09b3 // Else: SUB x21, x20, x19;
imem[4]: 0x00000013 // Exit: NOP;
```

Contents of program memory, data memory and register files before and after the if-then statement testing program is executed are listed as below.

- **program memory.**

  0x017b1863, //0

  0x014a89b3, //1

  0x00000663, //2

  0x415a09b3, //3

- **data memory (before).** all zeros
- **register file (before).**

  0x00000000, //0

  0x00000000, //1

  0x00000000, //2

  0x00000000, //3

  0x00000000, //4

  0x00000000, //5

  0x00000000, //6

  0x00000000, //7

  0x00000000, //8

  0x00000000, //9

  0x00000000, //10

  0x00000000, //11

  0x00000000, //12

  0x00000000, //13

  0x00000000, //14

  0x00000000, //15

0x00000000, //16

0x00000000, //17

0x00000000, //18

0xffffffff, //19

0x7fffffff, //20

0x00000001, //21

0x0000000f, //22

0x0000000f, //23

0x00000000, //24

0x00000000, //25

0x00000000, //26

0x00000000, //27

0x00000000, //28

0x00000000, //29

0x00000000, //30

0x00000000 //31

- **data memory (after).** all zeros
- **register file (after).**

0x00000000, //0

0x00000000, //1

0x00000000, //2

0x00000000, //3

0x00000000, //4

0x00000000, //5

0x00000000, //6

0x00000000, //7

0x00000000, //8

```
0x00000000, //9
0x00000000, //10
0x00000000, //11
0x00000000, //12
0x00000000, //13
0x00000000, //14
0x00000000, //15
0x00000000, //16
0x00000000, //17
0x00000000, //18
0x80000000, //19
0x7fffffff, //20
0x00000001, //21
0x0000000f, //22
0x0000000f, //23
0x00000000, //24
0x00000000, //25
0x00000000, //26
0x00000000, //27
0x00000000, //28
0x00000000, //29
0x00000000, //30
0x00000000 //31
```

6.4.2 While-Loop Program for Testing B-type Instructions:

The following testing program realizes the function of a `while-loop` statement in high-level programming language. To simplify, the base address of array a[i] and offset i are manually loaded into registers r25 and r22 as 0x00000000 and 0x00000001 in hexadecimal, respectively. Hence, array a[i] starts from address dmem[1] in data memory. The value of variable b is loaded into register r24 as 0x5af14901 in hexadecimal. In doing so, the while-loop statement is terminated after seven iterations because, array element a[8] is not equal to 0x5af14901.

```
/*
while (a[i] == b) {
   i++;
}
*/
// program memory
imem[0]: 0x002b1513 // Loop: SLLI x10, x22, 2;
imem[1]: 0x01950533 // ADD x10, x10, x25;
imem[2]: 0x00052483 // LW x9, 0x000(x10);
imem[3]: 0x01849663 // BNE x9, x24, Exit;
imem[4]: 0x001b0b13 // ADDI x22, x22, 0x001;
imem[5]: 0xfe0006e3 // BEQ x0, x0, Loop;
imem[6]: 0x00000013 // Exit: NOP;
// register file
r24: 0x5af14901
// data memory
dmem[0]: 0x00000000 // a[0]
dmem[1]: 0x5af14901 // a[1]
dmem[2]: 0x5af14901 // a[2]
dmem[3]: 0x5af14901 // a[3]
dmem[4]: 0x5af14901 // a[4]
dmem[5]: 0x5af14901 // a[5]
dmem[6]: 0x5af14901 // a[6]
dmem[7]: 0x5af14901 // a[7]
dmem[8]: 0xffffffff // a[8]
```

Contents of program memory, data memory and register files before and after the while-loop statement testing program is executed are listed as below.

- **program memory.**

  0x002b1513, //0

  0x01950533, //1

  0x00052483, //2

  0x01849663, //3

  0x001b0b13, //4

  0xfe0006e3, //5

- **data memory (before).**

  0x5af14901, //1

  0x5af14901, //2

  0x5af14901, //3

  0x5af14901, //4

  0x5af14901, //5

  0x5af14901, //6

  0x5af14901, //7

  0xffffffff, //8

- **register file (before).**

  0x00000000, //0

  0x00000000, //1

  0x00000000, //2

  0x00000000, //3

  0x00000000, //4

  0x00000000, //5

  0x00000000, //6

  0x00000000, //7

```
0x00000000, //8
0xeeeeeeee, //9
0xeeeeeeee, //10
0x00000000, //11
0x00000000, //12
0x00000000, //13
0x00000000, //14
0x00000000, //15
0x00000000, //16
0x00000000, //17
0x00000000, //18
0x00000000, //19
0x00000000, //20
0x00000000, //21
0x00000001, //22
0x00000000, //23
0x5af14901, //24
0x00000000, //25
0x00000000, //26
0x00000000, //27
0x00000000, //28
0x00000000, //29
0x00000000, //30
0x00000000 //31
```

- **data memory (after).**

```
0x5af14901, //1
0x5af14901, //2
```

0x5af14901, //3

0x5af14901, //4

0x5af14901, //5

0x5af14901, //6

0x5af14901, //7

0xffffffff, //8

- **register file (after).**

0x00000000, //0

0x00000000, //1

0x00000000, //2

0x00000000, //3

0x00000000, //4

0x00000000, //5

0x00000000, //6

0x00000000, //7

0x00000000, //8

0xffffffff, //9

0x00000020, //10

0x00000000, //11

0x00000000, //12

0x00000000, //13

0x00000000, //14

0x00000000, //15

0x00000000, //16

0x00000000, //17

0x00000000, //18

0x00000000, //19

```
0x00000000, //20
0x00000000, //21
0x00000008, //22
0x00000000, //23
0x5af14901, //24
0x00000000, //25
0x00000000, //26
0x00000000, //27
0x00000000, //28
0x00000000, //29
0x00000000, //30
0x00000000 //31
```

6.4.3 Procedure Call Program for Testing B-type Instructions:

The following testing program realizes the procedure call construct in high-level programming language. According to the RV32I reference manual, registers `x10-x17` are used to pass parameters and return value within a procedure call. In our case, we have four parameters-`a`, `b`, `c` and `d`-loaded into registers `x10-x13`, and one return value, `f`, loaded into register `x14`. At the start of the subroutine, values stored in registers `x5`, `x6` and, `x20` are pushed into stack in data memory because those registers are used for internal computation in the subroutine. In doing so, it prevents values stored in those registers from being polluted by the subroutine. In the stack operations, register `x2` is used as a stack pointer and it points to address (511) in data memory. To evaluate the expression `f = (a + b) - (c + d)`, registers `x5`, `x6` and `x20` are used and the return value is loaded into register `x14`.

After the subroutine is complete, registers `x5`, `x6` and `x20` are restored from stack, and the stack pointer (`x2`) is modified accordingly.

```
/*
int subroutine (int a, int b, int c, int d) {
   int f;
   f = (a + b) - (c + d);
   return f;
}
*/
// program memory
imem[0]: 0xff410113 // ADDI x2, x2, -12;
imem[1]: 0x00512423 // SW x5, 8(x2);
imem[2]: 0x00612223 // SW x6, 4(x2);
imem[3]: 0x01412023 // SW x20, 0(x2);
imem[4]: 0x00a582b3 // ADD x5, x10, x11;
imem[5]: 0x00c68333 // ADD x6, x12, x13;
imem[6]: 0x40628a33 // SUB x20, x5, x6;
imem[7]: 0x000a0513 // ADDI x10, x20, 0;
imem[8]: 0x00012a03 // LW x20, 0(x2);
imem[9]: 0x00412303 // LW x6, 4(x2);
imem[10]: 0x00812283 // LW x5, 8(x2);
imem[11]: 0x00c10113 // ADDI x2, x2, 12;
// data memory (stack)
dmem[508]: 0x00f000f0 // r20
dmem[509]: 0x0f0f0f0f // r6
dmem[510]: 0xf000f000 // r5
```

Contents of program memory, data memory and register files before and after the procedure call testing program is executed are listed as below.

- **program memory.**

  0xff410113, //0

  0x00512423, //1

0x00612223, //2

0x01412023, //3

0x00a582b3, //4

0x00c68333, //5

0x40628a33, //6

0x000a0513, //7

0x00012a03, //8

0x00412303, //9

0x00812283, //10

0x00c10113, //11

- **data memory (before).** all zeros
- **register file (before).**

0x00000000, //0

0x00000000, //1

0x000007fc, //2

0x00000000, //3

0x00000000, //4

0xf000f000, //5

0x0f0f0f0f, //6

0x00000000, //7

0x00000000, //8

0x00000000, //9

0x00000001, //10

0x7fffffff, //11

0xffffffff, //12

0x00000002, //13

0x00000000, //14

0x00000000, //15

0x00000000, //16

0x00000000, //17

0x00000000, //18

0x00000000, //19

0x00f000f0, //20

0x00000000, //21

0x00000000, //22

0x00000000, //23

0x00000000, //24

0x00000000, //25

0x00000000, //26

0x00000000, //27

0x00000000, //28

0x00000000, //29

0x00000000, //30

0x00000000 //31

- **data memory (after).**

  0x00f000f0, //508

  0x0f0f0f0f, //509

  0xf000f000, //510

  0x00000000 //511

- **register file (after).**

  0x00000000, //0

  0x00000000, //1

  0x000007fc, //2

  0x00000000, //3

```
0x00000000, //4
0xf000f000, //5
0x0f0f0f0f, //6
0x00000000, //7
0x00000000, //8
0x00000000, //9
0x7fffffff, //10
0x7fffffff, //11
0xffffffff, //12
0x00000002, //13
0x00000000, //14
0x00000000, //15
0x00000000, //16
0x00000000, //17
0x00000000, //18
0x00000000, //19
0x00f000f0, //20
0x00000000, //21
0x00000000, //22
0x00000000, //23
0x00000000, //24
0x00000000, //25
0x00000000, //26
0x00000000, //27
0x00000000, //28
0x00000000, //29
0x00000000, //30
```

0x00000000 //31

6.4.4 Nested Procedure Call Program for Testing B-type Instructions:

The following testing program realizes a recursive algorithm, which is implemented as a nested procedure call in high-level programming language. The algorithm performs recursive addition on integer `n`, and produces a sum which is equal to `n + (n−1) + (n−2) + ... + 2 + 1`. In this case, subroutine `iter_add` starts from address imem[10] in program memory, and parameter `n (n = 5)` is loaded into register `x10`. In addition, register `x1` is used to keep the returning addresses of a procedure call, and register `x2` serves as a stack pointer. In the body of subroutine `iter_add()`, values stored in registers `x1` and `x10` are recursively pushed back into stack until variable `n` is less than 1 ($n < 1$). By observing the stack in data memory, the first returning address is stored at dmem[510] as `0x00000068`, which refers to the returning address of subroutine `iter_add()`. Later on, the returning address of instruction `JAL r1, iter_add`, are pushed into stack one by one. After five iterations, stack growing terminates at `dmem[501]`, as variable `n` is less than 1 ($n < 1$). In this case, all the values of variable `n` and returning addresses are popped to registers recursively, and, finally, a sum which is equal to `n + (n−1) + (n−2) + ... + 2 + 1` is produced at register `x10`.

```
/*
int iter_add (int n) {
   if(n < 1) return (1);
   else return(n + iter_add(n-1));
}
*/
// program memory
imem[10]: 0xff810113 // iter_add: ADDI r2, r2, 0xff8
```

```
imem[11]: 0x00112223 // SW r1, 0x4(r2);
imem[12]: 0x00a12023 // SW r10, 0x4(r2);
imem[13]: 0xfff50293 // ADDI r5, r10, 0xfff;
imem[14]: 0x0002d863 // BGE r5, r0, L1;
imem[15]: 0x00100513 // ADDI r10, r10, 0x001;
imem[16]: 0x00810113 // ADDI r2, r2, 0x008;
imem[17]: 0x00008067 // JALR r0, 0x000(r1);
imem[18]: 0xfff50513 // L1: ADDI r10, r10, 0xfff;
imem[19]: 0xfddff0ef // JAL r1, iter_add;
imem[20]: 0x00050313 // ADDI r6, r10, 0x000;
imem[21]: 0x00012503 // LW r10, 0x0(r2);
imem[22]: 0x00412083 // LW r1, 0x4(r2);
imem[23]: 0x00810113 // ADDI r2, r2, 0x008;
imem[24]: 0x00a30533 // ADD r10, r10, r6;
imem[25]: 0x00008067 // JALR r0, 0x0(r1);
// data memory (stack)
dmem[501]: 0x00000001 // n-4
dmem[502]: 0x00000050 // returning address imem[20]
dmem[503]: 0x00000002 // n-3
dmem[504]: 0x00000050 // returning address imem[20]
dmem[505]: 0x00000003 // n-2
dmem[506]: 0x00000050 // returning address imem[20]
dmem[507]: 0x00000004 // n-1
dmem[508]: 0x00000050 // returning address imem[20]
dmem[509]: 0x00000005 // n
dmem[510]: 0x00000068 // returning address imem[26]
```

Contents of program memory, data memory and register files before and after the nested procedure call testing program is executed are listed as below.

- **program memory.**

  0xff810113, //10

  0x00112223, //11

0x00a12023, //12

0xfff50293, //13

0x0002d863, //14

0x00100513, //15

0x00810113, //16

0x00008067, //17

0xfff50513, //18

0xfddff0ef, //19

0x00050313, //20

0x00012503, //21

0x00412083, //22

0x00810113, //23

0x00a30533, //24

0x00008067, //25

- **data memory (before).** all zeros
- **register file (before).**

0x00000000, //0

0x00000068, //1

0x000007fc, //2

0x00000000, //3

0x00000000, //4

0x00000000, //5

0x00000000, //6

0x00000000, //7

0x00000000, //8

0x00000000, //9

0x00000005, //10

```
0x00000000, //11
0x00000000, //12
0x00000000, //13
0x00000000, //14
0x00000000, //15
0x00000000, //16
0x00000000, //17
0x00000000, //18
0x00000000, //19
0x00000000, //20
0x00000000, //21
0x00000000, //22
0x00000000, //23
0x00000000, //24
0x00000000, //25
0x00000000, //26
0x00000000, //27
0x00000000, //28
0x00000000, //29
0x00000000, //30
0x00000000 //31
```

- **data memory (after).**

```
0x00000001, //501
0x00000050, //502
0x00000002, //503
0x00000050, //504
0x00000003, //505
```

0x00000050, //506

0x00000004, //507

0x00000050, //508

0x00000005, //509

0x00000068, //510

- **register file (after).**

  0x00000000, //0

  0x00000068, //1

  0x000007fc, //2

  0x00000000, //3

  0x00000000, //4

  0x00000000, //5

  0x0000000a, //6

  0x00000000, //7

  0x00000000, //8

  0x00000000, //9

  0x0000000f, //10

  0x00000000, //11

  0x00000000, //12

  0x00000000, //13

  0x00000000, //14

  0x00000000, //15

  0x00000000, //16

  0x00000000, //17

  0x00000000, //18

  0x00000000, //19

  0x00000000, //20

```
0x00000000, //21
0x00000000, //22
0x00000000, //23
0x00000000, //24
0x00000000, //25
0x00000000, //26
0x00000000, //27
0x00000000, //28
0x00000000, //29
0x00000000, //30
0x00000000 //31
```

References

[1] Cloud FPGA, IBM Research Zurich. [Online]. Available:. https://www.zurich.ibm.com/cci/cloudFPGA/.

[2] FloPoCo Open Source Project. [Online]. Available:. http://flopoco.gforge.inria.fr/.

[3] ISO/IEC 9899:1999 PROGRAMMING LANGUAGES – C. [Online]. Available:. https://www.iso.org/standard/29237.html.

[4] Lattice Semiconductor: CE40 LP/HX/LM Family Handbook. [Online]. Available:. http://www.latticesemi.com/iCE40.

[5] LegUp Computing. https://www.legupcomputing.com/,

[6] Abstraction Levels and Hardware Design. [Online]. Available:. https://www.eetimes.com/abstraction-levels-and-hardware-design/, 2007.

[7] AMD Math Core Library Version 4.2 User's Guide (2008). [Online]. Available:. https://developer.amd.com/wordpress/media/2012/10/acml_userguide.pdf.pdf, 2008.

[8] Intel Math Kernel Library for the Linux os User's Guide (2008). [Online]. Available:. http://www.bgu.ac.il/intel_fortran_docs/mkl/userguide.pdf, 2008.

[9] AXI Reference Guide. [Online]. Available:. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, 2011.

[10] Cyclone V Device Overview: variable-precision DSP block. [Online]. Available:. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf, 2015.

[11] EDA is dead. What comes next is exciting. [Online]. Available:. https://medium.com/@saardrimer/eda-is-dead-what-comes-next-is-exciting-cd5f3301402b, 2016.

[12] Floating-Point IP Cores User Guide. [Online]. Available:. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altfp_mfug.pdf, 2016.

[13] NVidia Falcon Processor. [Online]. Available:. https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf, 2016.

[14] CORDIC v6.0 LogiCORE IP Product Guide. [Online]. Available:. https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf, 2017.

[15] CORDIC v6.0 LogiCORE IP Product Guide. [Online]. Available:. https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf, 2017.

[16] EC2 F1 Instances with FPGAs – Now Generally Available. [Online]. Available:. https://aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now-generally-available/, 2017.

[17] How to Program Your First FPGA Device. [Online]. Available:. https://software.intel.com/en-us/articles/how-to-program-your-first-fpga-device, 2017.

[18] FWRISC. [Online]. Available:. https://github.com/mballance/fwrisc, 2018.

[19] MicroBlaze Processor Reference Guide. [Online]. Available:. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf, 2018.

[20] Microsoft Launches FPGA-Powered Machine Learning for Azure Customers. [Online]. Available:. https://www.top500.org/news/microsoft-launches-fpga-powered-machine-learning-for-azure-customers/, 2018.

[21] Vexriscv: A FPGA friendly 32 bit RISC-V CPU implementation. [Online]. Available:. https://github.com/SpinalHDL/VexRiscv, 2018.

[22] Xilinx Vivado High-Level Synthesis. [online]. available:. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2018.

[23] Apple A12 Bionic specification. [Online]. Available:. https://en.wikipedia.org/wiki/Apple_A12, 2019.

[24] Approximations that depend on the floating point representation. [Online]. Available:. https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Approximations_that_depend_on_the_floating_point_representation, 2019.

[25] Elementary function. [Online]. Available:. https://en.wikipedia.org/wiki/Transcendental_function, 2019.

[26] LEON3 Processor. [Online]. Available:. https://www.gaisler.com/index.php/products/processors/leon3, 2019.

[27] Memory Prices (from 1957 to 2019). [Online]. Available:. https://jcmit.net/memoryprice.htm, 2019.

[28] Nios II Gen2 Processor Reference Guide, Altera Corp. [Online]. Available:. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf, 2019.

[29] OCRA: RISC-V by VectorBlox. [Online]. Available:. https://github.com/VectorBlox/orca, 2019.

[30] PicoRV32 - a size-optimized risc-v cpu. [Online]. Available:. https://github.com/cliffordwolf/picorv32, 2019.

[31] The Boston University RISC-V Processor Set (BRISC-V). [Online]. Available:. https://ascslab.org/research/briscv/release.html, 2019.

[32] The RISC-V Instruction Set Manual. [Online]. Available:. https://riscv.org/specifications/, 2019.

[33] Transcendental function. [Online]. Available:. https://en.wikipedia.org/wiki/Transcendental_function, 2019.

[34] Rashmi Agrawal, Sahan Bandara, Alan Ehret, Mihailo Isakov, Miguel Mark, and Michel A. Kinsy. The brisc-v platform: A practical teaching approach for computer architecture. In *Proceedings of the Workshop on Computer Architecture Education*, page 1. ACM, 2019.

[35] W.M.A.W. Ahmad, R.A.A. Rohim, Y. Norhayati, N.A. Aleng, and Z. Ali. Developing a new dimension of an applied exponential model: Application in biological sciences. *Engineering, Technology & Applied Science Research*, 8(4):3130–3134, 2018.

[36] Nikolaos Alachiotis and Alexandros Stamatakis. Efficient floating-point logarithm unit for FPGAs. In *IEEE Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[37] Amirhossein Alimohammad, Saeed F. Fard, and Bruce F. Cockburn. A unified architecture for the accurate and high-throughput implementation of six key elementary functions. *IEEE Transactions on Computers*, 59(4):449–456, 2009.

[38] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[39] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[40] Tomaš Bagala, Adam Fibich, Miroslav Hagara, Peter Kubinec, Oldrich Ondráček, Vladimír Štofanik, and Radovan Stojanović. Single clock square root algorithm based on binomial series and its fpga implementation. In *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4. IEEE, 2018.

[41] Sahan Bandara, Alan Ehret, Donato Kava, and Michel A. Kinsy. Brisc-v: An open-source architecture design space exploration toolbox. *arXiv preprint arXiv:1908.09992*, 2019.

[42] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *Advances in Neural Information Processing Systems*, pages 5145–5153, 2018.

[43] Jaisimha Bannur and A. Varma. The vlsi implementation of a square root algorithm. In *1985 IEEE 7th Symposium on Computer Arithmetic (ARITH)*, pages 159–165. IEEE, 1985.

[44] James F. Blinn. Floating-point tricks. *IEEE Computer Graphics and Applications*, 17(4):80–84, 1997.

[45] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, pages 33–36. ACM, 2011.

[46] Guangjie Cao, Huimin Du, Pengchao Wang, Qinqin Du, and Jialong Ding. A piecewise cubic polynomial interpolation algorithm for approximating elementary function. In *2015 14th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics)*, pages 57–64. IEEE, 2015.

[47] Y. Chandu and Maradi Megha. Design and implementation of high efficiency square root circuit using vedic mathematics. In *2017 2nd IEEE*

International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), pages 1148–1151. IEEE, 2017.

[48] Jing Chen. A pipelined, single precision floating-point logarithm computation unit in hardware. Master's thesis, 2012.

[49] Jing Chen and Xue Liu. A high-performance deeply pipelined architecture for elementary transcendental function evaluation. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 209–216. IEEE, 2017.

[50] Jing Chen, Xue Liu, and Jason H. Anderson. Software-specified fpga accelerators for elementary functions. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 54–61. IEEE, 2018.

[51] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[52] IEEE Standards Committee et al. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008:517, 2008.

[53] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. Multiplicative square root algorithms for fpgas. In *2010 International Conference on Field Programmable Logic and Applications*, pages 574–577. IEEE, 2010.

[54] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.

[55] Christian de Schryver. *FPGA Based Accelerators for Financial Applications*. Springer, 2015.

[56] Michael DeLorimier and André DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays (FPGA)*, pages 75–85. ACM, 2005.

[57] Jérémie Detrey and Florent de Dinechin. A parameterizable floating-point logarithm operator for FPGAs. In *Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005.*, pages 1186–1190. IEEE, 2005.

[58] Jérémie Detrey and Florent de Dinechin. A parameterized floating-point exponential function for FPGAs. In *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology (FPT), 2005.*, pages 27–34. IEEE, 2005.

[59] Jérémie Detrey, Florent de Dinechin, and Xavier Pujol. Return of the hardware floating-point elementary function. In *18th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 161–168. IEEE, 2007.

[60] Yong Dou, Stamatis Vassiliadis, Georgi Krasimirov Kuzmanov, and Georgi Nedeltchev Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays (FPGA)*, pages 86–95. ACM, 2005.

[61] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 36–43. IEEE, 2014.

[62] Paul George, Anmol Sahoo, Arjun Menon, and V. Kamakoti. Shakti: An open-source processor ecosystem.

[63] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John HeMessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Distributed shared memory: concepts and systems*, page 84, 1998.

[64] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[65] Jan Gray. Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–20. IEEE, 2016.

[66] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning (ICML)*, pages 1737–1746, 2015.

[67] John Harrison. A machine-checked theory of floating point arithmetic. In *International Conference on Theorem Proving in Higher Order Logics*, pages 113–130. Springer, 1999.

[68] Abul Hasnat, Tanima Bhattacharyya, Atanu Dey, Santanu Halder, and Debotosh Bhattacharjee. A fast fpga based architecture for computation of square root and inverse square root. In *2017 Devices for Integrated Circuit (DevIC)*, pages 383–387. IEEE, 2017.

[69] Tingting He, Jiyang Chen, Yuanwu Lei, Yuanxi Peng, and Baozhou Zhu. High-performance fp divider with sharing multipliers based on goldschmidt algorithm. *Chinese Journal of Electronics*, 26(2):292–298, 2017.

[70] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3):1–26, 2015.

[71] Wilson José, Ana Rita Silva, Horácio Neto, and Mário Véstias. Efficient implementation of a single-precision floating-point arithmetic unit on fpga. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.

[72] Andrew B. Kahng and Seokhyeong Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, pages 820–825. ACM, 2012.

[73] Nachiket Kapre and Jan Gray. Hoplite: Building austere overlay nocs for fpgas. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2015.

[74] Nachiket Kapre and Jan Gray. Hoplite: A deflection-routed directional torus noc for fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(2):14, 2017.

[75] Srinidhi Kestur, John D. Davis, and Oliver Williams. Blas comparison on FPGA, CPU and GPU. In *2010 IEEE computer society annual symposium on VLSI*, pages 288–293. IEEE, 2010.

[76] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban, and C-L Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18–27, 1993.

[77] Israel Koren. *Computer arithmetic algorithms.* AK Peters/CRC Press, 2001.

[78] Taek-Jun Kwon and Jeff Draper. Floating-point division and square root implementation using a taylor-series expansion algorithm with reduced look-up tables. In *2008 51st Midwest Symposium on Circuits and Systems*, pages 954–957. IEEE, 2008.

[79] Martin Langhammer and Bogdan Pasca. Single precision natural logarithm architecture for hard floating-point and dsp-enabled FPGAs. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 164–171. IEEE, 2016.

[80] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[81] Bingyi Li, Linlin Fang, Yizhuang Xie, He Chen, and Liang Chen. A unified reconfigurable floating-point arithmetic architecture based on cordic algorithm. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 301–302. IEEE, 2017.

[82] Erwan Libessart, Matthieu Arzel, Cyril Lahuec, and Francesco Andriulli. A scaling-less newton–raphson pipelined implementation for a fixed-point reciprocal operator. *IEEE Signal Processing Letters*, 24(6):789–793, 2017.

[83] Gerhard Lienhart, Andreas Kugel, and Reinhard Manner. Using floating-point arithmetic on FPGAs to accelerate scientific n-body simulations. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 182–191. IEEE, 2002.

[84] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[85] Janarbek Matai, Pingfan Meng, Lingjuan Wu, Brad Weals, and Ryan Kastner. Designing a hardware in the loop wireless digital channel emulator for software defined radio. In *2012 International Conference on Field-Programmable Technology (FPT)*, pages 206–214. IEEE, 2012.

[86] Eric Matthews, Zavier Aguila, and Lesley Shannon. Evaluating the performance efficiency of a soft-processor, variable-length, parallel-execution-unit architecture for fpgas using the risc-v isa. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8. IEEE, 2018.

[87] Eric Matthews and Lesley Shannon. Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[88] Andrew McCallum, Dayne Freitag, and Fernando C. N. Pereira. Maximum entropy markov models for information extraction and segmentation. In *International Conference on Machine Learning (ICML)*, volume 17, pages 591–598, 2000.

[89] Pramod K. Meher, Javier Valls, Tso-Bing Juang, K. Sridharan, and Koushik Maharatna. 50 years of cordic: Algorithms, architectures, and

applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(9):1893–1907, 2009.

[90] Suresh Mopuri and Amit Acharyya. Low-complexity methodology for complex square-root computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(11):3255–3259, 2017.

[91] Leonid Moroz, Volodymyr Samotyy, and Oleh Horyachyy. An effective floating-point reciprocal. In *The 4th IEEE International Symposium on Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems*, pages 280–285. Lviv, Ukraine, 2018.

[92] Jean-Michel Muller. *Elementary functions*. Springer, 2006.

[93] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2010.

[94] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015.

[95] Tao Niu and Haibin Shen. Low cost design for elementary function approximation based on piecewise quadratic interpolation. *Computer Engineering*, 39(8):285–287, 2013.

[96] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84. IEEE, 2016.

[97] John F. Palmer. The intel® 8087 numeric data processor. In *Proceedings of national computer conference*, pages 887–893, 1980.

[98] Behrooz Parhami. *Computer arithmetic*, volume 20. Oxford university press, 2010.

[99] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013.

[100] Rafat Rashid, J. Gregory Steffan, and Vaughn Betz. Comparing performance, productivity and scalability of the tilt overlay processor to

opencl hls. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 20–27. IEEE, 2014.

[101] Umesh Satpute, Kalyani Bhole, and Sushanta Reang. Optimized floating point square-root. In *2018 International Conference on Communication, Computing and Internet of Things (IC3IoT)*. IEEE, 2018.

[102] Süleyman Savas, Yassin Atwa, Tomas Nordström, and Zain Ul-Abdin. Using harmonized parabolic synthesis to implement a single-precision floating-point square root unit. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 621–626. IEEE, 2019.

[103] Philip Schneider and David H. Eberly. *Geometric tools for computer graphics*. Elsevier, 2002.

[104] T.R.W. Scogland, Heshan Lin, and Wu-chun Feng. A first look at integrated gpus for green high-performance computing. *Computer Science-Research and Development*, 25(3-4):125–134, 2010.

[105] Shashank Suresh, Spiridon F. Beldianu, and Sotirios G. Ziavras. Fpga and asic square root designs for high performance and power efficiency. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 269–272. IEEE, 2013.

[106] Jagadguru Swami, Sri Bharati Krisna, and Tirthaji Maharaja. Vedic mathematics or sixteen simple mathematical formulae from the veda, delhi (1965). *Motilal Banarsidas*, 1986.

[107] Ping-Tak Peter Tang. Table-driven implementation of the exponential function in ieee floating-point arithmetic. *ACM Transactions on Mathematical Software (TOMS)*, 15(2):144–157, 1989.

[108] Ping-Tak Peter Tang. Table-driven implementation of the logarithm function in ieee floating-point arithmetic. *ACM Transactions on Mathematical Software (TOMS)*, 16(4):378–400, 1990.

[109] Ping-Tak Peter Tang. Table-lookup algorithms for elementary functions and their error analysis. Technical report, Argonne National Lab., IL (USA), 1991.

[110] Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays (FPGA)*, pages 171–180. ACM, 2004.

[111] Frank Vahid. It's time to stop calling circuits "hardware". *Computer*, 40(9):106–108, 2007.

[112] Frank Vahid. *Digital Design with RTL Design, Verilog and VHDL*. John Wiley & Sons, 2010.

[113] Mário P. Vestias and Horácio C. Neto. Revisiting the newton-raphson iterative method for decimal division. In *2011 21st International Conference on Field Programmable Logic and Applications (FPL)*, pages 138–143, 2011.

[114] Oriol Vinyals and Gerald Friedland. A hardware-independent fast logarithm approximation with adjustable accuracy. In *IEEE Multimedia*, pages 61–65, 2008.

[115] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. PhD thesis, UC Berkeley, 2016.

[116] Henry Wong, Vaughn Betz, and Jonathan Rose. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, pages 5–14. ACM, 2011.

[117] Henry Wong, Vaughn Betz, and Jonathan Rose. Microarchitecture and circuits for a 200 mhz out-of-order soft processor memory system. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(1):7, 2016.

[118] Weng-Fai Wong and E. Gogo. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, 1994.

[119] Weng-Fai Wong and Eiichi Goto. Fast evaluation of the elementary functions in double precision. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 1, pages 349–358. IEEE, 1994.

[120] Weng-Fai Wong and Eiichi Goto. Fast evaluation of the elementary functions in single precision. *IEEE Transactions on Computers*, 44(3):453–457, 1995.

[121] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. On reconfiguration-oriented approximate adder design and its application. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 48–54. IEEE, 2013.

[122] Mohamed Zahran. Heterogeneous computing: Here to stay. *Communications of the ACM*, 60(3):42–45, 2017.