FAST ALGORITHMS FOR HIGH-PRECISION FLOATING-POINT ARITHMETIC

A DISSERTATION
SUBMITTED TO THE INSTITUTE FOR
COMPUTATIONAL AND MATHEMATICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

David Kai Zhang

December 2025

Abstract

Modern scientific computing faces a dilemma. Large-scale problems demand both *high performance*, to solve systems with billions of variables and terabytes of data, and *high precision*, to ensure that calculations are not invalidated by numerical rounding errors. These demands lie in tension, since existing methods for high-precision computer arithmetic are typically thousands of times slower than native machine-precision computation. Moreover, the 64-bit double precision standard, used by virtually all computers today, is no longer sufficient for challenging exascale workloads.

This dissertation presents a class of algorithms called *floating-point accumulation networks* (*FPANs*) for fast high-precision floating-point arithmetic. FPANs are the fastest known algorithms for this task, outperforming all existing software libraries by at least an order of magnitude. They reduce high-precision (128/192/256-bit) operations to branch-free sequences of several dozen machine-precision (64-bit) operations, enabling efficient parallel execution on modern SIMD CPUs and GPUs. To prove the correctness of these algorithms, we introduce a novel formal verification technique called the *SELTZO abstraction* that enables SMT solvers to analyze floating-point operations in a precision-independent fashion. We also describe an evolutionary search procedure used to discover new FPANs and benchmark the performance of our new algorithms.

Acknowledgments

My six-year journey at Stanford has been filled with adventure and excitement, love and loss, heartbreak and joy, and innumerably many unforgettable memories. If I were to give proper thanks to all of the wonderful people who have entered and influenced my life during the course of my Ph.D., I could easily fill up another dissertation. Instead, I will try to limit myself to a more reasonable (though inadequate) number of words of appreciation.

My first and foremost thanks go to my doctoral advisor, Alex Aiken. Alex took me under his wing when I was a fledgling Ph.D. student wandering aimlessly in the throes of the COVID-19 pandemic. He knows better than anyone else that my Ph.D. got off to a particularly rocky start. Nonetheless, Alex continued to assert his faith in my ability to succeed as a researcher, even at times when I found it difficult to believe in myself. He was always there to celebrate my successes and help me get back on my feet after failures. I thank Alex for improving my writing, lending me perspective, and most importantly, teaching me how to develop and pursue my own research ideas.

I also thank the other members of my doctoral committee, Fred Kjolstad, Gianluca Iaccarino, Eric Darve, and Li-Yang Tan, for their thoughtful and enthusiastic feedback on my dissertation and defense. Special thanks go to Gianluca and Eric for their steadfast leadership of ICME, keeping our community together in difficult and uncertain times.

Being at Stanford has given me the opportunity to learn from some of the most brilliant teachers and inspiring minds of our generation. I give particular thanks to Mary Wooters, Lisa Sauermann, Tadashi Tokieda, Emmanuel Candès, Dawson Engler, and Li-Yang Tan for teaching truly exceptional courses whose ways of thinking continue to influence me. I also thank Keith Winstein and Clark Barrett for memorable lessons outside the classroom.

I also participated in a variety of internships that exposed me to many intriguing corners of research outside academia. I thank Antoine Dujardin, Patrick Gill, and Pat McCormick for these enjoyable and informative experiences, and I hope we one day find an opportunity to work together again.

Turning now to my fellow students, I thank my colleagues in ICME, Tiffany Fan, Shaun Datta, Devansh Jalota, Josh Rozner, Yiping Lu, Tanya Parshakova, and Izzy Aguiar, for a fun and fulfilling first year with so many memorable days of teaching, learning, and solving problem sets together. In Computer Science, I thank Matthew Sotoudeh, Shiv Sundram, Zachary Yedidia, Rohan Yadav, AJ Root, Chris Gyurgyik, Alex Ozdemir, Anjiang Wei, Olivia Hsu, Scott Kovach, Prasanna Ramakrishnan, and Rohit Nema for all the entertaining and thought-provoking office conversations that kept bringing me into Gates and CoDa.

Outside of academics, Stanford also gave me a chance to explore the performing arts through its thriving student music scene. I give my thanks to all of the many students involved in Stanford's a cappella and musical theater organizations, with particular fondness for O-Tone, the group that took me in my first year. Becoming the director of O-Tone was the single experience that taught me the most about what it means to lead, inspire, and bring together a group of people. I've made too many lasting O-Tone friends to list here, but to name a few, I thank Vickie Wang, Stella Su, Thomas Knowles, Cynthia Liang, Jingwen Wu, Shina Peñaranda, Michelle Zheng, Ywen Lau, Sherwin Lai, Justin Huang, Sarah MacHarg, Valerie Tang, Nikhil Lyles, Ailsa Song, Parker Grove, Adam Sun, Kevin Chen, Jingbo Li, and Justin Park for the years we spent making music together.

I thank Hanna Lachnitt and Rebecca Grekin for our seemingly endless nights slaying demons in Gloomhaven together, and I congratulate both of you on our utterly heroic victory over the blood cultists.

I thank my old friend Jason Mitrovich for staying in touch after undergrad, making several trips out to the Bay Area, and regularly rekindling my interest in pure mathematics.

I thank Anthony Xie, Alan Cheng, Rowena Chen, and Carolyn Zou for the close personal bonds we've shared over the years and for all the help you've given me in particularly difficult times. I owe you all an immense debt of gratitude. I also thank Rachel Guo, Vivian Zhong, Michelle Cen, Lauren Ramlan, and Frieda Rong for your memorable contributions to my Stanford experience.

I of course give immense thanks to my family, my mother and father, my brother Alex and my sister Sunny, for bringing me to where I am today and supporting me through this difficult journey. And I give special thanks to all the members of the David Zhang Fan Club, who have made an effort to stay in touch with me through so many chapters of life.

Finally, my most heartfelt thanks go to my beloved partner, Lucy Zhang, who met me through O-Tone and supported me through so much of my directorial and doctoral journey. Lucy has always been there to lend a loving hand and a thoughtful ear, putting up with my long workdays and odd sleeping hours. She holds the singular distinction of being the one person who has brought me the most joy in my time at Stanford—a *truly* high bar. Words cannot express the love and gratitude I feel to have her in my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computers are indispensable tools in modern science and engineering. Computational modeling and simulation allow us to turn mathematical formulas and scientific theories into operational tools that solve problems and improve our understanding of the world. The successes of scientific computing abound in modern daily life, ranging from the nonlinear solvers used to design electrical grids to the molecular dynamics simulations used to discover new pharmaceuticals. Our vehicles are made safer and more energy-efficient by computational fluid dynamics and finite element analysis, while numerical weather prediction allows us to forecast floods, hurricanes, and other natural disasters, providing valuable early warnings that help avert the loss of human life.

Today, we live in the era of *exascale computing*, which means that our largest computer systems can execute more than 1,000,000,000,000,000,000 (one billion billion, or 10^{18}) arithmetic operations per second [92]. This immense computational power allows us to construct higher-fidelity models, simulate larger systems, and solve more challenging problems than ever before. However, this increase in scale is accompanied by increasingly stringent demands on *numerical precision*.

Since the 1980s, virtually all general-purpose computers have used 64-bit floating-point arithmetic to store and manipulate real numbers, an IEEE standard known as *double precision* or binary64 [44, 45, 46]. This is an inexact representation that computes each operation and rounds each number to a relative precision of roughly 16 decimal places. In other words, every time two numbers are added, subtracted, multiplied, or divided on a double-precision processor, the result carries a small *rounding error* of roughly one part in 10^{16}.

Historically, double precision has been sufficient for the majority of practical scientific computing tasks. However, in the exascale era, it is becoming increasingly common for a single simulation workload to involve 10^{18} or more floating-point operations. While an individual rounding error of one part in 10^{16} is relatively innocuous, the accumulation of 10^{18} such rounding errors can easily overwhelm a computation with noise and invalidate its results. This issue is even more pronounced in problems that exhibit *numerical instability*, i.e., heightened sensitivity to rounding errors, such as linear systems that are nearly singular. In the presence of numerical instability, even small-scale calculations can become unacceptably inaccurate when executed in double precision.

Numerical precision has become a practical concern in a variety of scientific fields, including computational fluid dynamics [7], numerical weather prediction [41], nonlinear dynamical systems [31], energy grid optimization [103], quantum chemistry [33, 34], lattice quantum chromodynamics [1], and full-genome metabolic modeling [67]. In response to these issues, scientific programmers have made repeated calls for the development and adoption of high-precision floating-point arithmetic [3, 4, 43, 66, 75].

Despite this growing threat to the accuracy, robustness, and reproducibility of scientific software, high-precision floating-point arithmetic is rarely employed in demanding computational workloads because current methods are tens to thousands of times slower than double precision. Conventional multiprecision libraries, such as GMP [38], MPFR [32], and Boost.Multiprecision [68], simulate high precision using software algorithms that often involve branching and dynamic memory allocation. Compared to double-precision computations, which use single-cycle operations natively supported in hardware, these software libraries are dramatically slower because they reimplement the logic of floating-point representation from scratch. This performance gap is even wider on *data-parallel* processors, such as SIMD CPUs and GPUs, since branching and dynamic memory allocation prevent efficient parallel execution.

An alternative approach that better leverages the capabilities of existing hardware is to use *error-free transformations* [79] to extend the precision of a floating-point processor. Error-free transformations are floating-point algorithms that exactly compute their own rounding errors, allowing them to be tracked and corrected or compensated for in a numerical program. For example, the Møller–Knuth TwoSum algorithm [60, 74] takes a pair of floating-point numbers (x, y) and computes both their rounded floating-point sum $s \coloneqq x \oplus y$ and the exact rounding error $e \coloneqq (x + y) - (x \oplus y)$ incurred in that sum. Here, $\oplus$ denotes rounded floating-point addition, while $+$ denotes exact mathematical addition.

Error-free transformations are useful algorithmic building blocks that have been employed by numerical programmers for over 50 years. They are used in dozens of software packages [5, 22, 28, 42, 55, 65, 87, 91, 99], including the Python [83] and Julia [93] standard libraries, and have been applied to solve problems in dense [64] and sparse [29] numerical linear algebra, high-precision quadrature [6], computational fluid dynamics [7, 41], robust computational geometry [89], quantum chemistry [33, 34], transcendental function evaluation [22, 91], and the discovery of new mathematical identities [3].

Unfortunately, the use of error-free transformations is laden with pitfalls. Tracking and correcting rounding errors is so tricky that even world experts in numerical analysis have made subtle mistakes in published research [54, 64, 73]. The fundamental issue is that different inputs to the same program can produce wildly different patterns of rounding error accumulation and propagation. A scheme for tracking and correcting these rounding errors must take all possible error patterns into account, and an oversight in just one pattern can produce catastrophic loss-of-precision bugs. In fact, some numerical analysts have called error-free transformations "an attractive nuisance, like an unfenced backyard swimming pool" [64], luring in unsuspecting programmers with promises of high precision and performance only to ensnare them in hidden depths of complexity.

To address these challenges, this dissertation introduces a class of algorithms called *floating-point accumulation networks* (FPANs) for fast high-precision floating-point arithmetic. FPANs are branch-free linear sequences of TwoSum operations that are used to implement high-precision arithmetic operations, including addition, subtraction, multiplication, division, and square root. As their name suggests, FPANs perform the task of *accumulation*, i.e., high-precision summation of multiple floating-point inputs with rounding errors explicitly tracked and corrected. By identifying FPANs as key subroutines that delimit the propagation of rounding errors, we reduce the analysis of all possible error patterns to a standardized subproblem with well-defined correctness conditions.

We then introduce a computer-aided verification technique called the *SELTZO abstraction* that leverages SMT solvers to automate the extensive casework of rounding error analysis. The SELTZO abstraction uses a coarse-grained representation of floating-point numbers to isolate the relevant variables involved in the FPAN correctness conditions. This dramatic reduction of the search space makes verification in the SELTZO abstraction millions of times faster than existing floating-point verification methods, such as bit-blasting. By reducing the analysis of thousands of rounding error patterns to an efficient computer-checkable form, our technique enables the development of FPANs with rigorous correctness guarantees and error bounds that provably hold for all inputs.

The availability of an efficient automatic verification procedure allows us to systematically explore the space of all FPANs to find the fastest possible algorithms for high-precision floating-point arithmetic. To achieve this, we present a stochastic search procedure that combines an evolutionary metaheuristic with simulated annealing to optimize over the space of FPANs that accomplish a specified task. Our search strategy models efficiency (which favors fewer operations) and robustness (which favors more operations) as competing evolutionary pressures whose interaction creates algorithms that are both fast and correct.

Using this evolutionary search procedure, we have discovered novel branch-free data-parallel algorithms for addition, subtraction, multiplication, division, and square root of two-term, three-term, and four-term floating-point expansions. These algorithms extend the effective precision of a floating-point processor to double, triple, or quadruple its native precision. Thus, on a double-precision processor, our algorithms provide fast, branch-free arithmetic operations at roughly quadruple, sextuple, or octuple precision. We present the best FPANs discovered for these tasks and benchmark their performance, demonstrating that our new algorithms significantly outperform all existing high-precision floating-point software libraries by 11.7×–69.3× in typical scientific computing workloads.

In summary, this dissertation makes the following contributions:

1. We introduce *floating-point accumulation networks* (*FPANs*) as a class of floating-point algorithms (Sections 3.1–3.3) and show that FPANs can be used to implement addition, subtraction, multiplication, division, and square root of floating-point expansions in a branch-free fashion (Section 3.4).

2. We formulate correctness conditions for FPANs that reduce the analysis of floating-point rounding errors to an efficiently computer-checkable form (Section 3.3).

3. We define the *SELTZO abstraction*, an abstract domain for reasoning about error-free transformations (Sections 4.1–4.2), and state a procedure that expresses the FPAN correctness conditions as SELTZO satisfiability problems (Sections 4.3–4.4).

4. We demonstrate that our procedure is millions of times faster than existing floating-point reasoning tools for verifying the FPAN correctness conditions (Section 4.5).

5. We devise an evolutionary search procedure that systematically explores the space of all FPANs to find fast candidate algorithms for a specified task (Section 5.1)

6. We present five novel FPANs with formally-verified error bounds for addition (Section 5.2) and multiplication (Section 5.3) of floating-point expansions with two, three, or four terms.

7. We demonstrate that our algorithms significantly outperform state-of-the-art software floating-point libraries in benchmarks of extended-precision BLAS kernels (Chapter 6).

Chapter 2

Background

Floating-point representation is the standard technique used to approximately represent real numbers in discrete information processing systems, such as digital computers. It is formally defined by IEEE Standard 754 [44, 45, 46] and used in virtually all general-purpose computing systems that exist today.

Despite its pervasive use, floating-point representation is often regarded as a difficult, esoteric subject whose details are poorly understood even among scientific and numerical programmers. Familiarity with those details is necessary to understand the main ideas of this dissertation, so in this chapter, we provide a concise, self-contained exposition of floating-point numbers, formats, and arithmetic.

2.1 Floating-Point Numbers

We begin by defining *floating-point representations* and their associated terminology. We must be careful to distinguish floating-point *representations* from floating-point *numbers* because a single number can have multiple distinct representations.

Definition 1 (*floating-point representation*, *base*, *precision*, *sign bit*, *exponent*, *mantissa*, *digit*)**.** A *floating-point representation* in *base* $b \in \mathbb{N}$ with *precision* $p \in \mathbb{N}$ is an ordered triple (s, e, m) consisting of a *sign bit* $s \in \{0, 1\}$, an *exponent* $e \in \mathbb{Z}$, and a *mantissa*

$m = (m_0, \ldots, m_{p-1})$, which is an ordered sequence of p *digits* $m_0, \ldots, m_{p-1} \in \{0, \ldots, b-1\}$.

The many-to-one correspondence between floating-point representations and floating-point numbers is specified by the *real value* function.

Definition 2 (*real value*, $\mathsf{RealVal}_{b,p}$)**.** Let (s, e, m) be a floating-point representation in base $b \in \mathbb{N}$ with precision $p \in \mathbb{N}$. The *real value* of (s, e, m), denoted by $\mathsf{RealVal}_{b,p}(s, e, m) \in \mathbb{R}$, is the following real number:

$$\mathsf{RealVal}_{b,p}(s, e, m) \coloneqq (-1)^s \times (m_0.m_1m_2 \ldots m_{p-1})_b \times b^e = (-1)^s \sum_{k=0}^{p-1} m_k b^{e-k} \tag{2.1}$$

Here, $(m_0.m_1m_2 \ldots m_{p-1})_b$ denotes the real number with integer part m_0 and fractional part $0.m_1m_2 \cdots m_{p-1}$ in base b, akin to a decimal expansion of the form 3.14159265.

Definition 3 (*floating-point number*)**.** A real number $x \in \mathbb{R}$ is a *floating-point number* in base $b \in \mathbb{N}$ with precision $p \in \mathbb{N}$ if there exists a floating-point representation (s, e, m) such that $x = \mathsf{RealVal}_{b,p}(s, e, m)$.

Example 1. The real number $2.25 = 2^1 + 2^{-2}$ is a floating-point number in base $b = 2$ with any precision $p \geq 4$. It has multiple representations whenever $p \geq 5$. For example, in precision $p = 6$, it admits three distinct floating-point representations:

$$\begin{aligned}
2.25 &= \mathsf{RealVal}_{2,6}(0, 1, (1, 0, 0, 1, 0, 0)) = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} \\
2.25 &= \mathsf{RealVal}_{2,6}(0, 2, (0, 1, 0, 0, 1, 0)) = 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} \\
2.25 &= \mathsf{RealVal}_{2,6}(0, 3, (0, 0, 1, 0, 0, 1)) = 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}
\end{aligned}$$

Note that 2.25 is *not* a floating-point number in base $b = 2$ with precision $p = 3$, but it *is* a floating-point number in base $b = 6$ with precision $p = 3$.

$$2.25 = \mathsf{RealVal}_{6,3}(0, 0, (2, 1, 3)) = 2 \cdot 6^0 + 1 \cdot 6^{-1} + 3 \cdot 6^{-2}$$

In general, the property of being a floating-point number is both base-dependent and precision-dependent. By definition, every floating-point number is a rational number whose denominator divides some power b^k of the base b, so in particular, no irrational number has a floating-point representation in any base with any precision.

To eliminate the ambiguity of a single number having multiple floating-point representations, we introduce a criterion that designates a particular canonical representation.

Definition 4 (*normalized*)**.** A floating-point representation is *normalized* if the first digit of its mantissa is nonzero.

Proposition 1 (Uniqueness of normalized representation)**.** *Every nonzero real number has at most one normalized floating-point representation in a particular base $b \geq 2$ with a particular precision $p \in \mathbb{N}$.*

Proof. Let $b \geq 2$ and $p \in \mathbb{N}$ be given. Suppose $x \in \mathbb{R} \setminus \{0\}$ has normalized floating-point representations $x = \mathsf{RealVal}_{b,p}(s, e, m) = \mathsf{RealVal}_{b,p}(s', e', m')$. We will show that (s, e, m) and (s', e', m') coincide. Clearly, $s = s'$, since no real number is simultaneously positive and negative. Moreover, normalization implies that $|x|$ lies in the half-open interval $[b^e, b^{e+1})$. Intervals of this form are disjoint for distinct values of e, which implies $e = e'$. Next, observe that we can write

$$x = \mathsf{RealVal}_{b,p}(s, e, m) = (-1)^s b^{e-(p-1)} \sum_{k=0}^{p-1} m_k b^{(p-1)-k} = (-1)^s b^{e-(p-1)} M \tag{2.2}$$

where M denotes the *integer mantissa* of the floating-point representation (s, e, m), i.e., the integer whose base-b expansion is the sequence $(m_0, \ldots, m_{p-1})$.

$$M := \sum_{k=0}^{p-1} m_k b^{(p-1)-k} \in \mathbb{Z} \tag{2.3}$$

We similarly have $x = (-1)^s b^{e-(p-1)} M'$ where $M' := \sum_{k=0}^{p-1} m'_k b^{(p-1)-k}$. Now, if $m_i \neq m'_i$ at any index i, then M and M' are distinct integers, which implies that:

$$|\mathsf{RealVal}_{b,p}(s, e, m) - \mathsf{RealVal}_{b,p}(s', e', m')| = b^{e-(p-1)} |M - M'| \geq b^{e-(p-1)} \tag{2.4}$$

This contradicts the assumption that $\mathsf{RealVal}_{b,p}(s, e, m) = \mathsf{RealVal}_{b,p}(s', e', m')$. Hence, we conclude that $m = m'$, which completes the proof. $\square$

The only real number that does not satisfy this uniqueness property is zero, which admits infinitely many floating-point representations of the form $(s, e, (0, \ldots, 0))$, none of which are normalized. We will later see that zero is separately handled as a special case in practical floating-point implementations.

2.2 Floating-Point Formats

To efficiently store and manipulate floating-point numbers in a digital computer, it is useful to encode them as bit vectors of a fixed finite size (usually 32 or 64 bits). This requires specifying a fixed base $b \in \mathbb{N}$, precision $p \in \mathbb{N}$, and exponent range $e \in \{e_{\text{min}}, \ldots, e_{\text{max}}\} \subset \mathbb{Z}$. A choice of these parameters is known as a *floating-point format*.

Definition 5 (*floating-point format*, *minimum normalized exponent*, *maximum exponent*)**.** A *floating-point format* is an ordered quadruple $(b, p, e_{\text{min}}, e_{\text{max}})$ consisting of a base $b \in \mathbb{N}$, a precision $p \in \mathbb{N}$, a *minimum normalized exponent* $e_{\text{min}} \in \mathbb{Z}$, and a *maximum exponent* $e_{\text{max}} \in \mathbb{Z}$ satisfying $b \geq 2$ and $e_{\text{min}} \leq e_{\text{max}}$.

IEEE Standard 754 [44, 45, 46] defines a collection of binary $(b = 2)$ and decimal $(b = 10)$ floating-point formats whose parameters are listed in Table 2.1. Among these, the most widely used in practice are binary32 and binary64, also known[1] as *single precision* and *double precision* or `float` and `double` in many programming languages. Smaller formats, such as binary16 and the nonstandard bfloat16 format, are becoming popular in machine learning applications where the precision of any single number is less important than the ability to manipulate large collections of numbers. However, outside this setting, binary32 and binary64 remain the primary workhorses of general-purpose and scientific computation.

[1]The original 1985 version of IEEE Standard 754 [44] referred to binary32 and binary64 as "single precision" and "double precision", respectively. These names were changed in the 2008 revision [45] to avoid ambiguity with the newly added decimal32 and decimal64 formats, but in colloquial usage outside the standard, the names "single precision" and "double precision" are overwhelmingly more common.

Format	Base b	Precision p	Exponent range $\{e_{\min}, \dots, e_{\max}\}$
bfloat16	$b = 2$	$p = 8$	$e \in \{-126, \dots, +127\}$
binary16	$b = 2$	$p = 11$	$e \in \{-14, \dots, +15\}$
binary32	$b = 2$	$p = 24$	$e \in \{-126, \dots, +127\}$
binary64	$b = 2$	$p = 53$	$e \in \{-1022, \dots, +1023\}$
binary128	$b = 2$	$p = 113$	$e \in \{-16382, \dots, +16383\}$
binary$\{k\}$	$b = 2$	$p = k - \lfloor 4 \log_2 k \rceil + 13$	$e \in \{-2^{k-p-1} + 2, \dots, 2^{k-p-1} - 1\}$
decimal32	$b = 10$	$p = 7$	$e \in \{-95, \dots, +96\}$
decimal64	$b = 10$	$p = 16$	$e \in \{-383, \dots, +384\}$
decimal128	$b = 10$	$p = 34$	$e \in \{-6143, \dots, +6144\}$
decimal$\{k\}$	$b = 10$	$p = 9k/32 - 2$	$e \in \{-3 \cdot 2^{k/16+3} + 1, \dots, 3 \cdot 2^{k/16+3}\}$

Table 2.1: Parameters of the floating-point formats defined by IEEE Standard 754 and the nonstandard bfloat16 format commonly used in deep learning accelerators. Two parametric families, binary$\{k\}$ and decimal$\{k\}$, are defined for values of $k \geq 128$ that are divisible by 32. Here, $\lfloor x \rceil$ denotes x rounded to the nearest integer. Note that bfloat16, binary16, and binary32 are special cases that do not follow the general pattern for binary$\{k\}$.

With these formats defined, we now describe the standard scheme for encoding floating-point numbers as fixed-size bit vectors. To simplify our exposition, we consider only base $b = 2$; other bases introduce additional complications that are irrelevant for our purposes.

Definition 6 (*IEEE binary compatible, exponent width*)**.** We say that a floating-point format $(b, p, e_{\min}, e_{\max})$ is *IEEE binary compatible* if $b = 2$ and $e_{\max} = 1 - e_{\min} = 2^{w-1} - 1$ for some $w \in \mathbb{N}$, which is called the *exponent width* of the format.

Definition 7 (*IEEE encoding, biased exponent*)**.** Let (s, e, m) be a normalized floating-point representation in an IEEE binary compatible format $(2, p, e_{\min}, e_{\max})$ with exponent width w. Assume $e_{\min} \leq e \leq e_{\max}$. The *IEEE encoding* of (s, e, m) is the following bit vector $(b_{k-1}, \dots, b_0)$ of size $k \coloneqq w + p$:

- The most significant bit $b_{k-1} \coloneqq s$ is the sign bit.
- The next w bits $(b_{k-2}, \dots, b_{p-1})$ are the binary expansion of the positive integer $E \coloneqq e - e_{\min} + 1$, which is called the *biased exponent*.
- The final $p - 1$ bits $(b_{p-2} \coloneqq m_1, b_{p-3} \coloneqq m_2, \dots, b_0 \coloneqq m_{p-1})$ are the elements of the

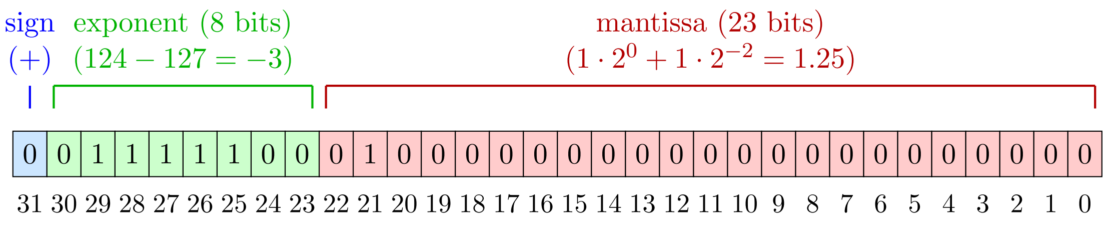


Figure 2.1: IEEE encoding of $2^{-3} \times 1.25 = 0.15625$ in the binary32 format with exponent width $w = 8$. Here, $(01111100)_2 = 124$ is the biased exponent, and $2^{w-1} - 1 = 127$ is the exponent bias. Recall that the mantissa leading bit $m_0 = 1$ is not explicitly stored.

mantissa, ordered from most to least significant, omitting the leading bit m_0.

To accord with common practice in computer science, we use opposite indexing conventions for bit vectors (b_{k-1} most significant, b_0 least significant) and mantissas (m_0 most significant, m_{p-1} least significant). The biased exponent E lies in the range $1 \leq E \leq 2^w - 2$, so its binary expansion always fits into w bits. Moreover, $m_0 = 1$ is guaranteed by the requirement that (s, e, m) be normalized, so there is no need to explicitly store m_0. An example of IEEE encoding in the binary32 format is illustrated in Figure 2.1.

The corresponding decoding scheme is obtained by reversing this process. Namely, given a bit vector $(b_{k-1}, \ldots, b_0)$, we extract the sign bit $s = b_{k-1}$, the biased exponent $E = b_{k-2} \cdot 2^{w-1} + \cdots + b_{p-1} \cdot 2^0$ (from which we recover the exponent $e = E + e_{\min} - 1$), and the mantissa $m = (1, b_{p-2}, \ldots, b_0)$. To fully specify the decoding process, we must define the interpretation of $E = 0$ and $E = 2^w - 1$, the two cases not permitted by Definition 7. IEEE Standard 754 uses $E = 0$ to represent numbers with very small magnitudes, including zero, while $E = 2^w - 1$ encodes special non-numeric values that are used for detecting and reporting error conditions.

Definition 8 (*floating-point domain*, $\mathbb{FP}(b, p, e_{\min}, e_{\max})$, *normal values*, *subnormal values*, *positive zero*, `+0.0`, *negative zero*, `-0.0`, *positive infinity*, `+Inf`, *negative infinity*, `-Inf`, *not-a-number*, `NaN`, *floating-point values*, *finite values*, *special values*)**.** Let $(b, p, e_{\min}, e_{\max})$ be a floating-point format. Its *floating-point domain*, denoted by $\mathbb{FP}(b, p, e_{\min}, e_{\max})$, is the set consisting of:

- all nonzero real values $\mathsf{RealVal}_{b,p}(s, e, m)$ of normalized floating-point representations satisfying $e_{\min} \leq e \leq e_{\max}$, which are called *normal values*;
- all nonzero real values $\mathsf{RealVal}_{b,p}(s, e_{\min}, m)$ of floating-point representations satisfying $e = e_{\min}$ and $m_0 = 0$, which are called *subnormal values*;
- the symbol `+0.0`, called *positive zero*;
- the symbol `-0.0`, called *negative zero*;
- the symbol `+Inf`, called *positive infinity*;
- the symbol `-Inf`, called *negative infinity*;
- the symbol `NaN`, called *not-a-number.*

The elements of $\mathbb{FP}(b, p, e_{\min}, e_{\max})$ are called *floating-point values.* The normal values, subnormal values, positive zero, and negative zero are collectively called *finite values.* The symbols `+0.0`, `-0.0`, `+Inf`, `-Inf`, and `NaN` are collectively called *special values.*

This definition introduces a distinction between floating-point *numbers*, which are real numbers that admit a floating-point representation, and floating-point *values*, which include the nonzero floating-point numbers and the special values `+0.0`, `-0.0`, `+Inf`, `-Inf`, and `NaN`.

Definition 9 (*IEEE value*, $\mathsf{IEEEVal}_{2,p,w}$)**.** Let $(2, p, e_{\min}, e_{\max})$ be an IEEE binary compatible floating-point format with exponent width w. The *IEEE value* of a bit vector $(b_{k-1}, \ldots, b_0)$ of size $k = w + p$, denoted by $\mathsf{IEEEVal}_{2,p,w}(b_{k-1}, \ldots, b_0)$, is the element of $\mathbb{FP}(2, p, e_{\min}, e_{\max})$ obtained as follows. Let $s := b_{k-1}$ and $E := b_{k-2} \cdot 2^{w-1} + \cdots + b_{p-1} \cdot 2^0$.

1. (Zero case.) If $E = 0$ and $b_{p-2} = \cdots = b_0 = 0$, then:

$$\mathsf{IEEEVal}_{2,p,w}(b_{k-1}, \ldots, b_0) := \begin{cases} \texttt{+0.0} \text{ if } s = 0 \\ \texttt{-0.0} \text{ if } s = 1 \end{cases} \tag{2.5}$$

2. (Subnormal case.) If $E = 0$ and any of the bits $(b_{p-2}, \ldots, b_0)$ are nonzero, then:

$$\mathsf{IEEEVal}_{2,p,w}(b_{k-1}, \ldots, b_0) \coloneqq \mathsf{RealVal}_{b,p}(s, e_{\min}, (0, b_{p-2}, \ldots, b_0)) \tag{2.6}$$

3. (Normal case.) If $1 \leq E \leq 2^w - 2$, then:

$$\mathsf{IEEEVal}_{2,p,w}(b_{k-1}, \ldots, b_0) \coloneqq \mathsf{RealVal}_{b,p}(s, E + e_{\min} - 1, (1, b_{p-2}, \ldots, b_0)) \tag{2.7}$$

4. (Infinity case.) If $E = 2^w - 1$ and $b_{p-2} = \cdots = b_0 = 0$, then:

$$\mathsf{IEEEVal}_{2,p,w}(b_{k-1}, \ldots, b_0) \coloneqq \begin{cases} \texttt{+Inf} \text{ if } s = 0 \\ \texttt{-Inf} \text{ if } s = 1 \end{cases} \tag{2.8}$$

5. (Not-a-number case.) If $E = 2^w - 1$ and any of $(b_{p-2}, \ldots, b_0)$ are nonzero, then:

$$\mathsf{IEEEVal}_{2,p,w}(b_{k-1}, \ldots, b_0) \coloneqq \texttt{NaN} \tag{2.9}$$

In summary, zero and subnormal values have biased exponent $E = 0$, normal values have $1 \leq E \leq 2^w - 2$, and infinity and not-a-number values have $E = 2^w - 1$. Aside from `NaN`, which is represented by many different bit patterns, all other floating-point values have a unique IEEE encoding. In particular, there is no overlap between normal values (which have magnitude $\geq b^{e_{\min}}$) and subnormal values (which have magnitude $< b^{e_{\min}}$).

In many cases, analysis involving floating-point numbers is considerably simplified by ignoring the bounded exponent range $e_{\min} \leq e \leq e_{\max}$, the multiple representations `+0.0` and `-0.0` of zero, and the special values `+Inf`, `-Inf`, and `NaN`. For these situations, we introduce the *unbounded floating-point domain.*

Definition 10 (*unbounded floating-point domain*, $\mathbb{FP}(b, p)$)**.** Let $b \geq 2$ and $p \in \mathbb{N}$. The *unbounded floating-point domain*, denoted by $\mathbb{FP}(b, p)$, is the set consisting of all real values $\mathsf{RealVal}_{b,p}(s, e, m)$ of floating-point representations (s, e, m) in base b with precision p, including zero, with no restriction on the exponent $e \in \mathbb{Z}$.

Note that $\mathbb{FP}(b, p)$ is a subset of $\mathbb{R}$ while $\mathbb{FP}(b, p, e_{\min}, e_{\max})$ is not. Similarly, $\mathbb{FP}(b, p)$ contains the real number zero while $\mathbb{FP}(b, p, e_{\min}, e_{\max})$ only contains the special symbols `+0.0` and `-0.0`.

2.3 Floating-Point Arithmetic

To perform calculations with floating-point numbers, we must define mathematical operations on the floating-point domain $\mathbb{FP}(b, p, e_{\min}, e_{\max})$. This is nontrivial even for basic arithmetic operations, such as addition and subtraction, because the sum or difference of two elements of $\mathbb{FP}(b, p, e_{\min}, e_{\max})$ may not lie in $\mathbb{FP}(b, p, e_{\min}, e_{\max})$. Therefore, the result of each operation must be *rounded* to the nearest floating-point number. We begin our discussion of floating-point arithmetic by formally defining this rounding procedure.

Definition 11 (*unbounded rounding function*, $\mathsf{RNE}_{b,p}$)**.** Let $b \geq 2$ and $p \in \mathbb{N}$. The *unbounded rounding function* $\mathsf{RNE}_{b,p} : \mathbb{R} \to \mathbb{FP}(b, p)$ sends each real number $x \in \mathbb{R}$ to the closest element of $\mathbb{FP}(b, p)$. When x is equidistant to two adjacent floating-point numbers $x_1, x_2 \in \mathbb{FP}(b, p)$, we define $\mathsf{RNE}_{b,p}(x)$ to be whichever of x_1 and x_2 has a mantissa whose least significant digit is even, or if both have the same[2] parity, whichever of x_1 and x_2 is larger in magnitude.

We now extend this rounding procedure from the unbounded floating-point domain $\mathbb{FP}(b, p)$ to the standard floating-point domain $\mathbb{FP}(b, p, e_{\min}, e_{\max})$. Here, we make use of the special symbols `+0.0`, `-0.0`, `+Inf`, and `-Inf` to absorb numbers that fall outside the bounded exponent range.

Definition 12 (*rounding function*, $\mathsf{RNE}_{b,p,e_{\min},e_{\max}}$, *overflow*, *underflow*)**.**
Let $(b, p, e_{\min}, e_{\max})$ be a floating-point format. The *rounding function* $\mathsf{RNE}_{b,p,e_{\min},e_{\max}}$:

[2]The same-parity case occurs in pathological situations, such as precision $p = 1$. In base $b = 2$, 1.5 is equidistant to $1.0 = 2^0 \times 1$ and $2.0 = 2^1 \times 1$, both of which have an odd mantissa. Thus, $\mathsf{RNE}_{2,1}(1.5) := 2.0$.

$\mathbb{R} \setminus \{0\} \to \mathbb{FP}(b, p, e_{\min}, e_{\max})$ sends each nonzero real number $x \in \mathbb{R} \setminus \{0\}$ to the floating-point value defined as follows. Let $\mathsf{RNE}_{b,p}(x) = \mathsf{RealVal}_{b,p}(s, e, m)$ denote the result of the unbounded rounding function.

- If $e_{\min} \leq e \leq e_{\max}$, then:

$$\mathsf{RNE}_{b,p,e_{\min},e_{\max}}(x) \coloneqq \mathsf{RealVal}_{b,p}(s, e, m) \tag{2.10}$$

- If $e > e_{\max}$, then:

$$\mathsf{RNE}_{b,p,e_{\min},e_{\max}}(x) \coloneqq \begin{cases} \texttt{+Inf} \text{ if } s = 0 \\ \texttt{-Inf} \text{ if } s = 1 \end{cases} \tag{2.11}$$

  In this situation, we say that *overflow* has occurred.

- If $e < e_{\min}$, then $\mathsf{RNE}_{b,p,e_{\min},e_{\max}}(x)$ is defined to be the closest element to x in the set $\{\mathsf{RealVal}_{b,p}(s, e_{\min}, m') : m'_0 = 0\}$ of subnormal floating-point numbers, breaking ties by the parity of the mantissa as in Definition 11. If the closest element is zero, then we define $\mathsf{RNE}_{b,p,e_{\min},e_{\max}}(x)$ to be `+0.0` if $s = 0$ and `-0.0` if $s = 1$. When $x \neq 0$ and $\mathsf{RNE}_{b,p,e_{\min},e_{\max}}(x)$ is zero, we say that *underflow* has occurred.

It follows from this definition that overflow occurs when $|x| \geq b^{e_{\max}}(b - \frac{1}{2}b^{-(p-1)})$ and underflow occurs when $0 < |x| \leq \frac{1}{2}b^{e_{\min}-(p-1)}$. To simplify notation, whenever the floating-point format $(b, p, e_{\min}, e_{\max})$ is clear from context, we simply write $\mathsf{RNE}(x)$.

The rounding function RNE described above is one of five *rounding-direction attributes* defined by IEEE Standard 754 [44, 45, 46], therein named $\mathsf{roundTiesToEven}$. This is specified to be the default rounding-direction attribute in all conforming floating-point implementations and is by far the most widely used in practice. In fact, $\mathsf{roundTiesToEven}$ is the only rounding-direction attribute available in many common programming environments, including Python, JavaScript, and WebAssembly [27, 85]. In analyses of floating-point arithmetic [11, 12], it is common to assume that $\mathsf{roundTiesToEven}$ is the only rounding-direction attribute in use. We adopt this standard assumption throughout this dissertation.

Assumption 1. All floating-point operations are rounded to nearest with ties broken to even (i.e., executed with the $\mathsf{roundTiesToEven}$ rounding-direction attribute).

The rounding function RNE allows us to define mathematical operations on the floating-point domain $\mathbb{FP}(b, p, e_{\min}, e_{\max})$ by composing RNE with the corresponding operations on the real numbers $\mathbb{R}$. To complete such a definition, we must also decide how to handle the special values $\{\texttt{+0.0}, \texttt{-0.0}, \texttt{+Inf}, \texttt{-Inf}, \texttt{NaN}\}$ and how to round zero, since RNE is defined on $\mathbb{R} \setminus \{0\}$. The general principle adopted by IEEE Standard 754 [46, Section 6.1] is to regard `+0.0`, `-0.0`, `+Inf`, and `-Inf` as representations of one-sided limiting processes. For example, the floating-point quotient $1 \oslash (\texttt{+0.0})$ is regarded as the one-sided limit $\lim_{x \to 0^+} 1/x = +\infty$ and is hence defined to be `+Inf`. Similarly, $(\texttt{-0.0}) \oplus (\texttt{-0.0})$ is regarded as the double limit $\lim_{x \to 0^-, y \to 0^-} x + y$, which approaches zero from below, and is hence defined to be `-0.0`.

The special value `NaN` is returned by operations that produce indeterminate limiting processes, such as $\lim_{x \to +\infty, y \to -\infty} x + y$, which can take on any real value as $x \to +\infty$ and $y \to -\infty$. Hence, $(\texttt{+Inf}) \oplus (\texttt{-Inf})$ is defined to be `NaN`. Most[3] operations specified by IEEE Standard 754 [46, Section 9.2.1] return `NaN` when any input is `NaN`, causing `NaN` to propagate through floating-point operations in an infectious manner. This virus-like mechanism is designed to allow floating-point programs to signal the occurrence of an error without trapping or otherwise interrupting program control flow.

To demonstrate the application of these principles, we state the formal definition of floating-point addition below. Observe that, outside a single invocation of $\mathsf{RNE}(x + y)$, the rest of the definition merely specifies behavior on special values and the sign of zero.

Definition 13 (floating-point sum, $\oplus$)**.** Let $(b, p, e_{\min}, e_{\max})$ be a floating-point format. Given two floating-point values $x, y \in \mathbb{FP}(b, p, e_{\min}, e_{\max})$, the *floating-point sum* of x and y, denoted by $x \oplus y \in \mathbb{FP}(b, p, e_{\min}, e_{\max})$, is the floating-point value defined as follows:

- If x is `NaN` or y is `NaN`, then $x \oplus y$ is defined to be `NaN`.

[3]The only exceptions to this rule are the minimumNumber and maximumNumber operations, which discard `NaN` inputs; the copySign operation, which discards `NaN` as its second input; the hypot operation, which discards `NaN` when its other input is infinity; and the power operations (pow and pown), which are defined so that $\texttt{NaN}^0 = 1^{\texttt{NaN}} = 1$.

- If x and y are both infinity with the same sign, then $x \oplus y$ is defined to be infinity of the same sign as x and y.
- If x and y are both infinity with opposite signs, then $x \oplus y$ is defined to be `NaN`.
- If x is infinity (of either sign) and y is finite, then $x \oplus y$ is defined to be x. Similarly, if x is finite and y is infinity, then $x \oplus y$ is defined to be y.
- If x and y are both finite and the exact real value of the sum $x + y$ is nonzero, then $x \oplus y$ is defined to be $\mathsf{RNE}(x + y)$. Here, `+0.0` and `-0.0` are both interpreted as the real number 0 for the purpose of determining the exact result $x + y$.
- If x and y are both finite, the exact real value of the sum $x + y$ is zero, and at least one of x and y is not `-0.0`, then $x \oplus y$ is defined to be `+0.0`.
- If $x =$ `-0.0` and $y =$ `-0.0`, then $x \oplus y$ is defined to be `-0.0`.

Although these rules are intuitively reasonable in light of the preceding discussion, they have many counterintuitive consequences demonstrated by the following examples.

Example 2. The function $x \mapsto x \oplus ($`+0.0`$)$ is *not* the identity function on $\mathbb{FP}(b, p, e_{\min}, e_{\max})$ because it sends `-0.0` to `+0.0`. In contrast, the function $x \mapsto x \oplus ($`-0.0`$)$ *is* the identity function. This means that the instruction $x \mapsto x \oplus ($`-0.0`$)$ can be safely optimized out of a floating-point program, while the instruction $x \mapsto x \oplus ($`+0.0`$)$ cannot be removed without potentially changing the behavior of the program on some inputs.

Example 3. Floating-point addition is *not* associative. Indeed, consider $100 \oplus 4 \oplus 2$ in base $b = 10$ with precision $p = 2$. If this sum is associated to the left, then its value is 100, since $100 \oplus 4 = \mathsf{RNE}(104) = 100$ and $100 \oplus 2 = \mathsf{RNE}(102) = 100$. On the other hand, if associated to the right, its value is 110, since $4 \oplus 2 = \mathsf{RNE}(6) = 6$ and $100 \oplus 6 = \mathsf{RNE}(106) = 110$.

For the sake of brevity, we omit full formal definitions of the remaining floating-point operations. We merely introduce our notation and give summary remarks. Following

Knuth's convention [60], we use circled operators to distinguish rounded operations on $\mathbb{FP}(b, p, e_{\min}, e_{\max})$ from their exact counterparts on $\mathbb{R}$.

$$\left.\begin{aligned} x \oplus y &\coloneqq \mathsf{RNE}(x + y) \\ x \ominus y &\coloneqq \mathsf{RNE}(x - y) \\ x \otimes y &\coloneqq \mathsf{RNE}(xy) \\ x \oslash y &\coloneqq \mathsf{RNE}(x/y) \\ \sqrt[\circ]{x} &\coloneqq \mathsf{RNE}(\sqrt{x}) \end{aligned}\right\} \begin{array}{l}\text{when } x \text{ and } y \text{ are finite} \\ \text{and the argument of } \mathsf{RNE} \\ \text{is well-defined and nonzero}\end{array} \tag{2.12}$$

Floating-point subtraction $x \ominus y$ is formally defined as $x \oplus (-y)$, where $-y$ is obtained from y by flipping its sign bit ($-\texttt{NaN} = \texttt{NaN}$). The sign of the result in floating-point multiplication and division is determined by taking the logical exclusive-or of the signs of the inputs. This, in particular, defines the sign of zero when it arises as a product or quotient. The floating-point square root of zero is defined to be zero with the same sign as the input (in particular, $\sqrt[\circ]{\texttt{-0.0}} \coloneqq \texttt{-0.0}$), while $\sqrt[\circ]{x} \coloneqq \texttt{NaN}$ for all nonzero negative x.

Most floating-point processors today also support a *fused multiply-add* operation, which combines multiplication and addition into a single operation that is only rounded once.

$$\mathsf{FMA}(x, y, z) \coloneqq \mathsf{RNE}(xy + z) \tag{2.13}$$

The lack of intermediate rounding causes $\mathsf{FMA}(x, y, z)$ to be more accurate than $x \otimes y \oplus z$ in many useful situations, such as the evaluation of polynomials and vector dot products.

2.4 Quantifying Rounding Errors

The presence of rounding causes each floating-point operation to introduce a small *rounding error* into a numerical program. These rounding errors can compound when multiple floating-point operations are chained together, significantly degrading the accuracy of the final result. In general, it can be very difficult to determine what floating-point precision p is necessary for a given algorithm to achieve a specified final error bound ϵ. The field of numerical analysis, which spans hundreds of textbooks and thousands of research papers, aims to answer this very question [94].

We begin by considering the rounding error introduced by a single application of RNE to a real number $x \in \mathbb{R}$. The distance between x and $\mathsf{RNE}(x)$ is characterized by a numerical constant called[4] the *unit roundoff* [86] defined for each floating-point format as follows.

Definition 14 (*unit roundoff*, $\mathbf{u}$)**.** Let $(b, p, e_{\min}, e_{\max})$ be a floating-point format. The *unit roundoff* for this format, denoted by $\mathbf{u} \in \mathbb{R}$, is the following[5] real constant:

$$\mathbf{u} := \frac{1}{2} b^{-(p-1)} \tag{2.14}$$

Intuitively, the unit roundoff is defined so that $\mathbf{u}x$ is an upper bound on the distance from x to $\mathsf{RNE}(x)$. In other words, $\mathbf{u}$ is an upper bound on the *relative error* $|\mathsf{RNE}(x) - x|/|x|$, as shown by the following proposition in the unbounded floating-point domain $\mathbb{FP}(b, p)$.

Proposition 2 (Unit roundoff bounds relative rounding error)**.** *Let $b \geq 2$ and $p \in \mathbb{N}$, and let $\mathsf{RNE} : \mathbb{R} \to \mathbb{FP}(b, p)$ denote the unbounded rounding function. For every $x \in \mathbb{R}$, there exists $\delta \in \mathbb{R}$ such that:*

$$\mathsf{RNE}(x) = (1 + \delta)x \qquad \textit{where} \qquad |\delta| \leq \mathbf{u} \tag{2.15}$$

Proof. Let $x \in \mathbb{R}$ be given. If $x \in \mathbb{FP}(b, p)$, then $x = \mathsf{RNE}(x)$ and the claim holds trivially. Otherwise, assume without loss of generality that $x > 0$, and let $x_1, x_2 \in \mathbb{FP}(b, p)$ denote the immediate predecessor and successor of x in $\mathbb{FP}(b, p)$, respectively. Let $e \in \mathbb{Z}$ denote the exponent of x_1. By definition, x_1 and x_2 are adjacent integer multiples of $b^{e-(p-1)}$, so the distance from x to the closer of x_1 and x_2 is at most $\frac{1}{2} b^{e-(p-1)}$. Moreover, $x \geq b^e$, so

[4]Some sources refer to this constant as the *machine epsilon* ϵ_{mach}. We avoid this term because it has several inequivalent definitions in common use. Some sources use machine epsilon synonymously with unit roundoff ($\epsilon_{\text{mach}} = \mathbf{u}$), while others define machine epsilon as the distance between consecutive floating-point numbers ($\epsilon_{\text{mach}} = 2\mathbf{u}$).

[5]Our definition of the unit roundoff $\mathbf{u} := \frac{1}{2} b^{-(p-1)}$ is appropriate when rounding to the nearest floating-point number. In other rounding modes, the alternative definition $\mathbf{u} := b^{-(p-1)}$ is used instead.

we can write

$$|\mathsf{RNE}(x) - x| \le \frac{1}{2} b^{e-(p-1)} \le \mathbf{u} x \tag{2.16}$$

which implies

$$(1 - \mathbf{u})x \le \mathsf{RNE}(x) \le (1 + \mathbf{u})x \tag{2.17}$$

from which the desired claim immediately follows. □

The interpretation of $\mathbf{u}$ is more complicated in the standard floating-point domain $\mathbb{FP}(b, p, e_{\text{min}}, e_{\text{max}})$ due to the presence of subnormal numbers. The leading zeros in the mantissa of a subnormal number reduce its effective precision, causing the distance between x and $\mathsf{RNE}(x)$ to sometimes exceed $\mathbf{u}x$. Instead of the relative error bound

$$|\mathsf{RNE}(x) - x| \le \mathbf{u}|x| \qquad \text{when} \qquad b^{e_{\text{min}}} \le |x| \le b^{e_{\text{max}}}\left(b - \frac{1}{2} b^{-(p-1)}\right) \tag{2.18}$$

which holds when $\mathsf{RNE}(x)$ is normalized, subnormal numbers are instead subject to a weaker *absolute error* bound:

$$|\mathsf{RNE}(x) - x| \le \frac{1}{2} b^{e_{\text{min}}-(p-1)} \qquad \text{when} \qquad |x| < b^{e_{\text{min}}} \tag{2.19}$$

To avoid this complication, it is common for analyses of floating-point algorithms to either assume that every nonzero floating-point number satisfies the hypotheses of Equation (2.18), or when special values are irrelevant, to work in the unbounded floating-point domain $\mathbb{FP}(b, p)$. This assumption is usually benign when working in the binary64 format, as is typical in scientific computing, since the binary64 exponent range ($2^{-1022} \approx 2.2 \times 10^{-308}$ to $2^{1023} \approx 1.8 \times 10^{308}$) is wide enough that overflow and underflow seldom occur. Indeed, the binary64 exponent range is wide enough to cover the dynamic range of the observable universe, which spans sixty orders of magnitude[6] from the quantum scale to the cosmological

[6]The range of any physical quantity, such as length, is bounded by the ratio of the longest theoretically measurable length (the diameter of the observable universe) to the shortest theoretically measurable length (the Planck length). This ratio is $\ell_{\text{universe}}/\ell_{\text{Planck}} \approx 5 \times 10^{61}$. The corresponding ratios for mass and time ($m_{\text{universe}}/m_{\text{Planck}} \approx 6 \times 10^{60}$ and $t_{\text{Hubble}}/t_{\text{Planck}} \approx 8 \times 10^{60}$) are both roughly 10^{60}. Even derived quantities, such as $[\text{Force}] = [\text{Length}] \cdot [\text{Mass}] \cdot [\text{Time}]^{-2} \le 10^{240}$, fall well under the binary64 overflow threshold.

scale, ten times over.

2.5 Error-Free Transformations

Beyond prescribing a universal bound u on the relative magnitude of all rounding errors, in some cases, it is possible to exactly calculate the rounding error of a specific floating-point operation. Remarkably, this can often be done using additional floating-point operations executed in the same precision. Algorithms of this type are called *error-free transformations* and form the building blocks of the techniques developed in this dissertation.

The first error-free transformation, known as the TwoSum algorithm, was discovered by Ole Møller in 1965 [74] and later proven correct by Donald Knuth in 1969 [60]. Given two floating-point numbers $x, y \in \mathbb{FP}(b, p, e_{\min}, e_{\max})$, the TwoSum algorithm computes both their floating-point sum $s \coloneqq x \oplus y$ and, assuming that no overflow occurs, the exact rounding error $e \coloneqq (x + y) - (x \oplus y)$ incurred in that sum. It is somewhat surprising that this task is even possible, since it is not obvious *a priori* that the rounding error e always admits a floating-point representation in the same format as the addends. It is therefore doubly surprising that this is not only possible, but achieved by a remarkably simple and elegant algorithm consisting only of floating-point addition and subtraction.

Algorithm 1: $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$

```
Input: $x, y \in \mathbb{FP}(b, p, e_{\min}, e_{\max})$
Output: $s, e \in \mathbb{FP}(b, p, e_{\min}, e_{\max})$ such that $s = x \oplus y$ and, if all values are finite
        and no overflow occurs, then $e = (x + y) - (x \oplus y)$ exactly.
1 $s \coloneqq x \oplus y$;
2 $x_{\text{eff}} \coloneqq s \ominus y$;
3 $y_{\text{eff}} \coloneqq s \ominus x_{\text{eff}}$;
4 $\delta_x \coloneqq x \ominus x_{\text{eff}}$;
5 $\delta_y \coloneqq y \ominus y_{\text{eff}}$;
6 $e \coloneqq \delta_x \oplus \delta_y$;
7 return $(s, e)$;
```

Intuitively, the values x_{eff} and y_{eff} represent the *effective values* that x and y contribute to the rounded sum $x \oplus y$. Indeed, it can be shown that $x_{\text{eff}} + y_{\text{eff}} = x \oplus y$ holds exactly in the absence of overflow. The rounding error is then reconstructed by measuring the difference between the true and effective values, as illustrated in the following example.

Example 4. Consider $\mathsf{TwoSum}(93.7, 7.54)$ in base $b = 10$ with precision $p = 3$.

1. $s \coloneqq x \oplus y = 93.7 \oplus 7.54 = \mathsf{RNE}(101.24) = 101$

   Here, a rounding error of 0.24 was lost in the floating-point sum.

2. $x_{\text{eff}} \coloneqq s \ominus y = 101 \ominus 7.54 = \mathsf{RNE}(93.46) = 93.5$

   Even though x is truly 93.7, it only managed to contribute 93.5 to the sum.

3. $y_{\text{eff}} \coloneqq s \ominus x_{\text{eff}} = 101 \ominus 93.5 = \mathsf{RNE}(7.5) = 7.5$

   Similarly, $y \coloneqq 7.54$ only managed to contribute 7.5 to the sum. Observe at this step that the effective values 93.5 and 7.5 exactly add up to the rounded sum 101.

4. $\delta_x \coloneqq x \ominus x_{\text{eff}} = 93.7 \ominus 93.5 = \mathsf{RNE}(0.2) = 0.2$

5. $\delta_y \coloneqq y \ominus y_{\text{eff}} = 7.54 \ominus 7.5 = \mathsf{RNE}(0.04) = 0.04$

   Note that δ_x and δ_y can be computed in parallel on a superscalar processor.

6. $e \coloneqq \delta_x \oplus \delta_y = 0.2 \oplus 0.04 = \mathsf{RNE}(0.24) = 0.24$

   By adding the discrepancies between the true and effective values, we exactly reconstruct the rounding error 0.24 that was lost in the initial rounded sum.

Proving the correctness of TwoSum requires lengthy analysis of what Knuth calls "a rather tedious list of special cases," enumerating all of the possible ways in which the mantissas of x and y can overlap each other [60, Section 4.2.2, Theorem A]. A formal computer-verified proof of the correctness of TwoSum is provided in the Flocq library [13].

Note that our formulation of TwoSum (Algorithm 1) only works under Assumption 1, or more generally, when all floating-point operations are rounded to nearest using any tie-breaking rule. More complicated variants of TwoSum are available for use with other rounding-direction attributes [72, Section 4.3.3], but this dissertation only uses Algorithm 1.

The preconditions for the correctness of TwoSum stipulate that all values must be finite and that overflow must not occur. If either of the inputs, x or y, is non-finite, or if the initial sum $s \coloneqq x \oplus y$ overflows, then the error term e computed by TwoSum is obviously not meaningful. In all such cases, e is `NaN`. What is less obvious is that, even when both x and y are finite and the sum $x \oplus y$ does not overflow, an overflow can occur later in the TwoSum algorithm, spuriously causing e to be `NaN` when the exact rounding error $(x + y) - (x \oplus y)$ does have a finite floating-point representation. In base $b = 2$, this failure mode can only occur when one of the addends is the largest representable floating-point number.

Proposition 3 (TwoSum is nearly immune to overflow)**.** *Let $x, y \in \mathbb{FP}(2, p, e_{min}, e_{max})$ be finite floating-point values in a base-2 floating-point format, and let $\Omega \coloneqq 2^{e_{max}}(2 - 2^{-(p-1)})$ denote the largest finite value representable in this format. If $|x| < \Omega$ and $x \oplus y$ is finite, then no overflow occurs in the execution of* TwoSum(x, y).

Proof. See [11, Theorem 6.2]. □

The TwoSum algorithm has a special property when executed in base $b = 2$ or $b = 3$. In these bases, one of the effective values x_{eff} or y_{eff} always coincides with the corresponding exact value, depending on which of the addends x or y is larger in magnitude. Thus, if the relative ordering of $|x|$ and $|y|$ is known in advance, several steps can be omitted from the TwoSum algorithm to produce a faster variant, known appropriately as FastTwoSum [25].

Algorithm 2: $(s, e) \coloneqq$ FastTwoSum(x, y)

Input: $x, y \in \mathbb{FP}(b, p, e_{\text{min}}, e_{\text{max}})$, where $b = 2$ or $b = 3$, such that $|x| \geq |y|$.
Output: $s, e \in \mathbb{FP}(b, p, e_{\text{min}}, e_{\text{max}})$ such that $s = x \oplus y$ and, if all values are finite and no overflow occurs, then $e = (x + y) - (x \oplus y)$ exactly.

```
1 $s \coloneqq x \oplus y$;
2 $y_{\text{eff}} \coloneqq s \ominus x$;
3 $e \coloneqq y \ominus y_{\text{eff}}$;
4 return $(s, e)$;
```

Although FastTwoSum is usually stated with the precondition $|x| \geq |y|$, recent work of Jeannerod and Zimmermann [51] shows that this requirement can be significantly weakened.

Proposition 4 (Generalized preconditions for $\mathsf{FastTwoSum}$)**.** *Let $x, y \in \mathbb{FP}(2, p, e_{min}, e_{max})$ be floating-point values in base $b = 2$ with precision $p \geq 2$. If x and y satisfy any of the following preconditions, then they are valid inputs to* $\mathsf{FastTwoSum}$*, even if $|x| < |y|$.*

- *At least one of x or y is `+0.0` or `-0.0`.*
- *Both x and y are finite, nonzero, and normalized, and their exponents $e_x, e_y \in \mathbb{Z}$ satisfy $e_x + \mathsf{ntz}_x \geq e_y$, where ntz_x denotes the number of trailing zeros in the mantissa of x.*

Proof. See [51, Theorem 1]. □

We will later see that the appearance of the trailing zero count ntz_x in this result is no mere coincidence. Hypotheses on the pattern of nonzero bits appearing in the mantissa of a floating-point number are central to the analysis techniques presented in this dissertation.

Surprisingly, $\mathsf{FastTwoSum}$ is more robust to internal overflow than TwoSum because it omits the problematic operations where spurious overflow can occur.

Proposition 5 ($\mathsf{FastTwoSum}$ is immune to overflow)**.** *Let $x, y \in \mathbb{FP}(2, p, e_{min}, e_{max})$ be finite floating-point values in a base-2 floating-point format, and suppose their exponents $e_x, e_y \in \mathbb{Z}$ satisfy $e_x \geq e_y$. If $x \oplus y$ is finite, then no overflow occurs in the execution of* $\mathsf{FastTwoSum}(x, y)$*.*

Proof. See [11, Theorem 5.1]. □

An important algebraic property of the TwoSum and $\mathsf{FastTwoSum}$ algorithms is that they are *idempotent* operations, i.e., after they are applied to any pair of inputs once, applying them again produces no further change. This property is crucial to establish the uniqueness of a special type of floating-point representation discussed in Section 3.2.

Proposition 6 (TwoSum is idempotent)**.** *Let $x, y \in \mathbb{FP}(b, p, e_{min}, e_{max})$ be finite floating-point values. If no overflow occurs in the computation of $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$, then $\mathsf{TwoSum}(s, e) = (s, e)$.*

Proof. Let $(s', e') \coloneqq \mathsf{TwoSum}(s, e)$. By definition, $s + e = x + y$, so we can write:

$$s' = \mathsf{RNE}(s + e) = \mathsf{RNE}(x + y) = s \tag{2.20}$$

We also have $s' + e' = s + e$ by definition, from which subtracting $s' = s$ yields $e = e'$. □

The equivalence of TwoSum and FastTwoSum implies that Proposition 6 also applies to FastTwoSum when the preconditions of Algorithm 2 or Proposition 4 are satisfied.

Floating-point multiplication also admits an error-free transformation called TwoProd, which is particularly simple to state using the FMA operation.

```
Algorithm 3: $(p, e) \coloneqq \mathsf{TwoProd}(x, y)$
  Input: $x, y \in \mathbb{FP}(b, p, e_{\min}, e_{\max})$
  Output: $p, e \in \mathbb{FP}(b, p, e_{\min}, e_{\max})$ such that $p = x \otimes y$ and, if all values are finite
          and no overflow or underflow occurs, then $e = xy - (x \otimes y)$ exactly.
1 $p \coloneqq x \otimes y$;
2 $e \coloneqq \mathsf{FMA}(x, y, -p)$;
3 return $(p, e)$;
```

The definition of FMA makes the correctness of Algorithm 3 follow trivially from the observation that the product of two p-digit numbers can have at most $2p$ digits. In situations where FMA is not available, an alternative TwoProd algorithm can be stated using only $\oplus$, $\ominus$, and $\otimes$, based on a technique known as *Veltkamp splitting* [95, 96, 25]. This algorithm is more computationally intensive, using five additions, five subtractions, and seven multiplications, compared to Algorithm 3, which uses just one multiplication and one FMA. We refer to the Handbook of Floating-Point Arithmetic [72, Section 4.4.2] for details.

Note that the preconditions for the correctness of TwoProd include a prohibition against underflow, in contrast to TwoSum and FastTwoSum, which only prohibit overflow. Underflow cannot occur in floating-point addition or subtraction because the exact result is always an integer multiple of $b^{e_{\min}-(p-1)}$, the smallest nonzero subnormal number. Multiplication does not have this property. Even when both factors are an integer multiple of $b^{e_{\min}-(p-1)}$, their product can be much smaller than $b^{e_{\min}-(p-1)}$, which introduces underflow as a possible failure mode of the TwoProd algorithm.

Error-free transformations do not exist for floating-point division $\oslash$ and square root $\sqrt[\circ]{\ }$, since rounding errors for these operations do not, in general, admit exact floating-point representations. Indeed, experience with schoolbook arithmetic shows that addition, subtraction and multiplication are finite procedures that always terminate, while long division and square root can produce infinitely many nonzero digits. The FMA operation does admit an error-free transformation [14], but we do not use this algorithm in this dissertation.

The existence of error-free transformations challenges the widely held misconception that floating-point rounding errors are random noise. Although it is common to *model* rounding errors as though they were random in numerical analysis, the determinism of the RNE procedure (Definitions 11 and 12) can be exploited to extend the precision of a floating-point computation beyond the precision p of the underlying floating-point format. This key idea underpins all of the techniques developed in this dissertation.

2.6 Beyond Machine Precision

The largest floating-point format supported by the vast majority of computer processors today is binary64, a status quo that has not changed since the publication of IEEE Standard 754 in 1985. To our knowledge, only IBM POWER9 and POWER10 CPUs include hardware support for binary128 and decimal128; no processors have ever featured hardware support[7] beyond 128-bit (quadruple precision) formats. Intel and AMD x86 CPUs nominally support an 80-bit extended floating-point format, but this is retained only for backward compatibility with the x87 floating-point coprocessor and is not intended for use in modern high-performance applications. Execution of x87 instructions is not pipelined and carries a significant performance penalty for switching in and out of legacy execution mode. Floating-point formats beyond binary64 are also completely absent from GPUs, which provide the bulk of computational horsepower in modern supercomputers. In practice, *machine*

[7]Quadruple-precision floating-point arithmetic is specified as an optional extension in the PA-RISC, SPARC, and RISC-V architectures, but to our knowledge, no hardware implementation of these architectures has ever implemented such an extension.

precision is almost always synonymous with binary64.

This limitation raises a natural question: what can we do when machine precision is insufficient to solve a particular numerical problem?

One obvious answer is to build a machine that supports larger floating-point formats. However, the cost of designing a new high-performance computer processor, estimated to lie in the billions of U.S. dollars [90, 39, 36], makes this avenue economically infeasible for all but the largest commercial-scale applications. Programmable logic devices, such as FPGAs [16, 50, 48, 49], can be cheaper than fully custom hardware but still carry significant development, validation, and deployment costs. Notably, FPGAs are rarely found in commodity computing resources, such as cloud providers and scientific computing clusters. None of the 100 fastest supercomputers on the June 2025 TOP500 list [92] use FPGAs to provide any significant fraction of their computational throughput.

Another approach, favored by numerical analysts, is to improve the *numerical stability* of the underlying numerical algorithm, i.e., redesign the algorithm to reduce its sensitivity to rounding errors. A more numerically stable algorithm can compute a final answer with greater accuracy even when executed in the same floating-point precision. However, techniques for improving numerical stability are typically restricted to specific classes of problems, and developing new techniques for a novel problem class requires deep mathematical expertise. Even within well-studied problem classes, a technique that improves numerical stability in one instance (say, an eigenvalue problem from quantum chemistry) might be useless, or even counterproductive, on another instance of the same problem (such as an eigenvalue problem from geoscience or medical imaging).

A third option, which is the primary focus of this dissertation, is to implement higher-precision floating-point operations in software. This option is particularly attractive because it is low-cost, requiring no specialized hardware, and highly general, applicable to any numerical algorithm with only superficial code changes. However, this approach is generally avoided in demanding high-performance applications because software floating-point emulation has historically been thousands of times slower than native machine-precision arithmetic. To make matters worse, software algorithms for floating-point arithmetic typically involve complex branching patterns that are ill-suited to *data-parallel processors*, such as SIMD CPUs and GPUs, which simultaneously execute each instruction on multiple inputs in parallel. If different inputs take distinct pathways through a branching program, then a data-parallel processor must either explore all branches or revert to serial execution, severely degrading performance. A $10\times$ to $100\times$ slowdown is typical in this situation.

Among software floating-point implementations, it is important to distinguish *extended-precision* libraries, which implement floating-point arithmetic at a specific fixed precision (e.g., binary128 or decimal128), from *arbitrary-precision* or *multiprecision* libraries, which provide generic algorithms for floating-point arithmetic at any precision requested by the user, limited only by the memory capacity of the machine. Although arbitrary-precision libraries are more flexible, generic precision-agnostic algorithms are significantly more complicated to implement and provide fewer avenues for performance optimization. Moreover, scientific and engineering applications rarely[8] need truly arbitrary precision. A modest multiple of machine precision, such as binary128 or binary256, is usually[9] sufficient to correct the numerical deficiencies of computing in machine precision. Thus, extended-precision software libraries are typically preferred in scientific computing.

The conventional approach to extended- or arbitrary-precision floating-point arithmetic in software is to first implement big integers in software, i.e., integers exceeding the capacity of one machine word, typically $2^{32}-1$ or $2^{64}-1$. Floating-point operations are then expressed in terms of big integer operations. The most common big integer implementation strategy, adopted by the GMP [38], MPFR [32], FLINT [40], and Boost.Multiprecision [68] libraries, is to use arrays of machine words to represent digits in base 2^{32} or 2^{64}.

An alternative technique, implemented in the MPRES-BLAS library [47], is to store a big integer N as a sequence of remainders $r_i := N \bmod m_i$ modulo pairwise coprime divisors $m_1, m_2, \ldots, m_n$. Certain arithmetic operations, including addition and multiplication, can

[8]Fields that truly demand arbitrary precision, such as cryptography and computational number theory, treat numbers as data or purely mathematical objects rather than measurements of the physical world.

[9]As previously noted, the scale of the observable universe spans roughly 60 orders of magnitude in all physical dimensions, so a hypothetical "octuple precision" floating-point format (binary256 or decimal256) would be sufficient to store any measurement possible under current models of fundamental physics.

be performed directly on this sequence of remainders, and the Chinese Remainder Theorem allows N to be uniquely reconstructed from $r_1, r_2, \ldots, r_n$ and $m_1, m_2, \ldots, m_n$.

Regardless of which big integer representation is used, implementing floating-point arithmetic on top of an integer abstraction requires sophisticated conditional logic to handle mantissa alignment, rounding, and normalization. Libraries that adopt this approach unavoidably include complex branching code that substantially degrades performance compared to native machine arithmetic.

Another approach that sidesteps big integers entirely is to directly reduce extended-precision floating-point arithmetic to machine-precision floating-point operations. In this framework, a high-precision constant $C \in \mathbb{R}$ is represented as a *floating-point expansion*, i.e., a sequence of successive machine-precision approximations of the following form:

$$\begin{aligned} x_0 &\coloneqq \mathsf{RNE}(C) \\ x_1 &\coloneqq \mathsf{RNE}(C - x_0) \\ x_2 &\coloneqq \mathsf{RNE}(C - x_0 - x_1) \\ &\vdots \\ x_{n-1} &\coloneqq \mathsf{RNE}(C - x_0 - x_1 - \cdots - x_{n-2}) \\ C &\approx x_0 + x_1 + x_2 + \cdots + x_{n-1} \end{aligned} \tag{2.21}$$

Provided that no overflow or underflow occurs in this process, the final n-term expansion $(x_0, \ldots, x_{n-1})$ approximates C with precision no less than np. This approach forms the primary focus of this dissertation and is developed in the following chapter.

Chapter 3

Algorithms

In this chapter, we introduce *floating-point accumulation networks* (*FPANs*), a class of algorithms that perform extended-precision floating-point arithmetic using a linear branch-free sequence of TwoSum operations. Although particular instances of FPAN-like algorithms have been studied in prior work [25, 64, 42, 63, 56, 55, 30], to our knowledge, this dissertation and its supporting papers [100, 101] are the first research works to propose a common theoretical framework that unifies all algorithms of this type. This unification allows us to formulate a computer-aided verification technique that automatically constructs a proof of correctness for a given FPAN (Chapter 4) and an evolutionary search strategy that systematically explores the space of all FPANs to find the fastest possible algorithm for a given task (Chapter 5). When combined, these techniques enable us to discover novel algorithms for extended-precision floating-point arithmetic that are faster than all known algorithms.

3.1 Assumptions

Before we proceed, we recall a critical assumption made in the previous chapter.

Assumption 1. Throughout this dissertation, all floating-point operations are assumed to be rounded to nearest with ties broken to even (i.e., executed with the roundTiesToEven rounding-direction attribute).

We also introduce two additional assumptions that hold throughout the remainder of this dissertation. Both of these assumptions are carefully formulated to accord with common practice in numerical analysis and scientific computing, considerably simplifying our analysis while retaining as much generality as possible in our results.

Assumption 2. From this point onward, we work exclusively in base $b = 2$ and fix some precision $p \geq 2$, which we call the *machine precision*.

Binary floating-point arithmetic is overwhelmingly more common than floating-point arithmetic in any other base, due not only to the intrinsic binary nature of digital circuits, but also the fact that base $b = 2$ minimizes the relative representation error of storing an arbitrary real number [72, Section 2.7.1]. For these reasons, it is completely standard to assume $b = 2$ both inside and outside the floating-point research community [11, 12, 51].

The fixed machine precision p is intended to represent the largest floating-point format supported by a given processor, which is $p = 53$ (binary64) in almost all cases of current practical interest. To develop efficient algorithms for extended-precision floating-point arithmetic, it is preferable to work in the largest native format available. Nonetheless, the results presented in this dissertation apply to binary floating-point arithmetic in any precision $p \geq 2$. We do not consider algorithms that mix multiple floating-point formats.

Assumption 3. All floating-point numbers are henceforth assumed to lie in the unbounded floating-point domain $\mathbb{FP}(2, p)$ (see Definition 10), and all nonzero floating-point numbers are assumed to be normalized. This means we identify `+0.0` with `-0.0`, exclude the special values `+Inf`, `-Inf`, and `NaN`, and ignore overflow, underflow, and subnormal numbers.

The algorithms developed in this dissertation use TwoSum, FastTwoSum, and TwoProd as basic building blocks. These operations are no more susceptible to overflow or underflow than the underlying operations $\oplus$ and $\otimes$ (see Propositions 3 and 5 and the discussion in Section 2.5). The binary64 exponent range ($2^{-1022} \approx 2.2 \times 10^{-308}$ to $2^{1023} \approx 1.8 \times 10^{308}$) is wide enough that overflow and underflow essentially never occur outside exceptional situations with known remedies (e.g., working with log-likelihood instead of direct likelihood).

In particular, this range is wide enough to represent any physically measurable quantity and hence implement any physics simulation (see footnotes in Sections 2.5 and 2.6).

Although `+0.0` and `-0.0` are technically distinct floating-point values with different IEEE encodings, it is standard to treat zero as an unsigned quantity. In fact, IEEE Standard 754 defines the floating-point equality operator to regard `+0.0` and `-0.0` as equal, so that `+0.0 == -0.0` evaluates to `true` in any conforming programming environment.

3.2 Floating-Point Expansions

The algorithms presented in this dissertation work with extended-precision numbers represented as sequences of multiple machine-precision numbers. A representation of this type is called a *floating-point expansion* and uses n machine-precision terms, each with a p-bit mantissa, to collectively represent a single number with at least np bits of precision.

Definition 15 (*floating-point expansion*, *length*, *term*, *real value*, $\mathsf{RealVal}(x_0, \ldots, x_{n-1})$)**.** A *floating-point expansion* of *length* $n \in \mathbb{N}$ is an ordered n-tuple of floating-point numbers $(x_0, \ldots, x_{n-1}) \in \mathbb{FP}(2, p)^n$. The elements of the tuple are called the *terms* of the expansion. The *real value* of a floating-point expansion $(x_0, \ldots, x_{n-1})$ is the exact sum of its terms.

$$\mathsf{RealVal}(x_0, \ldots, x_{n-1}) \coloneqq x_0 + \cdots + x_{n-1} \tag{3.1}$$

An example of a floating-point expansion in precision $p = 6$ is shown in Figure 3.1. This example demonstrates that the terms of a floating-point expansion should be *nonoverlapping* in order to maximize the overall precision of the expansion. In other words, no bit in the binary expansion of the constant $C \in \mathbb{R}$ should be redundantly represented in the mantissa of more than one term. Several inequivalent ways to formalize this notion have been presented in the research literature [72, Section 14.2]. The definition that we adopt is particularly strong and simple to state, but it is often cumbersome to work with in traditional pen-and-paper mathematical proofs.

$$\begin{array}{rl}
\text{high-precision constant} & C = 1\,0\,1\,.\,0\,1\,1\,1\,0\,1\,1\,0\,1\,0\,1\,1\,.\,.\,. \\[1ex]
\mathcal{S}\text{-nonoverlapping} & \left\{\begin{array}{l} x_0 = 1\,0\,1\,.\,0\,0\,0 \\ x_1 = \quad 0\,.\,0\,1\,1\,1\,0\,1\,1 \end{array}\right\} \text{ precision} < 2p \\[1ex]
\mathcal{P}\text{-nonoverlapping} & \left\{\begin{array}{l} x_0 = 1\,0\,1\,.\,0\,1\,1 \\ x_1 = \quad 0\,.\,0\,0\,0\,1\,0\,1\,1\,0\,1 \end{array}\right\} \text{ precision} \geq 2p \\[1ex]
\text{strongly nonoverlapping} & \left\{\begin{array}{l} x_0 = 1\,0\,1\,.\,1\,0\,0 \\ x_1 = \; -0\,.\,0\,0\,0\,0\,1\,0\,0\,1\,0\,1 \end{array}\right\} \text{ precision} \geq 2p+1
\end{array}$$

Figure 3.1: $\mathcal{S}$-nonoverlapping, $\mathcal{P}$-nonoverlapping, and strongly nonoverlapping floating-point expansions of a real number C with terms of precision $p = 6$. Light blue digits represent a shift stored in the exponent and are not explicitly represented in the mantissa. The strongly nonoverlapping expansion rounds x_0 up instead of down, causing x_1 to be negative and the mantissa of x_1 to contains the one's complement of the corresponding bits in C. This allows the sign bit of x_1 to provide an extra implicit bit of precision.

Definition 16 (*strongly dominates*, $\succ$)**.** Let x and y be floating-point numbers. We say that x *strongly dominates* y, denoted by $x \succ y$, if $x \oplus y = x$.

Note that $x \succ y$ implies $|x| \geq |y|$, but $x \succ y$ neither implies, nor is implied by, $x > y$. Every floating-point number, including zero, strongly dominates zero.

Definition 17 (*strongly nonoverlapping*)**.** A floating-point expansion $(x_0, \ldots, x_{n-1})$ is *strongly nonoverlapping* if $x_{k-1} \succ x_k$ for all $k = 1, \ldots, n-1$.

The intuitive meaning of $x \succ y$ is to assert that y is too small to push x even halfway toward either of its closest floating-point neighbors. This captures a strong notion of nonoverlapping. Indeed, if the mantissa of y were to overlap the mantissa of x, then the overlapping bits in y would flip the corresponding bits in x when added together, causing $x \oplus y$ to be different from x. The condition $x \oplus y = x$ prohibits this.

The error-free transformations TwoSum, FastTwoSum, and TwoProd can all be used to create strongly nonoverlapping expansions. The following proposition shows that the sum or product always strongly dominates the rounding error computed by these operations.

Proposition 7 (Result strongly dominates rounding error)**.** *Let x and y be arbitrary floating-point numbers.*

- *If* $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$, *then* $s \succ e$.
- *If* $(p, e) \coloneqq \mathsf{TwoProd}(x, y)$, *then* $p \succ e$.

Proof. The defining property of TwoProd (Algorithm 3) stipulates that its outputs (p, e) are the unique floating-point numbers satisfying $p = \mathsf{RNE}(xy)$ and $p + e = xy$. Hence,

$$p \oplus e = \mathsf{RNE}(p + e) = \mathsf{RNE}(xy) = p \tag{3.2}$$

which, by definition, proves $p \succ e$. The same argument applies to TwoSum by replacing xy with $x + y$ throughout the proof. □

It is natural to ask why floating-point expansions based on machine-precision numbers should be preferable to direct implementation of a larger floating-point format. After all, how can it be more efficient to manipulate n independent floating-point numbers, each with its own sign, exponent, and mantissa, than a single large number? The answer lies in the branching nature of floating-point representation, which is fundamentally defined using case analysis (Definition 9). Every floating-point arithmetic operation involves branching steps, such as mantissa alignment, rounding, and normalization, that significantly degrade performance when implemented in software, particularly on data-parallel processors. Algorithms based on floating-point expansions avoid implementing these steps in software by leveraging the native mantissa alignment, rounding, and normalization circuitry built into a floating-point processor. Computing with floating-point expansions requires more work in an absolute sense, but this work maps more efficiently onto existing hardware.

The performance advantage of floating-point expansions diminishes as the number of terms increases. At a certain length, the increasing cost of the arithmetic workload exceeds the fixed cost of branching, and direct implementation of a larger floating-point format becomes faster. The exact crossover point depends on the underlying computer architecture, occurring at roughly 4–8 terms on modern SIMD CPUs [102].

Long floating-point expansions are also impractical for another reason. Floating-point expansions cannot be made arbitrarily precise because they are subject to the same overflow and underflow thresholds as the underlying native format. A strongly nonoverlapping expansion can only hold $\lceil (e_{\max} - e_{\min} + p)/(p+1) \rceil$ terms before all subsequent terms are guaranteed to underflow. This theoretical limit is 39 terms in $\mathsf{binary64}$ and only 12 terms in $\mathsf{binary32}$. (The appearance of $p+1$ in the denominator is explained in Section 3.2.2.) Moreover, a floating-point expansion can only reach this theoretical limit if its terms span the full exponent range from $e_{\max}$ to $e_{\min}$. The practical limit for the majority of numerical applications, which do not make use of this full range, is considerably smaller.

For these reasons, floating-point expansions are typically used with a small fixed length, such as $n = 2$, 3, or 4 terms [64, 63, 22, 91, 42]. These fixed-length expansions are called *double/triple/quad-word* or *double/triple/quad-double* numbers. The latter names are used when the underlying machine-precision format is $\mathsf{binary64}$, but many algorithms for double/triple/quad-double arithmetic are precision-agnostic and also work in other formats. In particular, the algorithms presented in this dissertation work in any underlying machine precision $p \geq 2$.

3.2.1 Alternative Nonoverlapping Conditions

For completeness, we also state definitions for several alternative nonoverlapping conditions that appear in other work. The $\mathcal{S}$-nonoverlapping and $\mathcal{P}$-nonoverlapping conditions appear in research by Shewchuk [89] and Priest [82], while ulp-nonoverlapping and QD-nonoverlapping are used throughout the algorithms implemented in the CAMPARY [55] and QD [42] software libraries. As its name suggests, strong nonoverlapping is a strictly stronger condition than all of these alternative conditions.

Definition 18 ($\mathcal{S}$-*dominates*, $\succ_{\mathcal{S}}$, $\mathcal{P}$-*dominates*, $\succ_{\mathcal{P}}$, ulp-*dominates*, $\succ_{\mathsf{ulp}}$, QD-*dominates*, $\succ_{\mathsf{QD}}$)**.** Let $x, y \in \mathbb{FP}(2, p)$, and let $e_x, e_y \in \mathbb{Z}$ denote their exponents (undefined if x or y is zero). If x is nonzero, let ntz_x denote the number of trailing zeros in its mantissa.

- We say that x $\mathcal{S}$-*dominates* y, denoted by $x \succ_{\mathcal{S}} y$, if y is zero, or if x and y are both nonzero and $e_x \geq e_y + (p - \mathsf{ntz}_x)$.

- We say that x $\mathcal{P}$-*dominates* y, denoted by $x \succ_{\mathcal{P}} y$, if y is zero, or if x and y are both nonzero and $e_x \geq e_y + p$.
- We say that x ulp-*dominates* y, denoted by $x \succ_{\mathsf{ulp}} y$, if $|y| \leq 2^{e_x-(p-1)}$.
- We say that x QD-*dominates* y, denoted by $x \succ_{\mathsf{QD}} y$, if $|y| \leq 2^{e_x-p}$.

Definition 19 (*$\mathcal{S}$-nonoverlapping*, *$\mathcal{P}$-nonoverlapping*, ulp-*nonoverlapping*, QD-*nonoverlapping*)**.** A floating-point expansion $(x_0, \ldots, x_{n-1})$ is *$\mathcal{S}$-nonoverlapping* if $x_{k-1} \succ_{\mathcal{S}} x_k$ for all $k = 1, \ldots, n-1$. We similarly define *$\mathcal{P}$-nonoverlapping*, ulp-*nonoverlapping*, and QD-*nonoverlapping* expansions with $\succ_{\mathcal{P}}$, $\succ_{\mathsf{ulp}}$, and $\succ_{\mathsf{QD}}$ replacing $\succ_{\mathcal{S}}$ in the preceding definition.

These alternative nonoverlapping conditions are not used to establish the main results of this dissertation. Nonetheless, the verification technique that we develop in Chapter 4 is general enough to subsume all of these conditions as special cases. Thus, our work remains applicable in settings where these alternative conditions may be more appropriate than strong nonoverlapping.

We state the following proposition to precisely characterize the gap in logical strength between strong nonoverlapping and the weaker alternative nonoverlapping conditions. The striking complexity of this logical characterization explains why strong nonoverlapping, despite having a simple definition, is more difficult to work with in mathematical proofs than the alternative nonoverlapping conditions.

Definition 20 (*signed power of two*)**.** A *signed power of two* is a number of the form $\pm 2^k$ for any $k \in \mathbb{Z}$.

Equivalently, a signed power of two is a floating-point number whose mantissa has exactly one nonzero bit. Note that zero is not a signed power of two.

Proposition 8 (Characterization of strong nonoverlapping)**.** *Let x and y be floating-point numbers, and let s_x and s_y denote their sign bits. If x and y are nonzero, let e_x and e_y denote their exponents. Then $x \succ y$ if and only if one of the following conditions holds:*

1. y is zero.

2. x and y are both nonzero and $e_x > e_y + (p+1)$.

3. x and y are both nonzero, $e_x = e_y + (p+1)$, and at least one of the following sub-conditions holds:

   (a) $s_x = s_y$.

   (b) x is not a signed power of two.

   (c) y is a signed power of two.

4. x and y are both nonzero, $e_x = e_y + p$, y is a signed power of two, the trailing bit of the mantissa of x is zero, and at least one of the following sub-conditions holds:

   (a) $s_x = s_y$.

   (b) x is not a signed power of two.

Proof. Case 1 is immediate. To analyze the remaining cases, let x and y be nonzero, and assume without loss of generality that x is positive. Define $h := 2^{e_x-(p-1)}$ and suppose first that x is not a signed power of two. Under this assumption, the immediate floating-point predecessor and successor of x are $x \pm h$. In Cases 2 and 3, we have $|y| < 2^{e_x-(p+1)} = \frac{1}{4}h$ and $|y| < 2^{e_x-p} = \frac{1}{2}h$ respectively, both of which guarantee that $x + y$ lands in the interval of real numbers that round to x. In Case 4, if y is a signed power of two, then $y = \pm\frac{1}{2}h$, which means that $\mathsf{RNE}(x + y) = x$ if and only if the trailing bit of the mantissa of x is zero. If y is not a signed power of two, then $|y| > \frac{1}{2}h$, which implies $\mathsf{RNE}(x + y) \neq x$.

On the other hand, if x is a signed power of two, then its immediate floating-point predecessor is $x - \frac{1}{2}h$ while its immediate successor is $x + h$. The analysis proceeds identically as before if x and y have the same sign, but if x and y have different signs, then the threshold at which $\mathsf{RNE}(x + y) \neq x$ is reduced by a factor of 2. We leave it to the reader to verify that the statements of Cases 3 and 4 correctly account for this reduction. □

Signed powers of two play a special role in the analysis of strong nonoverlapping because they are the critical step sizes that can push a floating-point number precisely halfway to its neighbor. When this occurs, we must explicitly invoke the RNE tie-breaking rule (Definition 11) to determine whether x strongly dominates y. This can create a tangled web of logical conditions, splitting the analysis of a numerical algorithm into a myriad of cases depending on the mantissa parity of each floating-point number involved.

To complete our discussion of alternative nonoverlapping conditions, we make the following observations to compare their logical strength. These claims are stated without proof and are not used in the remainder of this dissertation; they merely serve to situate our results about strong nonoverlapping in the context of the floating-point research literature.

- Strong nonoverlapping implies QD-nonoverlapping. Indeed, the gap between strong nonoverlapping and QD-nonoverlapping is precisely that QD-nonoverlapping ignores the parity of the mantissa to avoid explicit analysis of the RNE tie-breaking rule. Outside these tie-breaking cases, strong nonoverlapping and QD-nonoverlapping are otherwise synonymous.

- QD-nonoverlapping implies $\mathcal{P}$-nonoverlapping.

- $\mathcal{P}$-nonoverlapping most directly captures the intuitive meaning of nonoverlapping. i.e., each mantissa bit should represent a distinct place value. However, it is difficult to maintain $\mathcal{P}$-nonoverlapping as an algorithmic invariant because many floating-point operations can create one bit of overlap.

- $\mathcal{P}$-nonoverlapping implies ulp-nonoverlapping. The gap between these conditions is that $x \succ_{\mathsf{ulp}} y$ allows y to overlap the trailing bit of x when y is a signed power of two.

- $\mathcal{P}$-nonoverlapping also implies $\mathcal{S}$-nonoverlapping.

- $\mathcal{S}$-nonoverlapping and ulp-nonoverlapping are incomparable (i.e., neither logically implies the other).

- Despite being one of the weakest nonoverlapping conditions, $\mathcal{S}$-nonoverlapping is useful in practice because there are particularly simple algorithms that produce $\mathcal{S}$-nonoverlapping floating-point expansions [72, Section 14.2].

3.2.2 Uniqueness and Renormalization

Just as a single floating-point number can have multiple representations, it is possible for distinct floating-point expansions to share the same real value. For example, any permutation of the terms of a floating-point expansion $(x_0, \ldots, x_{n-1})$ is another expansion with the same real value. Similarly, the real value remains unchanged when any two terms (x_i, x_j) are replaced by $\mathsf{TwoSum}(x_i, x_j)$. It is natural to ask whether a condition analogous to normalization can be imposed to eliminate this ambiguity.

Strong nonoverlapping is a clear candidate for this criterion. It requires the terms of a floating-point expansion to be sorted in magnitude from largest to smallest, eliminating permutation ambiguity, and ensures that the terms remain unchanged by the application of TwoSum. Indeed, it is possible to bring any floating-point expansion into strongly nonoverlapping form by repeatedly applying TwoSum to its terms until no overlap remains, a procedure known as *renormalization* [72, Section 14.2.1]. Intuitively, each application of TwoSum redistributes mantissa bits between a given pair of terms (x_i, x_{i+1}) to clear away overlapping bits. This redistribution may create new overlap in the adjacent pairs (x_{i-1}, x_i) and (x_{i+1}, x_{i+2}), but it can be shown that this process reaches a fixed point in a finite number of applications of TwoSum [12], producing a strongly nonoverlapping expansion.

Surprisingly, it is possible for two distinct floating-point expansions, both fully renormalized, to share the same real value. In other words, strong nonoverlapping is not strong enough to guarantee uniqueness of representation. To understand why this is the case, we first introduce a procedure that truly guarantees uniqueness by algorithmically constructing a distinguished floating-point expansion for any real number.

Definition 21 (*canonical floating-point expansion*)**.** Let $C \in \mathbb{R}$ and $n \in \mathbb{N}$. The *canonical floating-point expansion* of C with length n is the floating-point expansion $(x_0, \ldots, x_{n-1})$

computed as follows:

$$\begin{aligned} x_0 &\coloneqq \mathsf{RNE}(C) \\ x_1 &\coloneqq \mathsf{RNE}(C - x_0) \\ x_2 &\coloneqq \mathsf{RNE}(C - x_0 - x_1) \\ &\vdots \\ x_{n-1} &\coloneqq \mathsf{RNE}(C - x_0 - x_1 - \cdots - x_{n-2}) \end{aligned} \tag{3.3}$$

The canonical floating-point expansion $(x_0, \ldots, x_{n-1})$ defined above is the maximally accurate floating-point expansion in the sense that, for each $k = 1, \ldots, n$, the k-term approximation error $|C - x_0 - \cdots - x_{k-1}|$ is minimized. Any other floating-point expansion either differs only in RNE tie-breaking or has a strictly larger k-term approximation error for some value of k.

The relative k-term approximation error $|C - x_0 - \cdots - x_{k-1}|/|C|$ of the canonical floating-point expansion is at most $2^{-np-(n-1)}$. The appearance of np in the exponent is unsurprising since each of the n terms has a p-bit mantissa, but this does not account for the unexpected appearance of $n - 1$ additional bits. These additional bits arise from the sign bit of each subsequent term providing an extra *implicit* bit of precision between terms, as shown in Figure 3.1. This extra implicit precision occurs only when rounding to nearest, which provides a significant advantage over other rounding strategies.

With the canonical floating-point expansion defined, we are now prepared to understand why a strongly nonoverlapping expansion can fail to be canonical. The issue, perhaps unsurprisingly, arises from tie-breaking. In particular, two consecutive rounding midpoints can occur in an expansion with three or more terms. When considered together, these consecutive midpoints imply that the expansion, as a whole, is non-canonical. However, the TwoSum algorithm, which only operates on two terms at a time, is unable to correct this.

Example 5. The floating-point expansion $(1, 2^{-p}, 2^{-2p})$ is strongly nonoverlapping. To see this, observe that $1 + 2^{-p}$ lies exactly in the middle of two neighboring floating-point numbers, 1 and $1 + 2^{-(p-1)}$. The rounding function RNE prefers the former because the final entry of its mantissa $(1, 0, \ldots, 0)$ is even, so $1 \oplus 2^{-p} = \mathsf{RNE}(1 + 2^{-p}) = 1$. An analogous calculation shows that $2^{-p} \oplus 2^{-2p} = 2^{-p}$.

However, $(1, 2^{-p}, 2^{-2p})$ is not canonical because $\mathsf{RNE}(1 + 2^{-p} + 2^{-2p}) \neq 1$. This number slightly exceeds the midpoint $1 + 2^{-p}$, so it rounds up to the subsequent floating-point number $1 + 2^{-(p-1)}$. Therefore, the canonical floating-point expansion of $1 + 2^{-p} + 2^{-2p}$ is $(1 + 2^{-(p-1)}, -2^{-p} + 2^{-2p}, 0)$, which has fewer nonzero terms than $(1, 2^{-p}, 2^{-2p})$.

Although this chain-of-midpoints phenomenon demonstrates that strongly nonoverlapping expansions are not necessarily unique, in practice, it is exceedingly rare for such a chain of midpoints to arise without being deliberately crafted using pathological input data. Thus, in non-adversarial settings, it is typically safe to assume that any strongly nonoverlapping floating-point expansion is the canonical expansion of its real value.

3.3 Floating-Point Accumulation Networks

As their name suggests, floating-point accumulation networks (FPANs) perform the task of *accumulation*, i.e., extended-precision summation of multiple floating-point numbers. Although this task may seem modest, we will see that accumulation encapsulates the essential difficulties of computation with floating-point expansions. Once accumulation is solved, all remaining arithmetic operations, including addition, subtraction, multiplication, division, and square root, follow in a straightforward fashion.

To understand why accumulation is a nontrivial task, consider the problem of adding two floating-point expansions, $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1})$. We want to compute a floating-point expansion $(z_0, \ldots, z_{n-1})$ such that $\mathsf{RealVal}(z_0, \ldots, z_{n-1})$ is as close as possible to the exact sum $\mathsf{RealVal}(x_0, \ldots, x_{n-1}) + \mathsf{RealVal}(y_0, \ldots, y_{n-1})$. One naïve approach is to add the inputs term-by-term:

$$\begin{aligned} z_0 &\coloneqq x_0 \oplus y_0 \\ &\vdots \\ z_{n-1} &\coloneqq x_{n-1} \oplus y_{n-1} \end{aligned} \tag{3.4}$$

This strategy, while intuitively appealing, is completely incorrect, producing a result that is no more accurate than the machine-precision sum $x_0 \oplus y_0$. There are two issues at play:

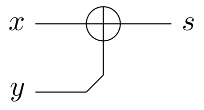
- Each of the floating-point sums $x_i \oplus y_i$ is rounded, and the rounding error $(x_i + y_i) - (x_i \oplus y_i)$ must be accounted for when computing the subsequent term $x_{i+1} \oplus y_{i+1}$.
- If the result of $x_i \oplus y_i$ is smaller in magnitude than x_i or y_i, then it may overlap the result of $x_{i+1} \oplus y_{i+1}$. Mantissa bits must then be redistributed between these two terms in order to maintain the nonoverlapping invariant.

Both of these issues can be resolved by using the TwoSum and FastTwoSum operations to compute and propagate rounding errors and to clear overlapping mantissa bits between adjacent terms (Proposition 7). These capabilities make error-free transformations fundamental building blocks for computation with floating-point expansions.

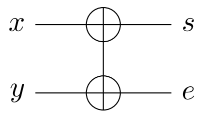
However, even with these powerful tools in hand, the development of *branch-free* algorithms for floating-point expansion arithmetic remains challenging. To construct such an algorithm, we must devise a single, fixed sequence of error-free transformations that correctly propagates rounding errors while removing overlapping bits between all adjacent terms. It is not difficult to find such a sequence for a particular input, but it is very difficult to construct a single sequence that does the job for all possible inputs. Designing sequences of error-free transformations with correct error propagation and nonoverlapping semantics is a remarkably difficult problem; the literature on this subject is punctuated by refutations and corrections [54, 73]. Some general constructions are known, but these algorithms are far from optimal, particularly when the number of inputs is small [21, 72]. This fundamental challenge motivates the study of floating-point accumulation networks.

We formally define floating-point accumulation networks as a class of branch-free algorithms using a graphical notation inspired by sorting networks [61].

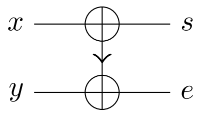
Definition 22 (*floating-point accumulation network*, *FPAN*, *wire*, *gate*, *discarded*)**.** A *floating-point accumulation network* (*FPAN*) is a diagram consisting of horizontal *wires* and vertical *gates*. Each gate connects exactly two input wires to one or two output wires. The input wires are drawn to the upper-left and lower-left of the gate, and the output wires are drawn to its upper-right and (if two output wires are present) its lower-right. If there is only one output wire, then we say that the lower wire is *discarded*. We define three types of gates corresponding to floating-point addition, TwoSum, and FastTwoSum, respectively:



$$s \coloneqq x \oplus y \tag{3.5}$$



$$(s, e) \coloneqq \mathsf{TwoSum}(x, y) \tag{3.6}$$



$$(s, e) \coloneqq \mathsf{FastTwoSum}(x, y) \tag{3.7}$$

The downward-pointing arrowhead on the FastTwoSum gate is intended to serve as a mnemonic reminder that the top input, if nonzero, must be larger in magnitude than the bottom input. Similarly, the larger-magnitude output is always placed on top.

An FPAN with n wires, of which k are discarded, represents the following algorithm with n floating-point inputs and $n - k$ floating-point outputs. Each input value $(x_0, \ldots, x_{n-1})$ enters on the left-hand side of each wire, ordered top-to-bottom unless otherwise specified by explicit labels. The values flow left-to-right along the wires, and whenever two values (x_i, x_j) encounter a gate, they are updated as specified by Equations (3.5), (3.6), and (3.7). After all gates have been executed, all values on non-discarded wires are returned in top-to-bottom order. To illustrate this definition, Figure 3.2 presents equivalent pseudocode and network diagram representations of Dekker's add2 algorithm, the first algorithm ever proposed for double-double addition [25].

The intended operation of an FPAN is to compute a strongly nonoverlapping floating-point expansion of the exact sum of its input values. By the defining property of TwoSum (Algorithm 1), this value is invariant under the application of a TwoSum gate to any two wires; it is only ever changed by discarding a wire. Therefore, an FPAN is correct if and only if the following correctness conditions hold:

```
Algorithm 4: add2($(x_0, x_1), (y_0, y_1)$)
  Input: floating-point expansions
         $(x_0, x_1)$ and $(y_0, y_1)$.
  Output: floating-point expansion
         $(z_0, z_1)$ for $x + y$.
1 $(s_0, s_1) \coloneqq \mathsf{TwoSum}(x_0, y_0)$;
2 $t \coloneqq x_1 \oplus y_1$;
3 $u \coloneqq s_1 \oplus t$;
4 $(z_0, z_1) \coloneqq \mathsf{FastTwoSum}(s_0, u)$;
5 return $(z_0, z_1)$;
```

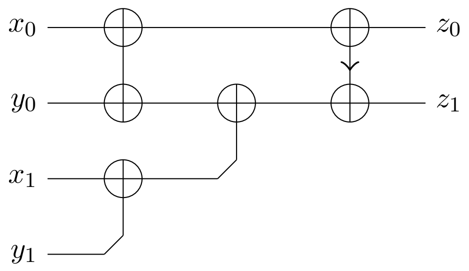


Figure 3.2: Pseudocode and FPAN representations of Dekker's add2 algorithm. Note that the intermediate variables s_0, s_1, t, u are anonymous in the FPAN representation, implicitly represented by the wire segments running between TwoSum gates.

- The inputs of every FastTwoSum gate must satisfy the hypotheses of Proposition 4.
- The output values must be strongly nonoverlapping for all possible input values.
- The rounding errors discarded by addition gates must be small relative to the leading output term z_0. In particular, an FPAN has q-bit precision if the absolute value of the sum of all discarded values is at most $2^{-q}|z_0|$.

Verifying these properties requires extensive case analysis of all possible rounding error patterns that can be created by a given sequence of sum, TwoSum, and FastTwoSum operations. This combinatorial explosion of cases is challenging and tedious to analyze by hand; we refer the reader to the proof of [54, Theorem 3.1] for an example of this phenomenon.

Dekker's add2 algorithm (Algorithm 4) is notable for having an extremely weak error bound that violates the third correctness condition. Assuming $\mathcal{P}$-nonoverlapping inputs (x_0, x_1) and (y_0, y_1), Dekker proved [25] that the relative difference between the sum (z_0, z_1) computed by add2 and the true sum $x_0 + x_1 + y_0 + x_1$ is bounded above by:

$$\frac{|(z_0 + z_1) - (x_0 + x_1 + y_0 + y_1)|}{|x_0 + x_1 + y_0 + y_1|} \leq 4\mathbf{u}^2 \frac{|x_0 + x_1| + |y_0 + y_1|}{|x_0 + x_1 + y_0 + y_1|} \tag{3.8}$$

Although this relative error bound is reasonably tight when (x_0, x_1) and (y_0, y_1) have the same sign, it can be extremely loose when (x_0, x_1) and (y_0, y_1) have different signs, which can cause $|x_0 + x_1| + |y_0 + y_1|$ to be orders of magnitude larger than $|x_0 + x_1 + y_0 + y_1|$. Joldes, Muller, and Popescu [54] identified example inputs for which add2 computes sums with 100% relative error, i.e., zero accurate bits compared to the true value of $x_0 + x_1 + y_0 + y_1$.

This observation highlights the surprising difficulty of computing accurate floating-point sums, even for as few as four inputs. At first glance, the network diagram shown in Figure 3.2 may not appear to have any obvious deficiencies. Indeed, when interpreted as a sorting network, this diagram gives a correct algorithm for partially sorting four inputs satisfying the preconditions $x_0 > x_1$ and $y_0 > y_1$. However, there are two fundamental differences that make floating-point accumulation harder than sorting. First, the outputs of an FPAN not only need to be sorted by magnitude, but also require a degree of mutual separation in order to be strongly nonoverlapping. Second, unlike a comparator which merely reorders its inputs, a TwoSum gate actually modifies its inputs, potentially introducing new overlap and ordering issues with every operation.

Kahan–Babuška–Neumaier (KBN) summation is another example of an FPAN-like algorithm proposed in prior work [58, 2, 76]. This algorithm uses FastTwoSum to compute floating-point sums with a running compensation term to improve the accuracy of the final result. This technique is frequently used in floating-point programs and is implemented in both the Python and Julia standard libraries. In particular, Python's built-in `sum()` function uses KBN summation when given floating-point inputs [83]. In our graphical FPAN notation, the KBN algorithm has a double staircase structure illustrated in Figure 3.3. The first staircase computes the naïve floating-point sum of the inputs, while the second staircase computes the running compensation term used to correct the naïve sum.

In addition to quantifying the number of bits of precision, an FPAN is also parameterized by its *size* (its total number of gates) and its *depth* (the number of gates encountered on the longest directed path from an input node to an output node). To maximize computational efficiency, it is desirable to minimize size and depth while maximizing precision.

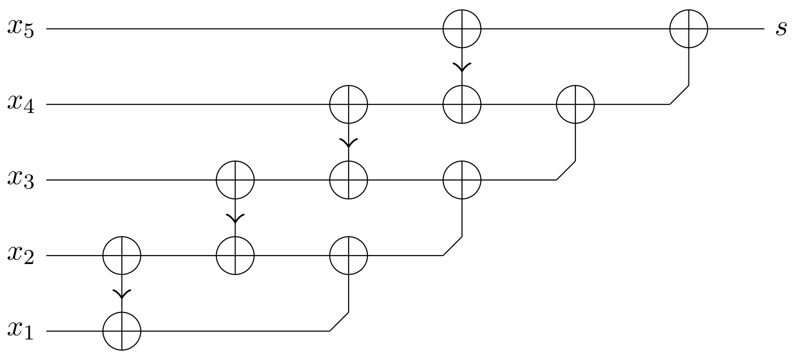


Figure 3.3: FPAN diagram for Kahan–Babuška–Neumaier summation applied to five inputs. This double staircase accumulation pattern generalizes to any number of inputs.

3.4 Arithmetic with Expansions

With FPANs formally defined, we are now prepared to state branch-free algorithms for addition, subtraction, multiplication, division, and square root of floating-point expansions. These algorithms are presented as abstract procedure templates that call FPANs as black-box subroutines to perform extended-precision accumulation. The actual FPANs that we plug into these templates to produce concrete implementable algorithms are produced by a stochastic program synthesis technique and are shown in Chapter 5.

Algorithms for floating-point expansion arithmetic proposed in prior work typically consist of two steps: an arithmetic step that produces an overlapping expansion, followed by a renormalization step that repeatedly applies TwoSum operations to produce a nonoverlapping expansion [42, 55, 21, 56]. This renormalization step is usually expensive and involves branching and/or looping to identify all pairs of potentially overlapping terms to which TwoSum must be applied. In contrast, the algorithms presented in this dissertation eliminate the need for an separate renormalization step by using FPANs to simultaneously perform arithmetic and renormalization in a single branch-free step.

Addition and subtraction are the most straightforward operations to implement using FPANs, which naturally compute extended-precision sums. Given two floating-point expansions, $(x_0, \dots, x_{n-1})$ and $(y_0, \dots, y_{n-1})$, we construct an FPAN with $2n$ interleaved inputs $(x_0, \pm y_0, \dots, x_{n-1}, \pm y_{n-1})$ and n strongly nonoverlapping outputs, with $+$ signs chosen for addition and $-$ signs chosen for subtraction. We assume the input expansions $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1})$ to be strongly nonoverlapping, which makes this task considerably easier than the more general problem of accumulating $2n$ arbitrary inputs.

Our strategy for multiplication with FPANs is based on the distributive property. Recall that the real value represented by the floating-point expansion $(x_0, \ldots, x_{n-1})$ is the exact sum $x \coloneqq x_0 + \cdots + x_{n-1}$ of its terms. Hence, the exact product of $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1})$ can be written as a sum of n^2 pairwise products:

$$xy = x_0 y_0 + x_0 y_1 + x_1 y_0 + \cdots + x_{n-1} y_{n-1} \tag{3.9}$$

Each of these pairwise products can be exactly computed by the $\mathsf{TwoProd}$ algorithm. Thus, by computing all pairwise error-free products $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$, we can write the product xy as the exact sum of the $2n^2$ machine-precision floating-point numbers $p_{0,0}, p_{0,1}, p_{1,0}, \ldots, p_{n-1,n-1}$ and $e_{0,0}, e_{0,1}, e_{1,0}, \ldots, e_{n-1,n-1}$. This strategy splits multiplication of floating-point expansions into two phases:

- an initial expansion phase that executes n^2 $\mathsf{TwoProd}$ operations; followed by
- an accumulation phase that executes an FPAN with $2n^2$ inputs.

We can significantly reduce the number of operations in both phases by observing that certain product terms can always be safely discarded when the inputs are strongly nonoverlapping. Let e_x and e_y denote the exponents of x_0 and y_0, respectively. To compute an n-term floating-point expansion of the exact product $z \coloneqq xy$, which is at least $2^{e_x+e_y}$, we can safely ignore any term whose exponent falls below $e_x + e_y - n(p+1)$. Strong nonoverlapping implies that the exponent of x_i is at most $e_x - i(p+1)$, and similarly, the exponent of y_j is at most $e_y - j(p+1)$. Hence, the exponents of $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$ are at most $e_x + e_y + 1 - (i+j)(p+1)$ and $e_x + e_y + 1 - (i+j+1)(p+1)$, respectively. This means we can safely ignore $p_{i,j}$ whenever $i + j \geq n$ and $e_{i,j}$ whenever $i + j + 1 \geq n$, simplifying the expansion phase from n^2 $\mathsf{TwoProd}$ operations to $n(n-1)/2$ $\mathsf{TwoProd}$ operations and n machine-precision floating-point products. This also reduces the number of FPAN inputs in the accumulation phase from $2n^2$ to n^2.

With branch-free addition and multiplication algorithms in hand, division and square root can be implemented in a branch-free fashion using classical algorithms based on division-free Newton–Raphson iteration. This approach is well-known in the computer arithmetic literature, so we only state the core ideas in this dissertation for completeness, referring to [59] for further details.

The basic principle of these algorithms is to apply the Newton–Raphson iterative root-finding method, defined by the recurrence formula

$$x_{n+1} \coloneqq x_n - \frac{f(x_n)}{f'(x_n)} \tag{3.10}$$

to the function

$$f(x) = \frac{1}{x} - a \tag{3.11}$$

which has a unique root at $x = 1/a$ for nonzero a, or the function

$$f(x) = \frac{1}{x^2} - a \tag{3.12}$$

which has two roots at $x = \pm 1/\sqrt{a}$. These functions are designed to compute inverses and inverse square roots, respectively. Substituting these functions into Equation (3.10) produces the iterative formula

$$x_{n+1} = x_n + x_n(1 - a x_n) \tag{3.13}$$

for computing inverses, and

$$x_{n+1} = x_n + \frac{1}{2} x_n (1 - a x_n^2) \tag{3.14}$$

for computing inverse square roots. Note that multiplication by $1/2$ is an exact operation that can be applied termwise in binary floating-point arithmetic. By taking the initial guess x_0 to be the machine-precision approximation $1 \oslash a$ or $1 \oslash \sqrt[\circ]{a}$, respectively, these iterative formulas allow rapid approximation of $1/a$ or $1/\sqrt{a}$ since the number of correct bits roughly doubles on every iteration. Finally, once $1/a$ or $1/\sqrt{a}$ is computed to the desired accuracy, we can obtain the quotient b/a by multiplying $1/a$ by b, or the square root $\sqrt{a}$ by multiplying $1/\sqrt{a}$ by a.

This technique can be optimized for use with floating-point expansions by reducing the number of terms used to represent the first few iterates. The initial approximation x_0 is only accurate to machine precision, so there is no need to store more than one term at this stage. The number of accurate bits doubles with each subsequent iteration, so the next iterate x_1 can be represented using a two-term expansion, then x_2 with a four-term expansion, and so on until the desired final precision is reached. The Karp–Markstein optimization [59] can also be applied to fuse the final Newton iteration with the multiplication of $1/a$ by b or $1/\sqrt{a}$ by a, eliminating several costly full-precision multiplication calls.

Chapter 4

Verification

In this chapter, we develop a computer-aided verification technique for the FPAN correctness conditions stated in Section 3.3. These conditions are remarkably difficult to prove because they require reasoning over the space of all possible inputs to a given FPAN, which usually consist of terms from multiple floating-point expansions. Even if these input expansions are assumed to be strongly nonoverlapping (as is the case in all of our algorithms), there are an exponential number of ways that two strongly nonoverlapping length-n expansions can interlace with each other, as shown in Figure 4.1. Each interlacing creates a different pattern of rounding error propagation through the gates of an FPAN, creating an exponential number of cases that each require separate analysis. To make matters worse, the preconditions of `FastTwoSum` (Proposition 4) and strong nonoverlapping (Proposition 8) also introduce their own case splits, producing a combinatorial explosion in the number of cases that must be considered to prove the FPAN correctness conditions.

Unfortunately, this explosion of cases makes the construction and analysis of FPANs tedious and error-prone. On several occasions, subtly flawed algorithms and incorrect error bounds have been published in the floating-point research literature, going unnoticed for many years. For example, after it was realized that Dekker's `add2` algorithm (Algorithm 4) has a catastrophically weak error bound for inputs with different signs, Li et al. proposed an improved algorithm for double-double addition, called `ddadd`, for implementation in the

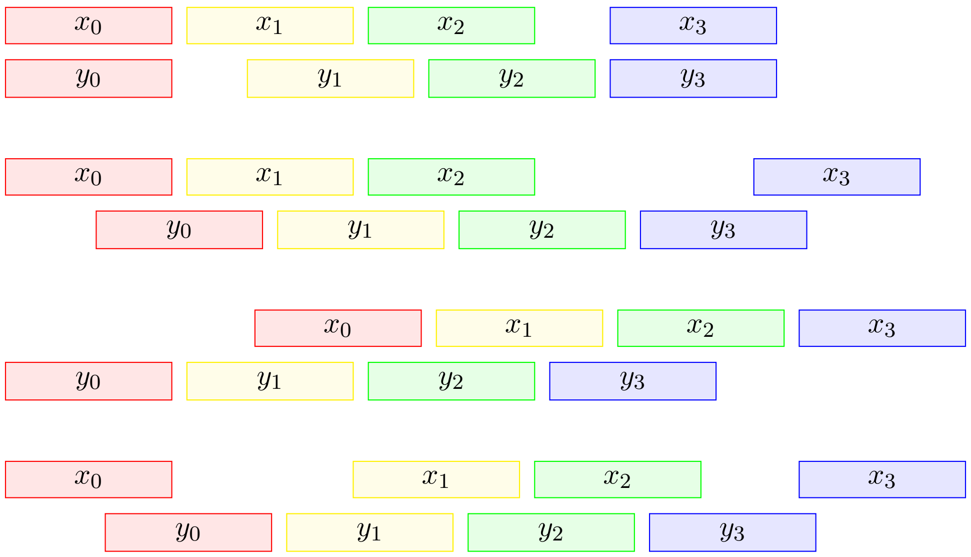


Figure 4.1: Schematic representation of several representative interlacing patterns that two floating-point expansions of length four can exhibit.

XBLAS extended-precision linear algebra library [64]. An FPAN diagram for the ddadd algorithm, which was also adopted by other math libraries [22, 91], is shown in Figure 4.2.

In their 2002 paper [64], Li et al. claimed without proof that the relative error of a sum computed by ddadd can be no larger than $2\mathbf{u}^2$, assuming that the input expansions (x_0, x_1) and (y_0, y_1) are both strongly nonoverlapping. Fifteen years later, in 2017, Joldes, Muller, and Popescu [54] refuted this claim by explicitly constructing strongly nonoverlapping inputs for which ddadd computes a sum with relative error $2.25\mathbf{u}^2$. They conjectured that $2.25\mathbf{u}^2$ was the optimal relative error bound for ddadd, but five years later, in 2022, Muller and Rideau [73] found a stronger counterexample with relative error $3\mathbf{u}^2$. This is now known to be the truly optimal error bound for ddadd, as shown by both a lengthy pen-and-paper mathematical proof [54] and a computer-checked formal proof in Rocq [73].

To be clear, the gap between the mistaken error bound $2\mathbf{u}^2$ and the true error bound $3\mathbf{u}^2$ is in no way catastrophic, nor does it invalidate the usefulness of the XBLAS library. The discovery of ddadd remains an impressive achievement that we in no way wish to impugn. Our purpose in presenting this case study is to illustrate that the analysis of FPANs is so difficult and error-prone that even world experts in numerical analysis, including ACM

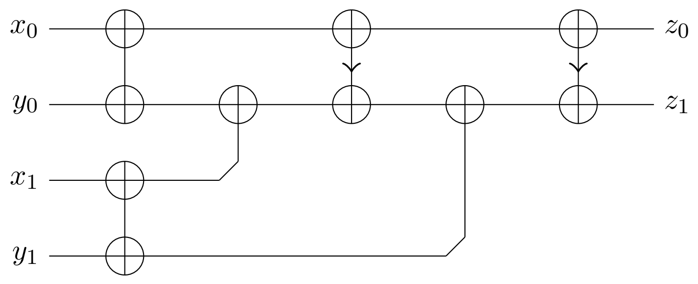


Figure 4.2: FPAN representation of the `ddadd` algorithm due to Li et al. [64].

Fellows, SIAM Fellows, and winners of the Gordon Bell Prize and Turing Award [64], can make mistakes—even in the simplest case of adding length-2 expansions!

This immense difficulty motivates us to consider computer-assisted methods for constructing and analyzing FPANs. In principle, these tasks should be well-suited to computer automation. Formal reasoning about FPANs requires identifying and managing a large number of cases, each of which involves straightforward algebraic manipulation of linear inequalities. This pattern of branching exploration interspersed with routine mechanical verification is precisely the type of workload that automated reasoning tools, such as automatic theorem provers and SMT solvers, should be best equipped to handle. However, current tools have limited capacity for reasoning about floating-point operations and error-free transformations, severely limiting their applicability to FPANs.

Existing techniques for automated floating-point verification fall into two broad classes. The first class is characterized by an approach that we call *projection from real arithmetic*, implemented in tools such as dReal [35] and Colibri2 [57]. Techniques of this type prove a property P of a floating-point program in two steps. First, they reformulate P by treating each floating-point variable as if it were an exact real number to obtain a modified property $P[\mathbb{R}]$, which is proven using standard computer algebra techniques, such as cylindrical algebraic decomposition. Then, they check whether the statement $P[\mathbb{R}]$ is sufficiently robust to small perturbations to remain true when a small rounding error is introduced into each arithmetic operation. Interval/ball arithmetic and polyhedral/relational domains are examples of methods used to perform these robustness tests [18, 71, 17, 84].

Techniques based on projection from real arithmetic are fundamentally incapable of reasoning about error-free transformations, including TwoSum, FastTwoSum, and TwoProd. In exact arithmetic, TwoSum is reduced to the trivial operation $\mathsf{TwoSum}(x, y) = (x + y, 0)$, and no statement about the trivialized TwoSum operation in this form remains true when rounding errors are reintroduced. The computation of rounding errors performed by error-free transformations is a phenomenon exclusive to finite-precision arithmetic that has no semantically equivalent analogue in the exact real domain.

The second class of techniques is called *bit-blasting*, implemented in tools including Z3 [24], CVC5 [8], MathSAT 5 [20], and Bitwuzla [77]. Rather than considering floating-point variables as approximate real numbers, bit-blasting treats each floating-point variable as an IEEE-encoded bit vector (Definition 7) and models each arithmetic operation as a Boolean circuit. Any property P of a floating-point program can then be written as a Boolean formula, which can be checked using a standard Boolean satisfiability (SAT) solver.

While bit-blasting is capable of expressing error-free transformations, it is far too expensive to apply to FPANs of nontrivial size. The Boolean circuits that implement floating-point addition involve a large number of internal variables that are necessary to implement mantissa alignment, rounding, and normalization. To make matters worse, these operations become deeply nested as TwoSum gates are chained together. As shown by our benchmarks in Section 4.5, solving a satisfiability problem of this complexity is far our of reach of even the fastest SAT solvers available today. Bit-blasting also exhibits exponentially increasing costs as the underlying machine precision p increases, which is especially problematic because FPANs are typically used with large floating-point formats, such as binary64.

4.1 The SELTZO Abstraction

An automatic verification technique for the FPAN correctness conditions should be able to deduce the general shape of a floating-point sum and its rounding error without getting bogged down by exactly computing every last bit. To achieve this goal, we introduce a novel technique for coarse modeling of floating-point numbers called the *SELTZO abstraction*.

Definition 23 (*SELTZO abstraction*)**.** Let x be a nonzero floating-point number. The *sign-exponent leading-trailing zeros-ones (SELTZO) abstraction* of x is the ordered 6-tuple $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ consisting of:

1. the sign bit $s_x \in \{0, 1\}$ and exponent $e_x \in \mathbb{Z}$ of x;
2. the counts $\mathsf{nlz}_x, \mathsf{nlo}_x \in \mathbb{N}$ of leading zeros and ones, respectively, in the mantissa of x, ignoring the implicit leading bit; and
3. the counts $\mathsf{ntz}_x, \mathsf{nto}_x \in \mathbb{N}$ of trailing zeros and ones, respectively, in the mantissa of x, ignoring the implicit leading bit.

Example 6. The SELTZO abstraction of $-2^7 \times 1.0010011111_2$ is $(1, 7, 2, 0, 0, 5)$, and the SELTZO abstraction of $+2^{-2} \times 1.1111111111_2$ is $(0, -2, 0, 10, 0, 10)$. Recall that the implicit leading bit is ignored when computing nlz_x, nlo_x, and nto_x.

The SELTZO abstraction is designed to allow the FPAN correctness conditions stated in Section 3.3 to be expressed as linear equations and inequalities in the SELTZO variables $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ and the machine precision p. This is important because the theory of *quantifier-free linear integer arithmetic* (*QF-LIA*), also known as *Presburger arithmetic*, is a decidable theory. This means that an algorithm can determine whether any logical combination of linear equations and inequalities can be satisfied over the integers. This algorithm has been implemented in many SMT solvers, including Z3 [24], CVC5 [8], MathSAT 5 [20], and Yices 2 [26], and is surprisingly efficient in practice despite having doubly exponential worst-case time complexity.

Our strategy for automatically verifying the FPAN correctness conditions is to formulate the existence of a counterexample as a satisfiability problem in QF-LIA. Here, a counterexample is a sequence of floating-point inputs that produces a failure of the FastTwoSum preconditions, outputs that fail to be strongly nonoverlapping, or a discarded error term whose magnitude exceeds some specified threshold. The variables of this satisfiability problem are not concrete floating-point numbers, but rather SELTZO tuples $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ for each input, output, and intermediate floating-point number x flowing through the wires of a given FPAN. This means that our QF-LIA problem *overapproximates* the true semantics of the underlying FPAN on concrete floating-point numbers.

This strategy of overapproximation in the SELTZO domain yields a *one-sided* decision procedure for the FPAN correctness conditions. If the QF-LIA statement S that expresses the existence of a SELTZO counterexample is unsatisfiable, then we can conclude that no concrete floating-point counterexample exists, since a concrete counterexample would give rise to a SELTZO counterexample. However, if S turns out to be satisfiable, then we cannot conclude whether a concrete floating-point counterexample exists. In other words, every correctness condition verified by the SELTZO abstraction is rigorously and provably true, but there are true statements that the SELTZO abstraction is unable to verify. Working in the SELTZO domain requires us to accept some reduction in the logical strength of the statements we are able to prove. Fortunately, we will see that this loss of logical strength is minor and compensated by many orders of magnitude of increased verification performance.

To demonstrate the viability of this strategy, the following proposition shows that the preconditions of FastTwoSum (Proposition 4) and strong nonoverlapping (Proposition 8), in addition to the alternative nonoverlapping conditions used in prior work (Definitions 18 and 19), can all be equivalently reformulated as linear inequalities in the SELTZO variables.

Proposition 9 (SELTZO correctness conditions)**.** *Let x and y be nonzero floating-point numbers, and let $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ and $(s_y, e_y, \mathsf{nlz}_y, \mathsf{nlo}_y, \mathsf{ntz}_y, \mathsf{nto}_y)$ denote their SELTZO abstractions.*

1. *x and y are valid inputs to* FastTwoSum *if:*

$$e_x + \mathsf{ntz}_x \geq e_y \tag{4.1}$$

2. *$x \succ y$ if and only if:*

$$\begin{aligned}\big[e_x > e_y + (p+1)\big] \lor \big[e_x = e_y + (p+1) \land (s_x = s_y \lor \mathsf{ntz}_x < p-1 \lor \mathsf{ntz}_y = p-1)\big] \lor \\ \big[e_x = e_y + p \land \mathsf{ntz}_y = p-1 \land \mathsf{ntz}_x > 0 \land (s_x = s_y \lor \mathsf{ntz}_x < p-1)\big]\end{aligned} \tag{4.2}$$

3. *$x \succ_{\mathcal{S}} y$ if and only if:*
$$e_x \geq e_y + (p - \mathsf{ntz}_x) \tag{4.3}$$

4. *$x \succ_{\mathcal{P}} y$ if and only if:*
$$e_x \geq e_y + p \tag{4.4}$$

5. *$x \succ_{\mathsf{ulp}} y$ if and only if:*
$$\big[e_x > e_y + (p-1)\big] \vee \big[e_x = e_y + (p-1) \wedge \mathsf{ntz}_y = p-1\big] \tag{4.5}$$

6. *$x \succ_{\mathsf{QD}} y$ if and only if:*
$$\big[e_x > e_y + p\big] \vee \big[e_x = e_y + p \wedge \mathsf{ntz}_y = p-1\big] \tag{4.6}$$

Proof. These claims follow immediately from Proposition 4, Proposition 8, Definition 18, and the observation that y is a signed power of two if and only if $\mathsf{ntz}_y = p - 1$. □

These statements ignore the possibility of x or y being zero because the exponent of zero is undefined when working in the unbounded floating-point domain. Our verifier implementation uses special representations of `+0.0` and `-0.0` described in Section 4.4.

The next ingredient of our automatic verification strategy is a way to express relative bounds of the form $|y| \leq C\mathbf{u}^k|x|$ for $C \in \mathbb{R}$ and $k \in \mathbb{Z}$. Because the SELTZO abstraction is formulated in terms of bit counts, these relative bounds are most naturally expressed when $C = 2^j$ is a power of two. We will restrict our attention to relative bounds of this form throughout the remainder of this dissertation. This means that our SELTZO verification technique is unable to prove the optimal ddadd error bound $|y| \leq 3\mathbf{u}^2|x|$; our analysis of ddadd will only prove the weaker bound $|y| \leq 4\mathbf{u}^2|x|$. This suboptimal constant factor C is an intentional sacrifice that we make in order to automate rigorous FPAN verification.

Proposition 10 (SELTZO relative bounds)**.** *Let x and y be nonzero floating-point numbers, let $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ and $(s_y, e_y, \mathsf{nlz}_y, \mathsf{nlo}_y, \mathsf{ntz}_y, \mathsf{nto}_y)$ denote their SELTZO* *abstractions, and let* $j, k \in \mathbb{Z}$. *If*

$$\big[e_x > e_y + (kp - j)\big] \vee \big[e_x = e_y + (kp - j) \wedge (\mathsf{nlo}_x > \mathsf{nlo}_y \vee \mathsf{nlz}_x < \mathsf{nlz}_y \vee \mathsf{ntz}_y = p - 1)\big] \tag{4.7}$$

then $|y| \le 2^j \mathbf{u}^k |x|$.

Proof. The unit roundoff (Definition 14) in base $b = 2$ is $\mathbf{u} := \frac{1}{2} b^{-(p-1)} = 2^{-p}$. Hence, our goal is to prove $|y| \le 2^{-(kp-j)}|x|$. If $e_x > e_y + (kp - j)$, then we can write

$$|x| \ge 2^{e_x} \ge 2^{e_y + (kp-j)+1} = 2^{e_y+1} 2^{kp-j} \ge 2^{kp-j}|y| \tag{4.8}$$

which is the desired result. Otherwise, if $e_x = e_y + (kp - j)$, then we need to prove that the mantissa of x is at least as large as the mantissa of y. This is true whenever x has strictly more leading ones, strictly fewer leading zeros, or if y has an all-zero mantissa.[1] □

Note that distinct SELTZO tuples correspond to disjoint sets of floating-point numbers. In particular, the bit counts $(\mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ are *not* cumulative; $\mathsf{ntz}_x = 3$ specifies floating-point numbers whose mantissa contains *exactly* three trailing zeros, no more. Formulating the definition in this way confers the useful property that the set of floating-point numbers having a particular SELTZO tuple $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ decreases exponentially in size as the bit counts $(\mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ increase. This is desirable because floating-point numbers with lots of leading zeros or ones are precisely the numbers closest to signed powers of two. FPANs tend to exhibit pathological behaviors near signed powers of two because these numbers lie on the boundaries between different exponent regimes, where rounding and tie-breaking analysis exhibits particularly tricky edge cases [54, 73].

[1]The statement of Proposition 10 can be strengthened by adding additional sufficient conditions for the mantissa of x to be greater than or equal to the mantissa of y. For example, this also occurs when x has an all-ones mantissa ($\mathsf{nto}_x = p - 1$). However, we have not found additional sufficient conditions of this type to be useful in proving FPAN relative error bounds.

4.2 The SE and SETZ Abstractions

It is natural to ask whether all six of the SELTZO variables $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ are truly necessary to reason about the FPAN correctness conditions. The statement of Proposition 9 only makes reference to s_x, e_x, and ntz_x, so it is not obvious whether it is helpful to explicitly model the leading bit counts $\mathsf{nlz}_x, \mathsf{nlo}_x$ or the trailing one count nto_x. To investigate this question, we define two simpler models, called the *SE* and *SETZ abstractions*, that involve subsets of the SELTZO variables.

Definition 24 (*SE abstraction*, *SETZ abstraction*)**.** Let x be a nonzero floating-point number. The *sign-exponent (SE) abstraction* of x is the ordered pair (s_x, e_x) consisting of its sign bit $s_x \in \{0, 1\}$ and exponent $e_x \in \mathbb{Z}$. The *sign-exponent-trailing-zeros (SETZ) abstraction* of x is the ordered triple $(s_x, e_x, \mathsf{ntz}_x)$ that additionally includes the number $\mathsf{ntz}_x \in \mathbb{N}$ of trailing zeros in the mantissa of x.

The SETZ abstraction is a particularly natural choice of model because it captures all of the variables referenced in the statement of Proposition 9. The SE abstraction, on the other hand, can only express the notion of $\mathcal{P}$-nonoverlapping and is too weak to capture the other nonoverlapping conditions. Despite this restriction, we will later see that both the SE and SETZ abstractions are capable of proving nontrivial FPAN correctness results, though their logical strength is meaningfully weaker than the full SELTZO abstraction.

Note that the statement of the relative bound $|y| \le 2^j \mathbf{u}^k |x|$ given in Proposition 10 involves the leading bit counts nlz_x and nlo_x omitted from the SE and SETZ abstractions. It is therefore necessary to weaken the sufficient condition (4.7) into

$$\big[e_x > e_y + (kp - j)\big] \vee \big[e_x = e_y + (kp - j) \wedge \mathsf{ntz}_y = p - 1\big] \tag{4.9}$$

when working in the SETZ abstraction, and to further weaken this condition into

$$e_x > e_y + (kp - j) \tag{4.10}$$

when working in the SE abstraction. The necessity of this weakening offers some suggestion of the reduced logical strength of these simpler models.

4.3 TwoSum Lemmas

The final ingredient necessary to implement our automatic verification strategy is a formal model of TwoSum in the SE, SETZ, and SELTZO abstractions. To be suitable for use in a QF-LIA satisfiability problem, this formal model should be a logical formula, consisting of linear equations and inequalities over the integers, that describes the possible input-output pairs of the TwoSum operation. For example, given the SETZ tuples $(s_x, e_x, \mathsf{ntz}_x)$ and $(s_y, e_y, \mathsf{ntz}_y)$ of two floating-point numbers, x and y, our formal model should predict the SETZ tuples $(s_s, e_s, \mathsf{ntz}_s)$ and $(s_e, e_e, \mathsf{ntz}_e)$ of the outputs $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$.

In general, it is possible for many floating-point numbers x to share the same SETZ abstraction tuple $(s_x, e_x, \mathsf{ntz}_x)$. We therefore cannot expect the SETZ tuples of the outputs (s, e) to be *uniquely* determined by the SETZ tuples of the inputs (x, y). We think of TwoSum as a *relation*, rather than a function, over the SETZ abstract domain. Our formal model should determine a *set* of possible SETZ outputs (s, e) for a given pair of SETZ inputs (x, y). Of course, these remarks also apply to the SE and SELTZO abstract domains.

Although FPANs can include three distinct gate types, corresponding to the floating-point sum, TwoSum, and $\mathsf{FastTwoSum}$ operations, it is only necessary to construct a formal model for TwoSum. Floating-point addition $s \coloneqq x \oplus y$ is equivalent to $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$ with the rounding error e discarded, and $\mathsf{FastTwoSum}(x, y)$ is equivalent to $\mathsf{TwoSum}(x, y)$ whenever the preconditions of Proposition 4 are satisfied. Since these preconditions are explicitly checked in our verification procedure, we can treat all $\mathsf{FastTwoSum}$ gates as though they were TwoSum gates for the purpose of formal modeling.

We begin by presenting a formal model of TwoSum in the SE abstraction. Despite being our simplest abstraction, modeling only signs and exponents, the edge cases of the rounding function RNE split our SE model of TwoSum into twelve distinct cases. Rather than presenting all of these cases amalgamated into a single enormous formula, we present each case as a separate TwoSum *lemma*, each consisting of a precondition on the SE input tuples (s_x, e_x) and (s_y, e_y), followed by a specification of the set of all possible SE output tuples $\{(s_s, e_s), (s_e, e_e)\}$ that can occur when these preconditions are satisfied.

In all of the following lemmas, let x and y be nonzero floating-point numbers, and let $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$. Let (s_x, e_x), (s_y, e_y), (s_s, e_s), and (s_e, e_e) denote the SE abstractions of x, y, s, and e, respectively.

Lemma 1 (SE Identity x)**.** *If $e_x > e_y + (p + 1)$, or if $e_x = e_y + (p + 1)$ and $s_x = s_y$, then $s = x$ and $e = y$.*

Lemma 2 (SE Identity y)**.** *If $e_x + (p + 1) < e_y$, or if $e_x + (p + 1) = e_y$ and $s_x = s_y$, then $s = y$ and $e = x$.*

Note that Lemmas 1 and 2 differ only by exchanging the roles of x and y. Because TwoSum is a commutative operation (i.e., $\mathsf{TwoSum}(x, y) = \mathsf{TwoSum}(y, x)$), each TwoSum lemma remains valid when x and y are interchanged. To avoid needless repetition, we state only one member of each symmetric lemma pair, adopting the convention that we prefer the lemma statement with $e_x \geq e_y$ whenever possible.

Lemma 3 (SE-S1)**.** *If $s_x = s_y$ and $e_x = e_y + p$, then exactly one of the following statements is true:*

- *$s = x$ and $e = y$.*
- *$s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, $s_e \neq s_y$, and $e_y - (p - 1) \leq e_e \leq e_x - p$.*

Lemma 4 (SE-S2)**.** *If $s_x = s_y$ and $e_x = e_y + (p - 1)$, then exactly one of the following statements is true:*

- *$s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, and $e = 0$.*
- *$s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, and $e_y - (p - 1) \leq e_e \leq e_x - p$. (When not explicitly specified, the sign bit s_e can be 0 or 1).*

Lemma 5 (SE-S3)**.** *If $s_x = s_y$ and $e_x = e_y + (p - 2)$, then exactly one of the following statements is true:*

- $s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, *and* $e = 0$.
- $s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, $s_e \neq s_y$, *and* $e_y - (p-1) \leq e_e \leq e_x - p$.
- $s_s = s_x$, $e_s = e_x$, $s_e = s_y$, *and* $e_y - (p-1) \leq e_e \leq e_x - p$.
- $s_s = s_x$, $e_s = e_x + 1$, $s_e = s_y$, *and* $e_y - (p-1) \leq e_e \leq e_x - (p-1)$.

Lemma 6 (SE-S4). *If* $s_x = s_y$, $e_x > e_y$, *and* $e_x < e_y + (p-2)$, *then exactly one of the following statements is true:*

- $s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, *and* $e = 0$.
- $s_s = s_x$, $e_s = e_x$, *and* $e_y - (p-1) \leq e_e \leq e_x - p$.
- $s_s = s_x$, $e_s = e_x + 1$, *and* $e_y - (p-1) \leq e_e \leq e_x - (p-1)$.

Lemma 7 (SE-S5). *If* $s_x = s_y$ *and* $e_x = e_y$, *then exactly one of the following statements is true:*

- $s_s = s_x$, $e_s = e_x + 1$, *and* $e = 0$.
- $s_s = s_x$, $e_s = e_x + 1$, *and* $e_e = e_x - (p-1)$.

Lemma 8 (SE-D1). *If* $s_x \neq s_y$ *and* $e_x = e_y + (p+1)$, *then exactly one of the following statements is true:*

- $s = x$ *and* $e = y$.
- $s_s = s_x$, $e_s = e_x - 1$, $s_e \neq s_y$, *and* $e_y - (p-1) \leq e_e \leq e_x - (p+2)$.

Lemma 9 (SE-D2). *If* $s_x \neq s_y$ *and* $e_x = e_y + p$, *then exactly one of the following statements is true:*

- $s = x$ *and* $e = y$.
- $s_s = s_x$, $e_s = e_x - 1$, *and* $e = 0$.
- $s_s = s_x$, $e_s = e_x - 1$, $s_e = s_y$, *and* $e_y - (p-1) \leq e_e \leq e_x - (p+2)$.

- $s_s = s_x$, $e_s = e_x - 1$, $s_e \neq s_y$, *and* $e_y - (p-1) \leq e_e \leq e_x - (p+1)$.
- $s_s = s_x$, $e_s = e_x$, $s_e \neq s_y$, *and* $e_y - (p-1) \leq e_e \leq e_x - p$.

Lemma 10 (SE-D3)**.** *If* $s_x \neq s_y$, $e_x > e_y + 1$, *and* $e_x < e_y + p$, *then exactly one of the following statements is true:*

- $s_s = s_x$, $e_x - 1 \leq e_s \leq e_x$, *and* $e = 0$.
- $s_s = s_x$, $e_s = e_x - 1$, *and* $e_y - (p-1) \leq e_e \leq e_x - (p+1)$.
- $s_s = s_x$, $e_s = e_x$, *and* $e_y - (p-1) \leq e_e \leq e_x - p$.

Lemma 11 (SE-D4)**.** *If* $s_x \neq s_y$ *and* $e_x = e_y + 1$, *then exactly one of the following statements is true:*

- $s_s = s_x$, $e_x - p \leq e_s \leq e_x$, *and* $e = 0$.
- $s_s = s_x$, $e_s = e_x$, *and* $e_e = e_x - p$.

Lemma 12 (SE-D5)**.** *If* $s_x \neq s_y$ *and* $e_x = e_y$, *then exactly one of the following statements is true:*

- $s = 0$ *and* $e = 0$.
- $e_x - (p-1) \leq e_s \leq e_x - 1$ *and* $e = 0$.

Lemmas 1–12 constitute a complete formal model of TwoSum in the SE domain, in the sense that every possible pair of SE input tuples, (s_x, e_x) and (s_y, e_y), satisfies the hypotheses of exactly one lemma. Each lemma has been formally verified using a bit-blasting SMT solver in the bfloat16, binary16, binary32, binary64, and binary128 floating-point formats, which have precision $p = 8$, 11, 24, 53, and 113, respectively. We expect these lemmas to hold for all values of $p \geq 2$, but bit-blasting can only verify each lemma for one particular value of $p \in \mathbb{N}$ at a time. We do not yet have an automatic verification technique[2] that can simultaneously prove these lemmas for all values of p using a single finite computation.

[2]Symbolic interval analysis using a computer algebra system may furnish such a technique for the SE abstraction, but this technique is not applicable to the SETZ and SELTZO abstractions because intervals are broken by constraints on trailing bits.

We have also verified, by exhaustive enumeration of all concrete input pairs (x, y) in the binary16 and bfloat16 floating-point formats, that all of these lemmas are stated in the strongest possible form. In other words, every element of the set of allowed SE output tuples listed in each lemma is actually witnessed by some pair of concrete floating-point inputs (x, y) satisfying the hypotheses of the lemma.

Formal statements of Lemmas 1–12 in the Z3 Python API are provided in the GitHub repository `https://github.com/dzhang314/FPANVerifier`. This repository also contains the aforementioned exhaustive enumeration program, used to verify that each lemma is stated in the strongest possible form, along with scripts that run a portfolio of bit-blasting SMT solvers, including Z3 [24], CVC5 [8], MathSAT 5 [20], and Bitwuzla [77], to verify all lemmas in parallel. We also provide six additional lemmas that formally characterize the TwoProd operation in the SE domain in an analogous fashion.

Given that the SE model of TwoSum is sufficiently complicated to require a dozen cases, it is reasonable to expect construction of a formal model of TwoSum in the SETZ or SELTZO domain to be a formidable task. We have constructed a complete formal model of TwoSum in the SETZ domain consisting of over sixty lemmas, presented in Appendix A. Like our SE lemmas, our SETZ lemmas are also complete in the sense of covering all possible SETZ input tuples. They have also been formally verified using bit-blasting SMT solvers and are confirmed by exhaustive enumeration to be stated in the strongest possible form.

Characterizing TwoSum in the SELTZO domain is far more challenging still. Despite containing several *hundred* lemmas, our formal model of TwoSum in the SELTZO domain remains incomplete, covering only a small fraction of all possible SELTZO input pairs. These SELTZO lemmas are so numerous and so complicated that we do not provide a human-readable listing in this dissertation. We refer to our implementation for the authoritative list (`https://github.com/dzhang314/FPANVerifier/blob/main/seltzo_lemmas.py`) of SELTZO lemmas. As with our SE and SETZ lemmas, all of our SELTZO lemmas have been formally verified using bit-blasting SMT solvers, and most (but not all) are confirmed by exhaustive enumeration to be stated in the strongest possible form.

Despite its incompleteness, our collection of SELTZO lemmas still covers enough of the

SELTZO domain to prove results that are meaningfully stronger than the SE and SETZ abstractions. This lack of completeness does not threaten the validity of our results because our verification technique is based on overapproximation. Incompleteness merely implies that our solver will be overly conservative in deducing possible TwoSum outputs in certain regions of the input space. In some cases, it may be possible to further strengthen our results by adding additional SELTZO lemmas, but we leave this task to future work.

4.4 Verifier Implementation

With all necessary ingredients in hand, we are now prepared to formally state our automatic procedure for verifying the FPAN correctness conditions stated in Section 3.3. To facilitate independent verification of our results by other researchers, a permissively-licensed open-source implementation of this procedure is provided in the following GitHub repository: `https://github.com/dzhang314/FPANVerifier`

We first address the issue of representing zero. Definitions 23 and 24 do not specify the SE, SETZ, or SELTZO abstraction of zero because zero has no well-defined exponent in the unbounded floating-point domain $\mathbb{FP}(b, p)$. In our implementation, we choose an arbitrary minimum exponent $e_{\text{min}} \in \mathbb{Z}$ and define zero to have exponent $e_{\text{min}} - 1$, while all nonzero floating-point numbers are assumed to be normalized with exponent $e \geq e_{\text{min}}$. This choice is particularly convenient because it coincides with IEEE encoding (Definition 7).

The correctness of any FPAN is independent of the choice of e_{min} because the floating-point sum, TwoSum, and FastTwoSum operations are all equivariant to global exponent translations. In other words, if all inputs are multiplied by some power of two, then all outputs scale by the same power of two. The FastTwoSum preconditions (Proposition 4), nonoverlapping conditions (Proposition 9), and relative error bounds (Proposition 10) only depend on relative differences between exponents, not absolute exponents, so the truth of any of these statements is invariant with respect to global exponent translation.

This observation also implies that any FPAN correctness condition that holds in the unbounded floating-point domain $\mathbb{FP}(2, p)$ also holds in the standard floating-point domain

$\mathbb{FP}(2, p, e_{\min}, e_{\max})$, assuming overflow does not occur. Any counterexample involving subnormal numbers can be scaled by some power of two to yield an equivalent normalized counterexample. Thus, the presence of subnormal numbers cannot create any counterexamples that would not already exist in the unbounded floating-point domain.

Now, suppose we are given an FPAN F and a property P that is expressible as a logical combination of linear equations and inequalities in the SELTZO variables $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$. We will construct a QF-LIA statement S that expresses the existence of an abstract counterexample to P. We then query an SMT solver to determine whether S is satisfiable. If S is unsatisfiable, then P has no abstract counterexamples, and hence, no concrete counterexamples. This constitutes a formal proof that F has property P.

We first assign a unique label v_i to every *wire segment* in F. We think of every gate as delineating a new segment of the two wires it connects, and we consider each addition gate to have a hidden second output wire carrying its discarded rounding error. Thus, an FPAN with n wires and g gates has $n + 2g$ distinct wire segments. We then introduce SELTZO variables $(s_{v_i}, e_{v_i}, \mathsf{nlz}_{v_i}, \mathsf{nlo}_{v_i}, \mathsf{ntz}_{v_i}, \mathsf{nto}_{v_i})$ indexed by the labels v_i, creating a total of $6n+12g$ such variables. We form the QF-LIA statement S by taking the logical conjunction of five types of conditions:

1. *consistency conditions* that require each of the SELTZO tuples $(s_{v_i}, e_{v_i}, \mathsf{nlz}_{v_i}, \mathsf{nlo}_{v_i}, \mathsf{ntz}_{v_i}, \mathsf{nto}_{v_i})$ to be populated with internally consistent integer values;

2. *input conditions* that enforce preconditions, such as strong nonoverlapping, on the inputs of F;

3. *execution conditions* that use the TwoSum lemmas to constrain the possible outputs of each gate;

4. FastTwoSum *conditions* that ensure each FastTwoSum gate receives valid inputs satisfying Proposition 4; and

5. *counterexample conditions* that encode the negation of the desired property P.

We first state the consistency conditions as Equations (4.11)–(4.18), which form a necessary and sufficient characterization of all valid SELTZO tuples. One copy of these consistency conditions is appended to the QF-LIA statement S for each label v.

1. The sign bit must be zero or one, and the exponent must be bounded below.

$$(s_v = 0) \lor (s_v = 1) \qquad\qquad e_v \geq e_{\min} - 1 \tag{4.11}$$

2. If a floating-point variable is zero (i.e., $e_v = e_{\min} - 1$), then its mantissa must consist entirely of zeros.

$$(e_v = e_{\min} - 1) \implies [(\mathsf{nlz}_v = \mathsf{ntz}_v = p - 1) \land (\mathsf{nlo}_v = \mathsf{nto}_v = 0)] \tag{4.12}$$

3. The leading and trailing bits of the mantissa are either 0 or 1.

$$[(\mathsf{nlz}_v > 0) \land (\mathsf{nlo}_v = 0)] \lor [(\mathsf{nlz}_v = 0) \land (\mathsf{nlo}_v > 0)] \tag{4.13}$$

$$[(\mathsf{ntz}_v > 0) \land (\mathsf{nto}_v = 0)] \lor [(\mathsf{ntz}_v = 0) \land (\mathsf{nto}_v > 0)] \tag{4.14}$$

4. The number of leading and trailing bits must be bounded by $p - 1$, the width of the mantissa.

$$(\mathsf{nlz}_v = \mathsf{ntz}_v = p - 1) \lor (\mathsf{nlz}_v + \mathsf{ntz}_v < p - 1) \tag{4.15}$$

$$(\mathsf{nlo}_v = \mathsf{nto}_v = p - 1) \lor (\mathsf{nlo}_v + \mathsf{nto}_v < p - 1) \tag{4.16}$$

$$(\mathsf{nlz}_v + \mathsf{nto}_v = p - 1) \lor (\mathsf{nlz}_v + \mathsf{nto}_v < p - 2) \tag{4.17}$$

$$(\mathsf{ntz}_v + \mathsf{nlo}_v = p - 1) \lor (\mathsf{ntz}_v + \mathsf{nlo}_v < p - 2) \tag{4.18}$$

The upper bound of $p - 2$ in consistency conditions (4.17) and (4.18) arises from the observation that the middle bit b in a bit vector of the form $(0, \ldots, 0, b, 1, \ldots, 1)$ must either belong to the group of leading zeros or the group of trailing ones. Thus, it is impossible for $\mathsf{nlz}_v + \mathsf{nto}_v$ to equal $p - 2$. An analogous condition holds for bit vectors of the form

$(1, \ldots, 1, b, 0, \ldots, 0)$, yielding the constraint $\mathsf{nlo}_v + \mathsf{ntz}_v \neq p - 2$.

After constructing the consistency conditions, we append the user-provided input conditions to S, followed by the execution conditions. The execution conditions consist of one copy of every TwoSum lemma applied to the input and output variables of each gate in F. Our verifier implementation can operate in one of three user-selectable modes, offering a choice between the SE, SETZ, and SELTZO abstractions. In SE mode, only Lemmas 1–12 are instantiated in this step. In SETZ mode, Lemmas 1–12 are ignored, and the SETZ lemmas in Appendix A are used instead. In SELTZO mode, we combine the SETZ lemmas from Appendix A with the full list of SELTZO lemmas. This combination allows the SETZ lemmas to provide coverage in regions where the SELTZO lemmas are incomplete.

We then append one copy of the FastTwoSum preconditions to S for each FastTwoSum gate in F, followed by the user-provided counterexample conditions. Unlike the consistency, input, and execution conditions, which are appended to S by conjunction, the FastTwoSum and counterexample conditions are appended to S by disjunction. We formulate these conditions as a logical disjunction of failure modes, such as a violation of a nonoverlapping condition $z_{i-1} \not\succ z_i$ or a desired error bound $|z_i| > 2^j \mathbf{u}^k |z_0|$, stated in QF-LIA using Propositions 9 and 10. The occurrence of any failure mode invalidates the property P being verified. This completes the construction of the QF-LIA statement S.

To finish our verification procedure, we query an SMT solver to determine whether the statement S is satisfiable. If S is unsatisfiable, then we have successfully proven that the FPAN F has the desired property P. However, if S turns out to be satisfiable, then we cannot conclude anything about the truth of P in general. In some cases, we may be able to construct an explicit counterexample to P by examining a satisfying assignment of S produced by the SMT solver, but this requires *ad hoc* analysis and is not always possible. To guard against bugs in any particular SMT reasoning engine, our verifier is compatible with all SMT solvers that implement the SMT-LIB 2 standard [9]. We have independently confirmed our results using a variety of SMT solvers, including Z3 [24], CVC5 [8], MathSAT 5 [20], Yices 2 [26], OpenSMT [15], and Bitwuzla [77].

In contrast to bit-blasting, which can only verify a claim for one precision $p \in \mathbb{N}$ at a time, our QF-LIA verification procedure treats the precision p as a variable which is implicitly universally quantified. Thus, a single run of our verification procedure simultaneously proves the property P over all values of $p \in \mathbb{N}$. In some cases, we must impose a lower bound on p to rule out degenerate edge-case behaviors that only occur in pathologically small floating-point formats. Our implementation assumes $p \geq 8$ by default.

The QF-LIA statements generated by our verification procedure tend to be very large, involving hundreds of variables and thousands of constraints, even for FPANs as small as ddadd (Figure 4.2). Any proof of unsatisfiability for such a statement is an enormous computational artifact that is essentially unreadable to humans. This unfortunate property makes our verification technique somewhat opaque, offering limited intuitive insight as to why some FPANs work when other similar-looking FPANs fail. Nonetheless, our verification procedure is fully rigorous, with the correctness of every step established by traditional (pen-and-paper) and/or formal (computer-verified) mathematical proofs.

4.5 Verifier Evaluation

We now apply the verification procedure described in the previous section to prove relative error bounds for both the previous best known double-double addition algorithm, ddadd (Figure 4.2), which is used in several production software libraries [64, 22, 91], and a novel algorithm that is simultaneously faster and more accurate than ddadd. Our new algorithm, named madd (for "More Accurate Double-Double addition"), reduces the relative error of double-double addition from $3\mathbf{u}^2$ to $2\mathbf{u}^2$ while lowering its circuit depth from 5 to 4, as shown in Figure 4.3. The process by which we discovered the madd algorithm is described in Chapter 5. In this section, we take the existence of madd for granted and use it alongside ddadd to evaluate our automatic verification procedure.

To obtain stronger bounds on the magnitude of the rounding errors discarded by ddadd and madd, we add an extra TwoSum gate to each FPAN that computes the total rounding error rather than analyzing the rounding error terms separately. This produces the *augmented FPANs* shown in Figure 4.4. These extra gates serve only to facilitate our analysis

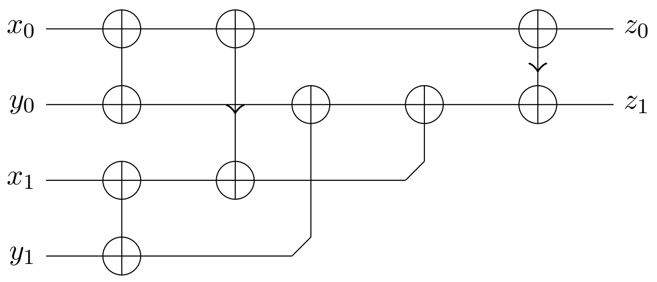


Figure 4.3: FPAN representation of madd, our new improved algorithm for double-double addition.

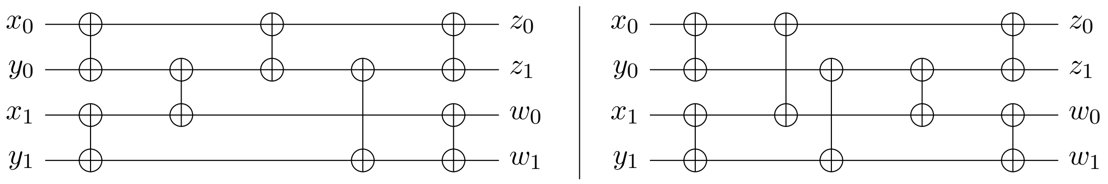


Figure 4.4: Augmented FPAN representations of ddadd (left) and madd (right) with error terms (w_0, w_1) explicitly computed and named. The extra TwoSum gate used to compute (w_0, w_1) serves only to facilitate our analysis and should not be included in an actual implementation of either algorithm.

and should not be included in an actual software implementation of ddadd or madd.

Theorem 1. *Let (x_0, x_1) and (y_0, y_1) be strongly nonoverlapping floating-point expansions. The ddadd algorithm (Figure 4.2) computes a strongly nonoverlapping floating-point expansion (z_0, z_1) that approximates the exact sum $x_0 + x_1 + y_0 + y_1$ with relative error*

$$\frac{|(z_0 + z_1) - (x_0 + x_1 + y_0 + y_1)|}{|x_0 + x_1 + y_0 + y_1|} \le (1 + 2\mathbf{u})4\mathbf{u}^2 = 4\mathbf{u}^2 + O(\mathbf{u}^3). \tag{4.19}$$

Theorem 2. *Let (x_0, x_1) and (y_0, y_1) be strongly nonoverlapping floating-point expansions. The madd algorithm (Figure 4.3) computes a strongly nonoverlapping floating-point expansion (z_0, z_1) that approximates the exact sum $x_0 + x_1 + y_0 + y_1$ with relative error*

$$\frac{|(z_0 + z_1) - (x_0 + x_1 + y_0 + y_1)|}{|x_0 + x_1 + y_0 + y_1|} \le (1 + 2\mathbf{u})2\mathbf{u}^2 = 2\mathbf{u}^2 + O(\mathbf{u}^3). \tag{4.20}$$

We prove Theorems 1 and 2 by using our SELTZO verification procedure to prove a suitably chosen property P stated in the following proof. Some subsequent algebraic manipulation is necessary to transform P into the desired relative error bound.

Proof. Consider the augmented FPANs shown in Figure 4.4, which include an extra TwoSum gate to compute the error terms (w_0, w_1). Assuming the input conditions $x_0 \succ x_1$ and $y_0 \succ y_1$, we use our automatic verification procedure to prove the property $P := |w_0| \leq 2^j \mathbf{u}^2 |z_0|$ (formulated in the SELTZO abstraction using Propositions 9 and 10), where $j = 2$ for ddadd and $j = 1$ for madd. Since (z_0, z_1) and (w_0, w_1) are both outputs of TwoSum, Proposition 7 implies that $z_0 = z_0 \oplus z_1$ and $w_0 = w_0 \oplus w_1$. Hence, by Proposition 2, we can write $z_0 + z_1 = (1 + \delta_z) z_0$ and $w_0 + w_1 = (1 + \delta_w) w_0$ for some $|\delta_z|, |\delta_w| \leq \mathbf{u}$. It follows that

$$\begin{aligned} \frac{|(z_0 + z_1) - (x_0 + x_1 + y_0 + y_1)|}{|x_0 + x_1 + y_0 + y_1|} &= \frac{|(1 + \delta_w) w_0|}{|(1 + \delta_z) z_0 + (1 + \delta_w) w_0|} \\ &\leq \frac{(1 + \mathbf{u})|w_0|}{(1 - \mathbf{u})(|z_0| - |w_0|)} \leq \frac{(1 + \mathbf{u}) 2^j \mathbf{u}^2}{1 - (1 - \mathbf{u}) 2^j \mathbf{u}^2} \leq (1 + 2\mathbf{u}) 2^j \mathbf{u}^2 \end{aligned} \tag{4.21}$$

which is the desired result. □

This proof demonstrates the technique of *augmenting* an FPAN with additional TwoSum gates to accumulate all discarded rounding errors into a single *principal error term.* This accumulation step can reveal cancellation patterns that would not be visible if each rounding error term were analyzed separately, strengthening the relative error bounds that the SELTZO abstraction is able to prove. We will implicitly apply this technique in all FPAN analyses presented in the remainder of this dissertation without further elaboration.

As previously noted, our relative error bound of $4\mathbf{u}^2$ for ddadd is slightly weaker than the $3\mathbf{u}^2$ bound proven by Joldes, Muller, and Popescu [54], since our use of the SELTZO abstraction forces the leading constant factor to be a power of two. We say that a bound of this form is *tight to the nearest bit* if its leading constant factor is at most twice the optimal constant factor. The following worst-case input for ddadd, discovered by Muller and Rideau [73], shows that our $4\mathbf{u}^2$ bound is indeed tight to the nearest bit.

$$x_0 := 1 \qquad x_1 := \mathbf{u} - \mathbf{u}^2 \qquad y_0 := -\frac{1}{2} + \frac{\mathbf{u}}{2} \qquad y_1 := -\frac{\mathbf{u}^2}{2} + \mathbf{u}^3 \tag{4.22}$$

It is straightforward to verify that ddadd computes the sum of these inputs with relative error $\approx 3\mathbf{u}^2$. We can therefore conclude that our bound of $4\mathbf{u}^2$ is tight to the nearest bit, since its leading constant factor is at most twice the optimal leading constant. Similarly, the following worst-case input for madd, discovered using a stochastic search procedure presented in Section 5.1, shows that our $2\mathbf{u}^2$ bound for madd is also tight to the nearest bit.

$$x_0 \coloneqq 1 + 2\mathbf{u} \qquad x_1 \coloneqq -\frac{\mathbf{u}}{2} - 2\mathbf{u}^2 \qquad y_0 \coloneqq -\mathbf{u} \qquad y_1 \coloneqq -\frac{\mathbf{u}^2}{2} - \mathbf{u}^3 \tag{4.23}$$

The sum computed by madd on these inputs has relative error $\approx 1.5\mathbf{u}^2$.

We now turn our attention to the question of verification performance. It is not obvious *a priori* that our procedure, which checks the satisfiability of linear equations and inequalities over the integers, should be any faster than bit-blasting. After all, Boolean satisfiability and QF-LIA satisfiability are both NP-complete problems.

In Table 4.1, we compare the speed of verifying a property P expressed in QF-LIA using the SELTZO abstraction to verifying the same property P expressed directly in the theory of floating-point numbers (QF-FP). The property P in question is the same property used to prove Theorems 1 and 2, as stated above, and is separately timed for both ddadd and madd. Our benchmarks evaluate a portfolio of state-of-the-art SMT solvers using the latest software versions available at the time of writing, including Z3 4.13.4, CVC5 1.2.0, MathSAT 5.6.11, Bitwuzla 0.7.0, and Colibri2 0.4. The first four of these SMT solvers implement floating-point reasoning by bit-blasting, while Colibri2 uses projection from real arithmetic. All of these SMT solvers were evaluated for their floating-point reasoning capabilities, while only Z3 was used to solve QF-LIA satisfiability problems via its Python API.

In all cases, our FPAN verification problems are many orders of magnitude faster to solve in the SELTZO abstraction. We observe measured speedups of roughly five orders of magnitude and implied speedups exceeding six orders of magnitude (more one million times faster) in trials that were terminated early due to failure to finish after three days of continuous runtime (labeled by "DNF" entries in Table 4.1). Moreover, our SELTZO solve times remain constant when the floating-point precision p is increased, enabling scalability to

FPAN	Format	Z3	CVC5	MathSAT	Bitwuzla	Colibri2	SELTZO (Z3)
ddadd	binary16	DNF	153 min	DNF	72 min	N/A	0.927 sec
madd	binary16	DNF	120 min	3898 min	72 min	N/A	0.713 sec
ddadd	bfloat16	DNF	704 min	DNF	71 min	N/A	0.838 sec
madd	bfloat16	DNF	946 min	DNF	99 min	N/A	0.689 sec
ddadd	binary32	DNF	1088 min	DNF	640 min	N/A	0.774 sec
madd	binary32	DNF	1019 min	DNF	518 min	N/A	0.722 sec
ddadd	binary64	DNF	DNF	DNF	DNF	N/A	0.623 sec
madd	binary64	DNF	DNF	DNF	DNF	N/A	0.923 sec
ddadd	binary128	DNF	DNF	DNF	DNF	N/A	0.880 sec
madd	binary128	DNF	DNF	DNF	DNF	N/A	0.991 sec

Table 4.1: Execution time for various SMT solvers to verify property P expressed in the theory of floating-point numbers (`QF_FP`) compared to the theory of linear integer arithmetic (`QF_LIA`) via the SELTZO abstraction. A "DNF" entry indicates that a solver did not terminate within three days, while an "N/A" entry indicates that a solver rejected the problem as unsolvable. These benchmarks were performed on an AMD Ryzen 9 9950X processor using Z3 4.13.4, CVC5 1.2.0, MathSAT 5.6.11, Bitwuzla 0.7.0, and Colibri2 0.4. SELTZO satisfiability problems were solved using Z3 4.13.4.

wide floating-point formats that are intractable for bit-blasting. We also note that Colibri2, an SMT solver whose floating-point reasoning engine uses projection from real arithmetic, immediately rejects all of these problems as unsolvable. This failure demonstrates our claim that methods of this type are fundamentally inapplicable to FPAN verification.

Finally, in Table 4.2, we compare the logical strength of the SE, SETZ, and SELTZO abstractions by computing the parameters $j, k \in \mathbb{Z}$ of the strongest relative error bound $|w_0| \leq 2^j \mathbf{u}^k |z_0|$ that is provable for ddadd and madd in each abstract domain. We determine these optimal parameters by using our automatic verification procedure to conduct an iterated binary search. We first hold j fixed at a large value ($j = 64$ in our implementation) and perform a binary search to determine the maximum value of k for which the statement $|w_0| \leq 2^j \mathbf{u}^k |z_0|$ holds. Then, we decrease the value of j until this statement becomes false, at which point we have identified the strongest bound provable in each abstract domain.

We observe that the SE, SETZ, and SELTZO abstractions exhibit stepwise increasing logical strength, with each model proving stronger relative error bounds for both ddadd

FPAN	SE	SETZ	SELTZO
ddadd	$2^{-(2p-7)} = 128\mathbf{u}^2$	$2^{-(2p-4)} = 16\mathbf{u}^2$	$2^{-(2p-2)} = 4\mathbf{u}^2$
madd	$2^{-(2p-6)} = 64\mathbf{u}^2$	$2^{-(2p-3)} = 8\mathbf{u}^2$	$2^{-(2p-1)} = 2\mathbf{u}^2$

Table 4.2: Strongest relative error bounds for ddadd and madd that are provable in the SE, SETZ, and SELTZO abstractions.

and madd than its predecessor. Notably, all of these bounds exhibit the same order of dependence on the unit roundoff $\mathbf{u}^k$, differing only in the leading constant 2^j. This suggests that the SE and SETZ abstractions may be useful to provide a coarse correctness check before initiating a more expensive SELTZO verification run.

Chapter 5

Synthesis

In the two previous chapters, we set up the basic theory of FPANs (Chapter 3) and formulated a procedure to automatically verify the FPAN correctness conditions (Chapter 4). In this chapter, we tackle the remaining question: how do we *find* candidate FPANs to verify?

As we have seen, reasoning about FPANs is a difficult combinatorial problem mired in exponential complexity, untamed by intuitive crutches or convenient shortcuts. There is often no clear high-level explanation as to why one FPAN works when another similar-looking FPAN fails. At our current state of knowledge, we are not aware of any intuitive guiding principles to steer us toward efficient or even correct FPANs. We therefore turn to a technique that is as simple as it is brutal: random evolutionary search.

5.1 Evolutionary Search

Consider the problem of finding an FPAN F that computes the sum of two floating-point expansions. For simplicity of notation, we assume that both addends, $\mathbf{x} \coloneqq (x_0, \dots, x_{n-1})$ and $\mathbf{y} \coloneqq (y_0, \dots, y_{n-1})$, and the sum $\mathbf{z} \coloneqq (z_0, \dots, z_{n-1})$ are strongly nonoverlapping floating-point expansions having the same length n. However, the search strategy that we develop in this section also works for mixed-length operations and alternative nonoverlapping conditions. Following the algorithmic template for addition presented in Section 3.4, we seek an FPAN F with $2n$ inputs and n outputs arranged in the following signature:

$$(z_0, \dots, z_{n-1}) \leftarrow F(x_0, y_0, \dots, x_{n-1}, y_{n-1}) \tag{5.1}$$

Note that the input expansions $\mathbf{x}$ and $\mathbf{y}$ are interlaced on the input wires of the FPAN F. This serves an important mathematical purpose that will be explained in Section 5.2.

It is sometimes necessary to consider the discarded rounding errors produced by the addition gates in F. We denote these discarded values by $\mathbf{w} \coloneqq (w_0, \dots, w_{n-1})$ and write

$$(z_0, \dots, z_{n-1}, w_0, \dots, w_{n-1}) \leftarrow F(x_0, y_0, \dots, x_{n-1}, y_{n-1}) \tag{5.2}$$

to denote a computation producing n output values $\mathbf{z} \coloneqq (z_0, \dots, z_{n-1})$, which are required to be strongly nonoverlapping, and n discarded values $\mathbf{w} \coloneqq (w_0, \dots, w_{n-1})$, which may overlap each other arbitrarily.

The only other requirement we place on the FPAN F is a relative error bound specified by some constant $\eta > 0$. We require the outputs $\mathbf{z}$ and discarded values $\mathbf{w}$ to satisfy

$$|w_0 + \cdots + w_{n-1}| \le \eta |z_0 + \cdots + z_{n-1}| \tag{5.3}$$

for all strongly nonoverlapping inputs $\mathbf{x}$ and $\mathbf{y}$. Since $\mathbf{z}$ is required to be nonoverlapping, z_0 is a close approximation of $z_0 + \cdots + z_{n-1}$, so in practice, we will often replace the right-hand side of Equation (5.3) simply by $\eta|z_0|$.

In principle, with all of our requirements specified, we could search for an FPAN that satisfies these requirements by generating random FPANs and checking their correctness with the SELTZO verification procedure developed in Chapter 4. However, this strategy is wildly ineffective. It is exceedingly rare to randomly stumble upon an FPAN that satisfies these requirements by blind luck. An effective search strategy requires some heuristic to bias the search toward promising candidates. Even though our SELTZO verification technique is millions of times faster than bit-blasting, it is still not fast enough to screen thousands or millions of candidate FPANs per second in the inner loop of a search algorithm.

Fortunately, one heuristic indicator of FPAN correctness is readily available: *testing!* Every pair of strongly nonoverlapping inputs $(\mathbf{x}, \mathbf{y})$ defines a *test case* that an FPAN F must pass. If the output $\mathbf{z} \coloneqq F(\mathbf{x}, \mathbf{y})$ fails to be strongly nonoverlapping or violates the relative error bound (5.3), then F can be eliminated from consideration. Of course, an FPAN must pass an infinite number of test cases to be formally correct, and there is no guarantee that any finite set of test cases is sufficient to completely determine correctness. Nonetheless, a large finite set of well-chosen test cases can serve as a useful heuristic filter to identify promising FPAN candidates.

This interplay between FPANs and test cases creates a sort of adversarial relationship that can be exploited to derive an effective search strategy. On one hand, we want to find efficient FPANs that compute accurate nonoverlapping results using as few gates as possible. However, using fewer gates tends to make an FPAN less robust, potentially introducing correctness issues. To detect these issues, we want difficult test cases that sniff out subtle failure modes to distinguish truly correct FPANs from subtly flawed ones. The presence of harder test cases strengthens the population of FPANs, while the presence of more FPANs creates more potential failure modes for test cases to find.

Algorithm 5: TwoSumRiffle$(x_0, \ldots, x_{n-1})$

```
Input: Floating-point numbers $x_0, \ldots, x_{n-1}$
Output: Floating-point numbers $x_0, \ldots, x_{n-1}$
1 for $i = 0$ to $\lfloor (n-2)/2 \rfloor$ do
2     $(x_{2i}, x_{2i+1}) \leftarrow$ TwoSum$(x_{2i}, x_{2i+1})$;
3 end
4 for $i = 0$ to $\lfloor (n-3)/2 \rfloor$ do
5     $(x_{2i+1}, x_{2i+2}) \leftarrow$ TwoSum$(x_{2i+1}, x_{2i+2})$;
6 end
7 return $(x_0, \ldots, x_{n-1})$
```

We therefore frame the problem of searching for an FPAN that satisfies our requirements as an adversarial evolutionary process in which a set of FPANs $\mathcal{F} \coloneqq \{F_i\}$ coevolves with a set of test cases $\mathcal{T} \coloneqq \{(\mathbf{x}_i, \mathbf{y}_i)\}$. We begin with an empty set of FPANs $\mathcal{F} \leftarrow \varnothing$ and a small seed population of, say, 20 randomly generated test cases obtained by calling Renormalize

Algorithm 6: Renormalize$(x_0, \ldots, x_{n-1})$

```
Input: Floating-point numbers $x_0, \ldots, x_{n-1}$
Output: A strongly nonoverlapping floating-point expansion $(x_0, \ldots, x_{n-1})$ whose
        real value is the exact sum of the inputs
1 repeat
2     $(x_0, \ldots, x_{n-1}) \leftarrow$ TwoSumRiffle$(x_0, \ldots, x_{n-1})$;
3 until $(x_0, \ldots, x_{n-1}) =$ TwoSumRiffle$(x_0, \ldots, x_{n-1})$;
4 return $(x_0, \ldots, x_{n-1})$;
```

(Algorithm 6) on random floating-point numbers. We then perform cyclically repeating phases of *FPAN generation*, *test case generation*, and *test case minimization*.

Algorithm 7: RelativeError$(F, (\mathbf{x}, \mathbf{y}))$

```
Input: FPAN $F$ with $2n$ wires, strongly nonoverlapping floating-point expansions
       $(\mathbf{x}, \mathbf{y})$
Output: Relative error of the strongly nonoverlapping sum computed by $F$, or
        $+\infty$ if the computed sum fails to be strongly nonoverlapping
1  $(\mathbf{z}, \mathbf{w}) \leftarrow F(\mathbf{x}, \mathbf{y})$;
2  if $\mathbf{z} \neq$ TwoSumRiffle$(\mathbf{z})$ then
3      return $+\infty$;
4  end
5  $\mathbf{w} \leftarrow$ Renormalize$(\mathbf{w})$;
6  if $z_0 = 0$ and $w_0 = 0$ then
7      return $0$;
8  else
9      return $|w_0 \oslash z_0|$;
10 end
```

In the FPAN generation phase, we use the current test case set $\mathcal{T}$ and relative error bound η to produce new candidate FPANs $F :=$ GenerateFPAN$(\mathcal{T}, \eta)$ (Algorithm 9). The greedy pruning step (lines 9–15 in Algorithm 9) ensures that the FPAN F produced by GenerateFPAN is minimal. We then call ImproveFPAN$(F, \mathcal{T}, \eta, t)$ (Algorithm 10) to explore the local neighborhood of F using a search strategy inspired by simulated annealing [10]. ImproveFPAN randomly proposes mutations of F that add, remove, or swap its TwoSum gates, accepting only mutations that produce an FPAN passing all test cases in $\mathcal{T}$. Initially,

Algorithm 8: PassesTestCases($F, \mathcal{T}, \eta$)

```
Input: FPAN $F$ with $2n$ wires, set of test cases $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$ , relative error
       bound $\eta > 0$
Output: Strongly nonoverlapping floating-point expansions $(\mathbf{x}, \mathbf{y})$ such that
        $F(\mathbf{x}, \mathbf{y})$ fails to be strongly nonoverlapping or has large relative error
1 for $(\mathbf{x}, \mathbf{y})$ in $\mathcal{T}$ do
2     if RelativeError($F, (\mathbf{x}, \mathbf{y})$) $> \eta$ then
3         return False;
4     end
5 end
6 return True;
```

Algorithm 9: GenerateFPAN($\mathcal{T}, \eta$)

```
Input: Set of test cases $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$, relative error bound $\eta > 0$
Output: FPAN with $2n$ wires that passes all test cases in $\mathcal{T}$
1 $F \leftarrow$ empty FPAN;
2 repeat
3     $i \leftarrow$ random integer between 0 and $2n - 1$;
4     $j \leftarrow$ random integer between 0 and $2n - 1$;
5     if $i \neq j$ then
6         append TwoSum gate to $F$ joining wire $i$ to wire $j$;
7     end
8 until PassesTestCases($F, \mathcal{T}, \eta$);
9 repeat
10    $F' \leftarrow F$;
11    remove a random gate from $F'$;
12    if PassesTestCases($F', \mathcal{T}, \eta$) then
13        $F \leftarrow F'$;
14    end
15 until removal of every gate has been tried;
16 return $F$;
```

Algorithm 10: ImproveFPAN($F, \mathcal{T}, \eta, t$)

```
Input: FPAN $F$ with $2n$ wires, set of test cases $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$, relative error
       bound $\eta > 0$, time bound $t$
Output: FPAN with $2n$ wires that passes all test cases in $\mathcal{T}$
1  $F_{\text{best}} \leftarrow F$;
2  $s \leftarrow 1$;
3  repeat
4      $p_{\text{insert}} \leftarrow 1/(1 + \sqrt{s} + \log s)$;
5      $p_{\text{delete}} \leftarrow \sqrt{s}/(1 + \sqrt{s} + \log s)$;
6      $p_{\text{swap}} \leftarrow \log s/(1 + \sqrt{s} + \log s)$;
7      $F' \leftarrow F$;
8      with probability $p_{\text{insert}}$ append a random TwoSum gate to $F'$;
9      with probability $p_{\text{delete}}$ delete a random TwoSum gate from $F'$;
10     with probability $p_{\text{swap}}$ swap two random TwoSum gates in $F'$;
11     if PassesTestCases($F', \mathcal{T}, \eta$) then
12         $F \leftarrow F'$;
13         if $F$ has fewer gates than $F_{best}$ then
14             $F_{\text{best}} \leftarrow F$;
15         end
16     end
17 until time $t$ has elapsed;
18 return $F_{best}$;
```

additions, removals, and swaps are proposed with roughly equal probability, encouraging exploration of the local neighborhood. Over time, the probability of adding gates is decreased and the probability of removing gates is increased, introducing a bias that drives the search toward more efficient FPANs with fewer gates. We call GenerateFPAN and ImproveFPAN several thousand times during each generation phase, using a time bound t of roughly 0.05 seconds. These calls to GenerateFPAN and ImproveFPAN can be performed in parallel across an arbitrarily number of processors, and all FPANs generated by this process are added to the set $\mathcal{F}$ in preparation for the next phase.

During the course of our evolutionary search algorithm, all FPANs consist only of TwoSum gates with no addition or FastTwoSum gates. We adopt the convention that the top n wires carry the output values $\mathbf{z}$ while the bottom n wires carry the discarded values $\mathbf{w}$. We only consider exchanging TwoSum gates for FastTwoSum or addition gates during offline analysis outside the evolutionary search algorithm.

Algorithm 11: GenerateTestCase(F, t)

```
Input: FPAN $F$ with $2n$ wires, time bound $t$
Output: Strongly nonoverlapping floating-point expansions $(\mathbf{x}, \mathbf{y})$ such that
        $F(\mathbf{x}, \mathbf{y})$ fails to be strongly nonoverlapping or has large relative error
1  bestErr $\leftarrow 0$;
2  result $\leftarrow ((0, \dots, 0), (0, \dots, 0))$;
3  repeat
4      $\mathbf{x} \leftarrow$ Renormalize($n$ random floating-point numbers);
5      $\mathbf{y} \leftarrow$ Renormalize($n$ random floating-point numbers);
6      relErr $\leftarrow$ RelativeError($F, (\mathbf{x}, \mathbf{y})$);
7      if relErr $= +\infty$ then
8          return $(\mathbf{x}, \mathbf{y})$;
9      else if relErr $>$ bestErr then
10         bestErr $\leftarrow$ relErr;
11         result $\leftarrow (\mathbf{x}, \mathbf{y})$;
12     end
13 until time $t$ has elapsed;
14 return result;
```

In the test case generation phase, we randomly pick an FPAN F from the size-depth Pareto frontier of the set $\mathcal{F}$, i.e., the subset of FPANs having minimal size for their depth

Algorithm 12: ImproveTestCase($F, (\mathbf{x}, \mathbf{y})$)

```
Input: FPAN $F$ with $2n$ wires, strongly nonoverlapping floating-point expansions
       $(\mathbf{x}, \mathbf{y})$
Output: Strongly nonoverlapping floating-point expansions $(\mathbf{x}, \mathbf{y})$ such that
        $F(\mathbf{x}, \mathbf{y})$ fails to be strongly nonoverlapping or has large relative error
1 bestErr $\leftarrow$ RelativeError($F, (\mathbf{x}, \mathbf{y})$);
2 repeat
3     $(\mathbf{x}', \mathbf{y}') \leftarrow (\mathbf{x}, \mathbf{y})$;
4     flip a random bit in the binary encoding of $(\mathbf{x}', \mathbf{y}')$;
5     if RelativeError($F, (\mathbf{x}', \mathbf{y}')$) $>$ bestErr then
6         $(\mathbf{x}, \mathbf{y}) \leftarrow (\mathbf{x}', \mathbf{y}')$;
7     end
8 until every bit flip has been tried with no improvement to bestErr;
9 return $(\mathbf{x}, \mathbf{y})$;
```

and minimal depth for their size. We then repeatedly call $(\mathbf{x}, \mathbf{y}) \coloneqq$ GenerateTestCase(F, t) (Algorithm 11) and ImproveTestCase($F, (\mathbf{x}, \mathbf{y})$) (Algorithm 12) to generate several dozen new test cases tailored to F. If one of these test cases causes F to fail, i.e., compute a result that is not strongly nonoverlapping or violates the relative error bound η, then F is removed from the set $\mathcal{F}$. We execute this process in parallel for several hundred FPANs chosen randomly from the Pareto frontier of $\mathcal{F}$, generating several thousand test cases in total, all of which are added to $\mathcal{T}$ in preparation for the next phase.

Finally, in the test case minimization phase, we test all FPANs in $\mathcal{F}$ against all new test cases added to $\mathcal{T}$. Any FPANs found to fail any new test case are removed from $\mathcal{F}$. We then shrink $\mathcal{T}$ by keeping only those test cases that maximize relative error for some FPAN. If multiple test cases achieve the same maximum relative error for a given FPAN, we run a greedy set cover algorithm to obtain a minimal subset of $\mathcal{T}$ that contains an error-maximizing test case for every FPAN in $\mathcal{F}$. This produces a minimal set of all the hardest test cases in $\mathcal{T}$ that will be used to generate the next generation of FPANs. We now repeat the cycle by returning to the FPAN generation phase.

Note that this search algorithm is *evolutionary* but not *genetic*. In particular, there is no notion of heredity between FPANs or test cases. Each new FPAN is created solely from

$\mathcal{T}$ with no knowledge or influence from other FPANs, and conversely, each new test case is created from an FPAN F on the Pareto frontier of $\mathcal{F}$ with no knowledge or influence from other test cases. The sets $\mathcal{F}$ and $\mathcal{T}$ coevolve only by interaction with each other without mutation or crossover dynamics internal to either set.

A permissively-licensed open-source implementation of this search algorithm is provided in the `https://github.com/dzhang314/ComparatorNetworks.jl` GitHub repository. Our implementation of Algorithms 5–11 uses SIMD acceleration to simultaneously evaluate 8 binary64 test cases on a single AVX-512-capable CPU core. In principle, TwoSumRiffle (Algorithm 5) could be implemented as a simple left-to-right sweep, but we found the riffle pattern to be significantly faster because it enables simultaneous superscalar dispatch of multiple vectorized TwoSum operations. To enable this optimization, it is important that the expansion length n be a compile-time constant so that these loops can be fully unrolled. Our implementation also uses JIT compilation to generate optimized native code for each FPAN in $\mathcal{F}$, significantly accelerating the test case generation and minimization phases.

Many aspects of our evolutionary search algorithm are the result of *ad hoc* design choices and preliminary exploration. We have not performed any systematic analysis of different search strategies and make no claim that the choices made in our search algorithms are in any way optimal. For example, the rate functions 1, $\sqrt{s}$, and $\log s$ that appear in Algorithm 10 are arbitrary choices that happened to work well in preliminary testing and have no deeper principled meaning. It is likely that further examination of these choices could produce significant performance improvements. Nonetheless, the evolutionary search algorithm described in this section is sufficient to discover all of the FPANs presented in this dissertation, which outperform state-of-the-art algorithms for extended-precision floating-point arithmetic by more than an order of magnitude.

5.2 Addition FPANs

By applying the evolutionary search algorithm developed in the previous section, we have discovered three novel branch-free algorithms for addition and subtraction of strongly

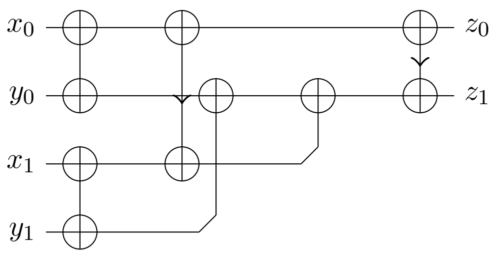


Figure 5.1: Provably optimal FPAN with size 6 and depth 4 for double-word addition. Here, (x_0, x_1) and (y_0, y_1) denote the input expansions to be added, and (z_0, z_1) denotes the output expansion. The relative error of sums computed by this FPAN is at most $2\mathbf{u}^2 + O(\mathbf{u}^3)$.

nonoverlapping floating-point expansions of lengths 2, 3, and 4. The first of these algorithms, shown in Figure 5.1, is the madd algorithm referenced in Section 4.5, with size and depth (6, 4). As previously noted, this strictly improves upon the size and depth (6, 5) of ddadd, the previous best known algorithm for double-word addition, while simultaneously reducing relative error from $3\mathbf{u}^2$ to $2\mathbf{u}^2$.

The corresponding FPANs for triple-word and quad-word arithmetic are shown in Figures 5.2 and 5.3, respectively, with size and depth (16, 10), and (31, 13). To our knowledge, these are first known algorithms for branch-free addition and subtraction of strongly nonoverlapping floating-point expansions of lengths 3 and 4. Prior algorithms for triple-word and quad-word arithmetic either used branching renormalization schemes [42, 30, 56] or failed to guarantee strongly nonoverlapping outputs for all possible inputs [63, 21, 99].

We have proven by exhaustive enumeration that madd is *the* optimal double-word addition algorithm. Every other FPAN with size up to 6 and depth up to 4 either fails to produce a nonoverlapping result or computes a sum whose relative error strictly exceeds $2\mathbf{u}^2$. Unfortunately, the exponential growth in the number of FPANs as a function of size makes exhaustive enumeration intractable for the triple-word and quad-word FPANs shown in Figures 5.2 and 5.3.

The three addition FPANs presented in this section (Figures 5.1–5.3) share the notable feature that they all begin with an initial layer of TwoSum gates that pair matching terms $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ between the two input expansions. Because TwoSum is a commutative operation, this structure guarantees that the sums computed by these FPANs are

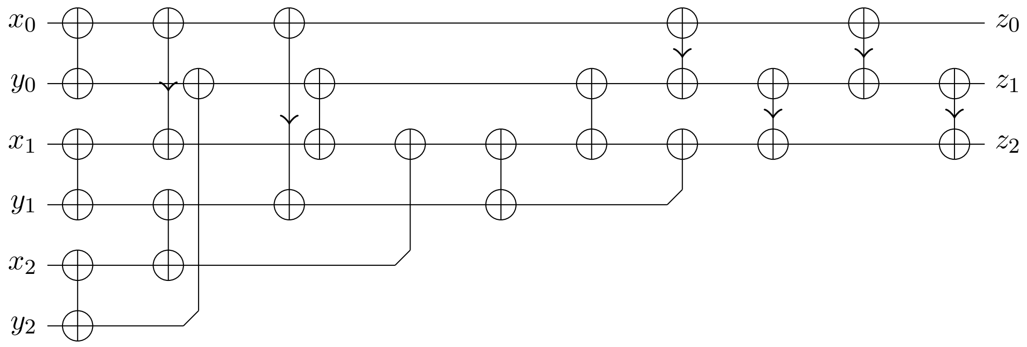


Figure 5.2: FPAN with size 16 and depth 10 for triple-word addition. Here, (x_0, x_1, x_2) and (y_0, y_1, y_2) denote the input expansions to be added, and (z_0, z_1, z_2) denotes the output expansion. The relative error of sums computed by this FPAN is at most $8\mathbf{u}^3 + O(\mathbf{u}^4)$.

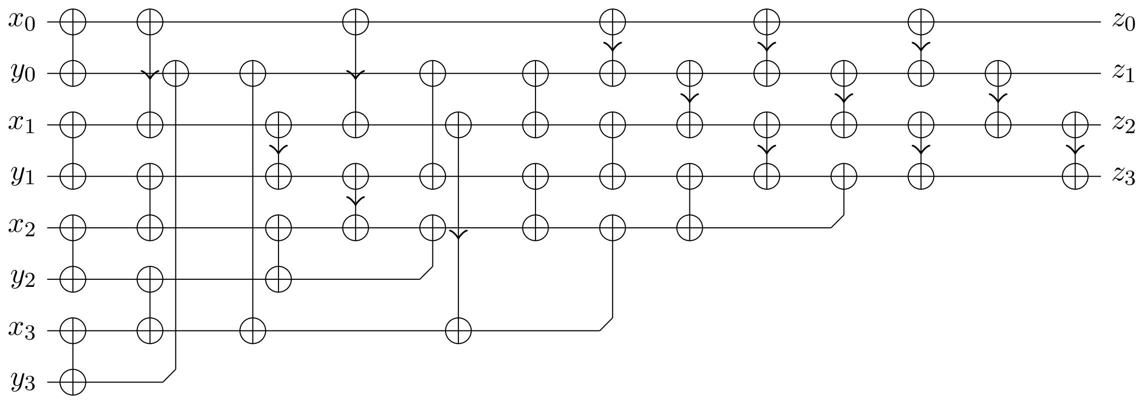


Figure 5.3: FPAN with size 31 and depth 13 for quad-word addition. Here, (x_0, x_1, x_2, x_3) and (y_0, y_1, y_2, y_3) denote the input expansions to be added, and (z_0, z_1, z_2, z_3) denotes the output expansion. The relative error of sums computed by this FPAN is at most $8\mathbf{u}^4+O(\mathbf{u}^5)$.

invariant under exchanging the inputs $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1})$. This is a desirable commutativity property that will be revisited in our subsequent discussion of multiplication.

5.3 Multiplication FPANs

We have also applied our evolutionary search algorithm to discover novel branch-free algorithms for multiplication of strongly nonoverlapping floating-point expansions. Recall from Section 3.4 that our strategy for FPAN-based multiplication first computes the pairwise error-free products $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$, then uses an FPAN to accumulate the terms $p_{i,j}$ and $e_{i,j}$. The FPANs shown in Figures 5.4, 5.5, and 5.6 show only the accumulation phase of the multiplication algorithm starting from the pairwise products $(p_{i,j}, e_{i,j})$.

The double-word multiplication FPAN with size and depth (3, 3) shown in Figure 5.4 was already known in prior work [54, Algorithm 10]. In fact, this is the original double-double multiplication algorithm proposed by Dekker in 1971 [25]. We merely include this algorithm for completeness and point out that it is also optimal by exhaustive enumeration.

In Figures 5.5 and 5.6, we present novel FPANs with size and depth (13, 8) and (33, 14) for *commutative* triple-word and quad-word multiplication. We emphasize commutativity here because some multiplication algorithms proposed in prior work violate the commutative property of multiplication [54, Algorithms 11–12]. In other words, they compute a different result when the inputs $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1})$ are swapped. Of course, it is well-known that floating-point arithmetic violates many of the algebraic properties enjoyed by exact real arithmetic, such as the associative and distributive properties. However, the lack of commutativity is particularly problematic in applications involving complex numbers because it causes the complex conjugate product $(a + bi)(a - bi)$ to have a small but nonzero imaginary part. This creates significant rounding artifacts that severely degrade the performance of certain numerical algorithms, such as eigensolvers.

In a similar fashion to the addition FPANs described in the previous section, we can enforce the commutative property in our multiplication FPANs by adding an initial layer of TwoSum gates that pair the symmetric values $(p_{i,j}, p_{j,i})$ and $(e_{i,j}, e_{j,i})$. For addition FPANs, this initial commutativity layer happens to naturally occur in the optimal FPANs discovered by our evolutionary search procedure. However, this does not naturally occur in multiplication FPANs, and we must deliberately impose the presence of the commutativity layer in our search procedure.

We observe that the search space for quad-word multiplication FPANs is unusually complicated, exhibiting a Pareto frontier that contains several thousand non-isomorphic FPANs all sharing the same size and depth. The FPAN we present in Figure 5.6 is derived from one member of this frontier. It is presently unclear whether this is an inherent mathematical property of quad-word multiplication or merely indicates that our evolutionary search

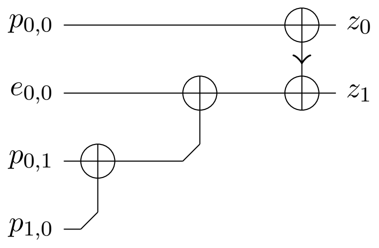


Figure 5.4: Provably optimal FPAN with size 3 and depth 3 for commutative double-word multiplication. Here, (x_0, x_1) and (y_0, y_1) denote the input expansions to be multiplied, $(p_{i,j}, e_{i,j}) := \mathsf{TwoProd}(x_i, y_j)$ denote the FPAN inputs, and (z_0, z_1) denotes the output expansion. The relative error of products computed by this FPAN is at most $8\mathbf{u}^2 + O(\mathbf{u}^3)$.

algorithm has not yet converged.

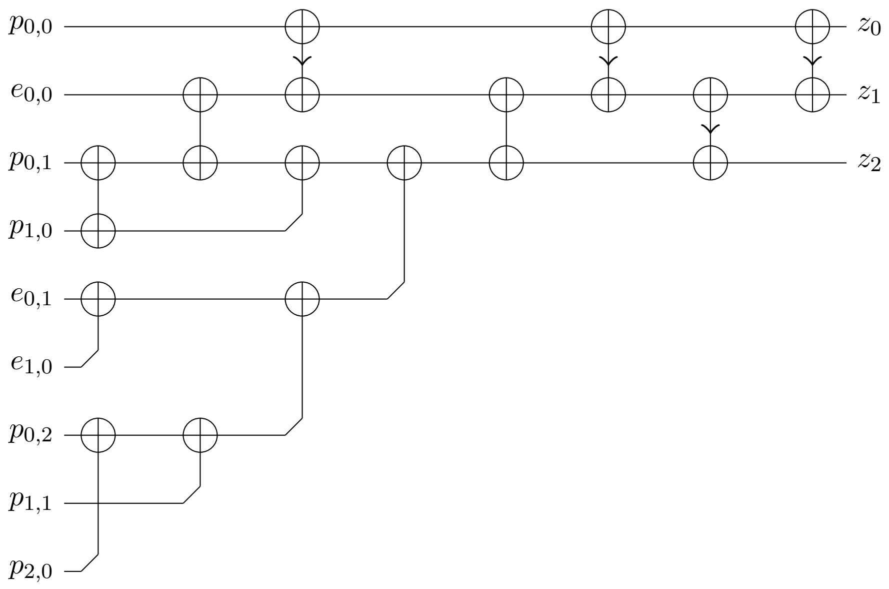


Figure 5.5: FPAN with size 13 and depth 8 for commutative triple-word multiplication. Here, (x_0, x_1, x_2) and (y_0, y_1, y_2) denote the input expansions to be multiplied, $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$ denote the FPAN inputs, and (z_0, z_1, z_2) denotes the output expansion. The relative error of products computed by this FPAN is at most $64\mathbf{u}^3 + O(\mathbf{u}^4)$.

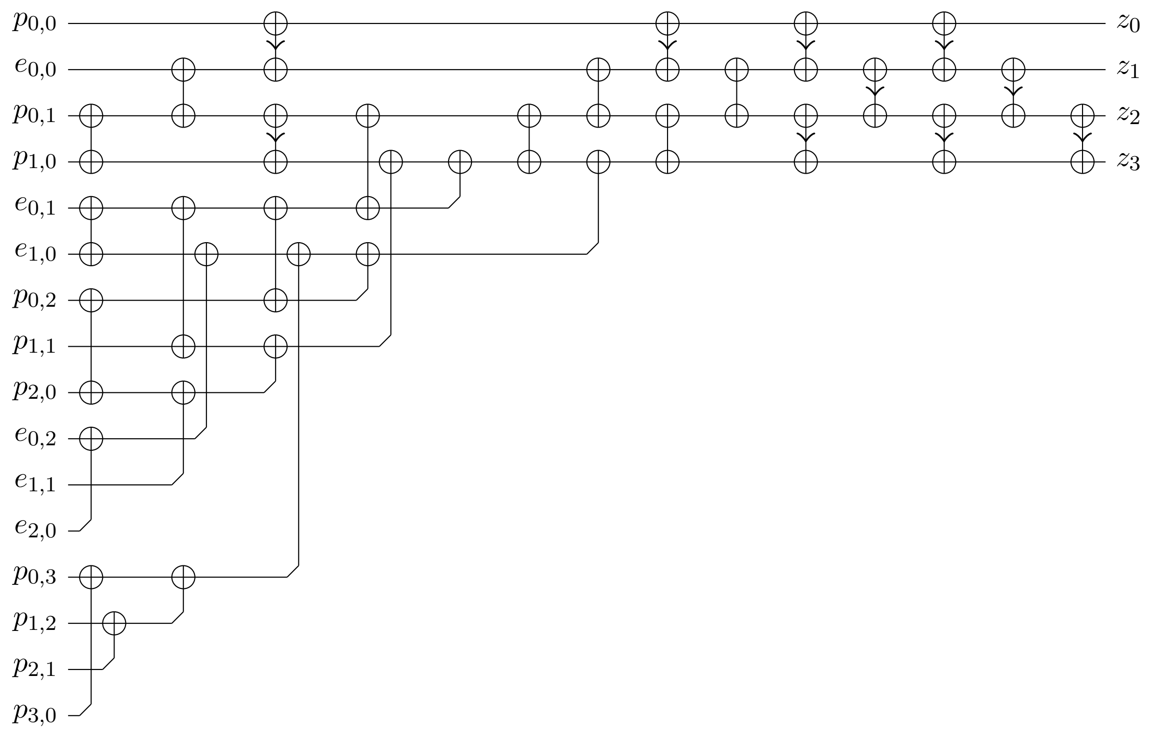


Figure 5.6: FPAN with size 31 and depth 14 for commutative quad-word multiplication. Here, (x_0, x_1, x_2, x_3) and (y_0, y_1, y_2, y_3) denote the input expansions to be multiplied, $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$ denote the FPAN inputs, and (z_0, z_1, z_2, z_3) denotes the output expansion. The relative error of products computed by this FPAN is at most $256\mathbf{u}^4 + O(\mathbf{u}^5)$.

Chapter 6

Evaluation

To assess the performance of our new algorithms in a practical scientific computing context, we used them to implement four extended-precision BLAS kernels that exercise typical computational patterns found in scientific software.

- AXPY: vector-vector operations
- DOT: vector-vector reduction operations
- GEMV: matrix-vector operations
- GEMM: matrix-matrix operations

We developed *MultiFloats*, a prototype C++ library that implements these BLAS kernels on two-term (quadruple precision), three-term (sextuple precision), and four-term (octuple precision) expansions using the addition and multiplication FPANs shown in Figures 5.1–5.6. Our library provides a class template `MultiFloat<T,N>` parameterized by an underlying floating-point type `T` and a floating-point expansion length $\texttt{N} = 1, 2, 3, 4$. (`MultiFloat<T,1>` is simply an alias for `T`.)

Allowing the user to select the underlying floating-point type `T` significantly enhances the portability of our library. For example, datatypes like `MultiFloat<float,4>` can be used to provide extended-precision arithmetic on machines that lack double-precision hardware. On the other hand, processors with native support for IEEE quadruple precision can use `MultiFloat<quad,2>` to provide fast octuple-precision arithmetic. Nonetheless, we expect `MultiFloat<T,N>` to be used with `T` = `double` on the vast majority of current high-performance computer hardware.

We compared the performance of our library, MultiFloats, to the following suite of extended-precision arithmetic libraries:

- GMP 6.3.0 [38]
- MPFR 4.2.1 [32]
- FLINT (formerly known as Arb) 3.2.1 [40, 53]
- Boost.Multiprecision 1.86 [68]
- QD[1] 2.3.23 [4]
- CAMPARY[2] 01.06.17 [55]
- libquadmath[3] 14.2 [37]

The latest version of each library available at the time of writing was selected for testing. This is an exhaustive list of all extended-precision floating-point libraries that we are aware of, excluding (1) libraries for base-10 floating-point arithmetic, such as mpdecimal; (2) libraries that merely wrap the interface of another library, such as MPFR++ and mppp; (3) libraries that are not thread-safe, such as CLN; (4) libraries targeting dynamic languages, such as mpmath and bignumber.js; and (5) unmaintained libraries that no longer compile on modern hardware, including CUMP and MPRES-BLAS. We also exclude XBLAS [64] from consideration because its interface does not allow extended-precision numbers to be passed into or out of the library.

[1]QD only supports two-term and four-term floating-point expansions.

[2]CAMPARY provides two sets of arithmetic algorithms: a "certified" set that is provably correct but uses branching, and a "fast" set that is branch-free but known to be incorrect on some classes of inputs. In some cases, the "fast" algorithms exhibit catastrophic loss of precision, degrading the accuracy of the result to machine precision. We benchmark only the "certified" algorithms to provide a fair comparison to our algorithms, which are provably correct on all inputs.

[3]libquadmath is the library used to provide the built-in `__float128` type in the GCC and Clang compilers. It only supports IEEE quadruple-precision arithmetic and does not provide any other precision levels.

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**135.22**	**35.35**	**11.32**	**5.60**
GMP	0.67	0.64	0.63	0.63
MPFR	1.45	1.13	0.75	0.50
FLINT	1.39	1.01	0.86	0.79
Boost.Multiprecision	1.33	0.61	0.36	0.33
QD	N/A	24.13	N/A	0.50
CAMPARY	133.80	32.44	0.35	0.24
libquadmath	N/A	1.05	N/A	N/A

Table 6.1: Measured AXPY performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 16-core AMD Zen 5 CPU (Ryzen 9 9950X). "N/A" entries indicate lack of library support for a specific precision level.

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**117.35**	**30.87**	**11.75**	**5.77**
GMP	0.65	0.64	0.64	0.63
MPFR	1.44	1.16	0.78	0.55
FLINT	1.62	1.21	1.00	0.92
Boost.Multiprecision	1.40	0.63	0.34	0.32
QD	N/A	4.66	N/A	0.51
CAMPARY	52.84	5.40	0.36	0.25
libquadmath	N/A	1.13	N/A	N/A

Table 6.2: Measured DOT performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 16-core AMD Zen 5 CPU (Ryzen 9 9950X). "N/A" entries indicate lack of library support for a specific precision level.

To ensure optimal conditions for fair comparison, each combination of library and BLAS kernel was compiled using the latest available GCC (version 14.2) and Clang (version 20.1) C++ compilers, using both medium (`-O2`) and full (`-O3`) optimization levels, with all available ISA extensions (`-march=native`) enabled on the most recent high-performance processor microarchitectures (AMD Zen 5 and Apple M3) available to us at the time of writing. All BLAS kernels were implemented with identical parallelization strategies, using `ij` loop ordering for GEMV and `ikj` loop ordering for GEMM. In addition, each kernel was run in both thread-per-physical-core and thread-per-logical-core configurations using the OpenMP thread affinity API (`OMP_PROC_BIND`).

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**225.18**	**38.87**	**12.14**	**5.86**
GMP	0.66	0.66	0.66	0.64
MPFR	1.51	1.21	0.79	0.59
FLINT	1.63	1.22	0.98	0.90
Boost.Multiprecision	1.34	0.63	0.38	0.33
QD	N/A	4.68	N/A	0.51
CAMPARY	58.65	5.32	0.36	0.25
libquadmath	N/A	1.12	N/A	N/A

Table 6.3: Measured GEMV performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 16-core AMD Zen 5 CPU (Ryzen 9 9950X). "N/A" entries indicate lack of library support for a specific precision level.

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**328.98**	**42.18**	**12.34**	**5.93**
GMP	0.62	0.61	0.61	0.60
MPFR	1.50	1.18	0.79	0.55
FLINT	1.61	1.22	1.01	0.94
Boost.Multiprecision	1.30	0.63	0.37	0.31
QD	N/A	26.47	N/A	0.51
CAMPARY	310.29	37.42	0.36	0.25
libquadmath	N/A	1.13	N/A	N/A

Table 6.4: Measured GEMM performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 16-core AMD Zen 5 CPU (Ryzen 9 9950X). "N/A" entries indicate lack of library support for a specific precision level.

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**328.98**	**42.18**	**12.34**	**5.93**
GMP	0.62	0.61	0.61	0.60
MPFR	1.50	1.18	0.79	0.55
FLINT	1.61	1.22	1.01	0.94
Boost.Multiprecision	1.30	0.63	0.37	0.31
QD	N/A	26.47	N/A	0.51
CAMPARY	310.29	37.42	0.36	0.25
libquadmath	N/A	1.13	N/A	N/A

Table 6.5: Measured AXPY performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 12-core ARMv8.6-A CPU (Apple M3 Pro). "N/A" entries indicate lack of library support for a specific precision level.

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**12.50**	**1.19**	**0.52**	**0.31**
GMP	0.16	0.16	0.16	0.16
MPFR	0.73	0.66	0.43	0.25
FLINT	0.44	0.30	0.27	0.23
Boost.Multiprecision	0.62	0.34	0.18	0.15
QD	N/A	1.16	N/A	0.17
CAMPARY	6.81	0.94	0.24	0.16
libquadmath	N/A	N/A	N/A	N/A

Table 6.6: Measured DOT performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 12-core ARMv8.6-A CPU (Apple M3 Pro). "N/A" entries indicate lack of library support for a specific precision level.

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**15.59**	**1.26**	**0.51**	**0.34**
GMP	0.16	0.16	0.16	0.16
MPFR	0.78	0.68	0.42	0.25
FLINT	0.45	0.32	0.27	0.23
Boost.Multiprecision	0.59	0.33	0.18	0.15
QD	N/A	1.16	N/A	0.17
CAMPARY	8.95	0.95	0.25	0.14
libquadmath	N/A	N/A	N/A	N/A

Table 6.7: Measured GEMV performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 12-core ARMv8.6-A CPU (Apple M3 Pro). "N/A" entries indicate lack of library support for a specific precision level.

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**46.53**	**6.78**	**2.02**	**0.98**
GMP	0.16	0.16	0.16	0.16
MPFR	0.84	0.69	0.45	0.25
FLINT	0.48	0.32	0.27	0.25
Boost.Multiprecision	0.61	0.32	0.18	0.14
QD	N/A	2.76	N/A	0.17
CAMPARY	41.10	4.77	0.27	0.19
libquadmath	N/A	N/A	N/A	N/A

Table 6.8: Measured GEMM performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 12-core ARMv8.6-A CPU (Apple M3 Pro). "N/A" entries indicate lack of library support for a specific precision level.

In Tables 6.1–6.8, we report the maximum AXPY, DOT, GEMV, and GEMM performance achieved by each library on one-term (double precision), two-term (quadruple precision), three-term (sextuple precision), and four-term (octuple precision) floating-point expansions. For libraries not based on floating-point expansions, we statically specified an equivalent number of bits of precision (53, 103, 156, and 208 bits, respectively) based on experimentally measured relative error bounds for our FPAN-based algorithms. The numbers reported in these tables represent the maximum computational throughput, in billions of extended-precision operations per second, achieved over all possible choices of compiler, optimization level, and thread count. To eliminate the effect of memory bandwidth, we measured performance on the largest matrix and vector sizes that each library can fit into L3 cache.

We adopt the usual convention in numerical linear algebra that one arithmetic operation consists of one multiplication followed by one addition [70]. Thus, given vectors of size n and matrices of size $n \times n$, AXPY and DOT execute n operations, GEMV executes n^2 operations, and GEMM executes n^3 operations. Note that each extended-precision operation conists of several dozen to several hundred native machine-precision FLOPs.

On AMD Zen 5, our new FPAN-based algorithms significantly outperformed all competing libraries in all benchmarks, often by more than an order of magnitude. Only two libraries, QD and CAMPARY, achieved comparable AXPY and GEMM performance in the two-term case by using previously known, albeit suboptimal, branch-free algorithms. They are unable to match our two-term DOT and GEMV performance because they do not provide SIMD reduction operators and their code is too complex for either GCC 14.2 or Clang 20.1 to automatically vectorize. Moreover, at three-term (156-bit) and four-term (208-bit) precision levels, no competing libraries come within a factor of $10\times$ or $5\times$ of our algorithms in any of the four tested kernels. We also observe that our algorithms exhibit a modest but consistent trend of increasing computational throughput across vector-vector operations (AXPY and DOT), matrix-vector operations (GEMV), and matrix-matrix operations (GEMM), representing different points on a roofline curve.

On Apple M3, our FPAN-based algorithms also outperformed all competing libraries

Kernel	1-Term	2-Term	3-Term	4-Term
AXPY	44.25	21.63	15.77	9.71
DOT	84.83	56.72	38.14	28.44
GEMV	170.77	92.37	28.42	31.92
GEMM	466.43	277.37	170.50	81.11

Table 6.9: Measured GPU performance, in billions of extended-precision operations per second, of our FPAN-based algorithms on an AMD RDNA 3 GPU (RX 7900 XTX).

in all benchmarks, though the ratio of improvement is less dramatic. Compared to AMD Zen 5, this architecture deprioritizes SIMD performance (128-bit NEON vs. 512-bit AVX), so efficiently-vectorizable branch-free algorithms experience a smaller performance uplift compared to branching scalar code. Nonetheless, our algorithms are still consistently the fastest, and some order-of-magnitude improvements over existing libraries are still observed.

Finally, in Table 6.9, we report the performance of our algorithms on an AMD RDNA3 GPU using the ROCm 6.4.1 toolchain. Unlike our CPU benchmarks, our GPU implementation uses `T = float` as the underlying base type instead of `T = double` because this architecture lacks double precision units. We observe significant performance uplift over CPUs, particularly for high-precision GEMM operations, which are more than an order of magnitude faster. These experiments demonstrate that the branch-free nature of our algorithms makes them highly suitable for GPUs, in addition to their utility for extending the precision of single-precision hardware.

Chapter 7

Conclusion

In this dissertation, we have introduced a new approach to extended-precision floating-point arithmetic that significantly outperforms all existing software libraries while providing stronger correctness guarantees. These advances have been made possible through a combination of three novel technical contributions:

- the introduction of floating-point accumulation networks (FPANs), which our work identifies as key algorithmic primitives that enable branch-free algorithms for extended-precision floating-point arithmetic (Chapter 3);
- the SELTZO abstraction, which automates formal verification of the FPAN correctness conditions, eliminating the possibility of missing cases and other subtle mistakes in traditional pen-and-paper rounding error analyses (Chapter 4); and
- an evolutionary search metaheuristic that systematically explores the space of all FPANs to find the fastest correct algorithm for a given task (Chapter 5).

By combining these techniques, we have discovered five novel branch-free algorithms for addition (Section 5.2) and multiplication (Section 5.3) of strongly nonoverlapping floating-point expansions with two, three, or four terms, which in turn, yield new branch-free algorithms for subtraction, division, and square root (Section 3.4). Our new FPAN-based algorithms significantly outperform all state-of-the-art software libraries for high-precision floating-point arithmetic by factors of 11.7×–69.3× (Chapter 6), opening new frontiers in high-performance large-scale computational modeling and simulation.

To encourage adoption of our algorithms and independent verification of our proofs by other researchers in numerical analysis and scientific computing, we have released all of our verification, search, benchmarking, and optimization tools under permissive open-source licenses in the following GitHub repositories:

- `https://github.com/dzhang314/FPANVerifier`
- `https://github.com/dzhang314/ComparatorNetworks.jl`
- `https://github.com/dzhang314/MultiprecisionBenchmarks`

We have taken substantial care to design clean, well-documented interfaces that make our software as flexible, extensible, and reusable as possible without sacrificing performance.

7.1 Related Work

Floating-point arithmetic is a fundamental and far-reaching topic that has been studied in its modern form by applied mathematics and computer scientists for over sixty years [97]. In fact, the basic underlying concept of an inexact positional number system, consisting of sequences of digits scaled by powers of a base, was known to Sumerian and Babylonian mathematicians in the third century BCE [19]. It is hardly surprising that a topic so classical and so widely studied, of broad interest to all scientists and engineers, has been approached from many angles in a large body of related work. In this section, we situate the findings of this dissertation in the broader landscape of research on floating-point arithmetic.

Double-double, triple-double, and quad-double arithmetic. The prior works most directly comparable to the approach presented in this dissertation are the existing algorithms for double-double [25, 64], triple-double [30, 63], and quad-double [42, 65] arithmetic. These algorithms also implement addition, subtraction, multiplication, division, and square root on fixed-length floating-point expansions. However, to our knowledge, all such algorithms either include an expensive branching renormalization step or fail to guarantee strongly nonoverlapping outputs for all strongly nonoverlapping inputs. In many cases, these prior algorithms have weak or conjectural correctness claims that are stated without proof. This lack of formal guarantees makes these algorithms risky to apply in large-scale computational workloads where rare rounding edge cases are more likely to occur.

Our FPAN-based arithmetic algorithms are the first known branch-free algorithms to provably preserve strong nonoverlapping with rigorous relative error bounds. This novel combination of speed and correctness makes our algorithms uniquely suited to the demands of high-performance scientific computation.

Adaptive floating-point expansions. Another family of techniques, developed in work by Priest [82] and Shewchuk [89], uses floating-point expansions of *dynamic* length to iteratively refine a computation until a specified error tolerance is met. For example, suppose we want to determine whether a point lies inside a circle. In most cases, a machine-precision computation is provably sufficient, but additional precision is necessary when the point lies on or near the boundary of the circle. Adaptive methods retry the computation using floating-point expansions of increasing length until the rounding errors are small enough to conclusively determine the answer.

Adaptive-length floating-point expansions are most useful for simple calculations whose rounding errors can be analyzed to derive provably sufficient error bounds. Their inherently branching nature prevents effective SIMD parallelization. These properties make adaptive methods suitable for computational geometry tasks, such as circle containment and ray-triangle intersection, but not for heavy scientific computing workloads.

Compensated algorithms. Beyond FPANs, error-free transformations are also employed in a class of floating-point algorithms called *compensated algorithms*, such as Kahan–Babuška–Neumaier summation [58, 2, 76]. Unlike floating-point expansions, which involve a fixed number of terms, these algorithms operate on a variable number of inputs and only perform partial tracking and correction of rounding errors, making no attempt to satisfy rigorous worst-case error bounds.

Other approaches to high-precision arithmetic. Libraries for arbitrary-precision arithmetic, including GMP, MPFR, and FLINT, make no internal use of floating-point operations [38, 32, 40, 53]. Instead, they implement arithmetic purely in terms of digit-by-digit integer operations. This approach allows for truly arbitrary precision, unconstrained by floating-point overflow and underflow limits, and avoids the complexity of propagating rounding errors that accompanies the use of error-free transformations. However, at practical extended precision levels (2–8 machine words), these algorithms are many times slower than FPANs, requiring complex branching logic that precludes efficient data-parallel execution. While FPANs are hard to discover and prove correct, they enable high-performance branch-free arithmetic that massively accelerates high-precision scientific applications.

Interactive floating-point verification. As discussed in Chapter 4, error-free transformations are particularly difficult for existing floating-point verification methods to handle. To our knowledge, the only formal reasoning techniques that were successfully applied to error-free transformations before our work used *interactive*, rather than automatic, theorem provers. Interactive verification tools, such as Flocq [13] and Gappa [23], have been used to prove the correctness of algorithms involving error-free transformations [22, 91], but these tools demand a high degree of user expertise to construct sophisticated proof scripts. This requires the user to manually split the verification task into tractable cases and correctly identify the lemmas needed to resolve each case. The SELTZO abstraction enables an SMT solver to automate the tedious processes of case management and lemma application.

Scalable abstraction in other domains. Recent work on the Bitwuzla SMT solver [78] has used lemmas for integer multiplication and division to accelerate bit vector verification, enabling scalability to previously intractable bit widths. The SELTZO abstraction can be thought of as a floating-point analogue of this approach, characterizing the TwoSum operation in a precision-independent fashion to avoid full-width bit-blasting. Unlike bit vector methods, our approach does not require abstraction refinement tailored to a specific FPAN or correctness condition.

Sorting networks. FPANs are closely related to sorting networks, and the graphical FPAN notation presented in this dissertation is heavily inspired by the diagrammatic representation of sorting networks [61]. Although they compute different operations, both are branch-free algorithms that sort or accumulate a fixed number of inputs by performing pairwise operations in a data-parallel fashion. This close relationship inspires many natural research questions connecting the well-established theory of sorting networks to the relatively unexplored theory of FPANs, which are discussed in the following section.

Program synthesis. Our method for discovering and verifying FPANs is an example of search-based program synthesis, a family of methods for using a search procedure to discover a program that satisfies both correctness and performance requirements. Search-based synthesis methods have been used to superoptimize assembly code [69, 88], deep learning computations [52, 52], cryptographic primitives [62], and quantum algorithms [98, 81]. These techniques combine a fast heuristic search that uses testing to identify plausible candidate programs with a full formal verification procedure that confirms whether the candidate is correct. Our method uses a different search procedure than previous work (simulated annealing) combined with a novel and highly elaborate verifier [100].

7.2 Future Work

The techniques introduced in this dissertation solve previously intractable classes of problems in numerical analysis and formal verification, opening many natural lines of inquiry in the development of floating-point algorithms. In this section, we propose directions for future work, ranging from straightforward applications that exploit the strengths of our tools to deep theoretical questions that explore the fundamental capabilities and limitations of FPANs and the SELTZO abstraction.

FPANs for longer expansion lengths. The most obvious limitation of the algorithms presented in this dissertation is that they only apply to floating-point expansions of length up to four. Discovering addition and multiplication FPANs for expansions of length five and beyond is an immediate next step that would likely yield similarly dramatic performance improvements over all existing algorithms. We expect this task to be computationally expensive but not fundamentally out of reach of our tools. Our addition and multiplication FPANs for expansions of length four took several thousand iterations of our evolutionary search procedure, executed in parallel across hundreds of CPU cores, to discover.

Exploring weaker nonoverlapping conditions. All of the algorithms presented in this dissertation assume strongly nonoverlapping inputs and produce strongly nonoverlapping outputs. It is natural to ask whether strong nonoverlapping can be replaced with a weaker alternative condition, such as ulp-nonoverlapping, $\mathcal{P}$-nonoverlapping, or $\mathcal{S}$-nonoverlapping, to produce simpler or faster algorithms. The SELTZO abstraction is capable of expressing these alternative nonoverlapping conditions (Proposition 9), but a modification of Algorithm 11 is necessary to generate test inputs that exercise these weaker preconditions.

Characterizing and applying the SELTZO abstraction. As discussed in Chapter 4, the SELTZO abstraction trades off some logical power in favor of efficient automatic verification. To understand the limitations of the SELTZO abstraction, it would be useful to precisely characterize the gap in logical strength between the SELTZO domain and the true domain of concrete floating-point numbers. This could take the form of a statement which is true in the concrete floating-point domain but false in the SELTZO domain or vice versa. It may also be fruitful to explore whether the SELTZO verification procedure can be applied to other floating-point verification problems beyond FPANs.

Simpler FPAN correctness conditions. The verification of sorting networks, which are diagrammatic algorithms closely analogous to FPANs, is considerably simplified by the *0-1 Principle* [61], which says that a sorting network is correct on all inputs if and only if it is correct on inputs containing only zeros and ones. A similar sufficient condition for the correctness of FPANs could dramatically speed up FPAN verification, especially if it is stated in the language of the SELTZO abstraction. Alternative formulations of the FPAN correctness conditions may also be useful for proving lower bounds on the necessary size or depth for an FPAN to preserve a certain nonoverlapping invariant or achieve a certain relative error bound. These lower bounds would be useful to determine how far our current FPAN-based algorithms are from being truly optimal.

Optimal FPAN instruction selection. In a recent blog post [80], Pavel Pachenka proposes an alternative implementation strategy for $\mathsf{TwoSum}(x, y)$ that simultaneously computes both $\mathsf{FastTwoSum}(x, y)$ and $\mathsf{FastTwoSum}(y, x)$, compares $|x|$ to $|y|$, and uses a conditional move instruction to choose the correct output based on the result of the comparison. This alternative TwoSum algorithm executes more instructions than the conventional algorithm but exposes more instruction-level parallelism, yielding lower latency on superscalar processors. Hence, every TwoSum gate presents an opportunity to optimize for either throughput (using the conventional algorithm) or latency (using Pachenka's alternative algorithm). In some FPANs, it may even be preferable to mix latency-optimized and throughput-optimized TwoSum gates depending on the presence of a critical path.

FPAN compilation and library tuning. In addition to the instruction selection problem described above, FPANs also present other low-level optimization challenges, including instruction scheduling and register allocation. These issues are particularly pronounced in large FPANs with many parallel wires and gates, which involve an unusually large number of temporary variables compared to typical numerical programs. Compiler heuristics tuned for other classes of programs may therefore be suboptimal for FPANs. A dedicated FPAN compiler could be a useful tool to explore this design space and optimize over architecture-dependent factors, and tuned libraries of FPAN-based algorithms, such as high-precision BLAS libraries [64], would be very useful tools for practitioners solving high-precision, large-scale, and/or ill-conditioned scientific computing problems.

FPAN specialization and transcendental functions. The FPANs presented in this dissertation were constructed and verified to be correct for all strongly nonoverlapping inputs. It may be possible to derive simpler or faster algorithms in situations where the class of possible inputs is further restricted. For example, squaring or cubing a single number is a more restricted problem than multiplying two distinct numbers.

An important application where these restrictions arise is the evaluation of transcendental functions, such as trigonometric functions, exponentials, and logarithms. These are often implemented as piecewise polynomial or rational approximations where the coefficients of the polynomial or rational approximant are known in advance. In these situations, adding and multiplying specific known coefficients is a simpler operation that may admit further optimization beyond addition and multiplication of two arbitrary numbers.

Appendix A

SETZ Lemmas

Let x and y be floating-point numbers, and let $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$. Let $(s_x, e_x, \mathsf{ntz}_x)$, $(s_y, e_y, \mathsf{ntz}_y)$, $(s_s, e_s, \mathsf{ntz}_s)$, and $(s_e, e_e, \mathsf{ntz}_e)$ denote the SETZ abstractions of x, y, s, and e, respectively. The following lemmas completely characterize the TwoSum operation in the SETZ abstract domain by specifying all possible SETZ values of s and e given the SETZ values of x and y.

These lemmas have been verified by exhaustive enumeration of the binary16 and bfloat16 floating-point formats to be stated in the strongest possible form. In other words, every element of the set of allowed SETZ output tuples listed in each lemma is actually witnessed by some pair of concrete floating-point inputs (x, y) satisfying the hypotheses of the lemma.

Since TwoSum is a commutative operation (i.e., $\mathsf{TwoSum}(x, y) = \mathsf{TwoSum}(y, x)$), each lemma remains true when x and y are interchanged. As in Section 4.3, we state only one member of each symmetric lemma pair to avoid needless repetition, adopting the convention that we prefer the lemma statement with $e_x \geq e_y$ whenever possible.

Many SETZ lemmas admit simpler statements if we make a change of variables from $(s_x, e_x, \mathsf{ntz}_x)$ to (s_x, e_x, f_x) where $f_x \coloneqq e_x - (p - \mathsf{ntz}_x - 1)$. We call this value the *trailing exponent* of x. The trailing exponent is the place value of the last nonzero bit in the mantissa of x. It acts as the dual of e_x, bounding the mantissa from below.

Lemma Family SETZ-Z

Lemmas in Family SETZ-Z apply when one or both of the inputs (x, y) are zero.

Lemma SETZ-Z1:

$$\mathsf{TwoSum}(+0.0, +0.0) = (+0.0, +0.0)$$
$$\mathsf{TwoSum}(+0.0, -0.0) = (+0.0, +0.0)$$
$$\mathsf{TwoSum}(-0.0, -0.0) = (-0.0, +0.0)$$

Lemma SETZ-Z2: If x is nonzero, then $\mathsf{TwoSum}(x, \pm 0.0) = (x, +0.0)$.

We henceforth assume that x and y are both nonzero in all remaining SETZ lemmas.

Lemma Family SETZ-I

Lemma SETZ-I (for "identical") gives necessary and sufficient conditions for the inputs (x, y) to be returned unchanged by TwoSum.

Lemma SETZ-I: $(s, e) = (x, y)$ if and only if any of the following conditions hold:

1. $e_x > e_y + (p + 1)$.
2. $e_x = e_y + (p + 1)$ and any of the following conditions hold: $e_y = f_y$, $s_x = s_y$, or $e_x > f_x$.
3. $e_x = e_y + p$, $e_y = f_y$, $e_x < f_x + (p - 1)$, and $s_x = s_y$ or $e_x > f_x$.

Lemma Family SETZ-F

Lemmas in Family SETZ-F apply to addends with the same trailing exponent (i.e., $f_x = f_y$).

Lemma SETZ-FS0: If $s_x = s_y$, $f_x = f_y$, and $e_x > e_y + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_x + 1 \leq f_s \leq e_x - 1$, and $e = +0.0$.

2. $s_s = s_x$, $e_s = e_x + 1$, $f_x + 1 \leq f_s \leq e_y$, and $e = +0.0$.

3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, and $e = +0.0$.

Lemma SETZ-FS1: If $s_x = s_y$, $f_x = f_y$, and $e_x = e_y + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_x + 1 \leq f_s \leq e_x - 2$, and $e = +0.0$.

2. $s_s = s_x$, $e_s = e_x + 1$, $f_x + 1 \leq f_s \leq e_y$, and $e = +0.0$.

3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, and $e = +0.0$.

Lemma SETZ-FS2: If $s_x = s_y$, $f_x = f_y$, $e_x = e_y$, and $e_x > f_x$, then $s_s = s_x$, $e_s = e_x + 1$, $f_x + 1 \leq f_s \leq e_x$, and $e = +0.0$.

Lemma SETZ-FS3: If $s_x = s_y$, $f_x = f_y$, $e_x = e_y$, and $e_x = f_x$, then $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, and $e = +0.0$.

Lemma SETZ-FD0: If $s_x \neq s_y$, $f_x = f_y$, and $e_x > e_y + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $f_x + 1 \leq f_s \leq e_y$, and $e = +0.0$.

2. $s_s = s_x$, $e_s = e_x$, $f_x + 1 \leq f_s \leq e_x$, and $e = +0.0$.

Lemma SETZ-FD1: If $s_x \neq s_y$, $f_x = f_y$, and $e_x = e_y + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $f_x + 1 \leq e_s \leq e_x - 1$, $f_x + 1 \leq f_s \leq e_s$, and $e = +0.0$.

2. $s_s = s_x$, $e_s = e_x$, $f_x + 1 \leq f_s \leq e_x - 2$, and $e = +0.0$.

3. $s_s = s_x$, $e_s = e_x$, $f_s = e_x$, and $e = +0.0$.

Lemma SETZ-FD2: If $s_x \neq s_y$, $f_x = f_y$, and $e_x = e_y$, then exactly one of the following statements is true:

1. $s = +0.0$ and $e = +0.0$.

2. $f_x + 1 \leq f_s \leq e_s \leq e_x - 1$ and $e = +0.0$.

Lemma Family SETZ-E

Lemmas in Family SETZ-E (for "exact") apply to addends with different trailing exponents whose floating-point sum is exact (i.e., the rounding error is zero).

Lemma SETZ-EN0: If $s_x = s_y$ or $e_x > f_x$, $f_x > e_y$, and $e_x < f_y + p$, then $s_s = s_x$, $e_s = e_x$, $f_s = f_y$, and $e = +0.0$.

Lemma SETZ-EN1: If $s_x \neq s_y$, and one of the following statements holds:

1. $e_x = f_x$, $f_x > e_y + 1$, $e_x < f_y + (p + 1)$

2. $e_x = f_x + 1$, $f_x = e_y$, $e_y > f_y$

then $s_s = s_x$, $e_s = e_x - 1$, $f_s = f_y$, and $e = +0.0$.

Lemma SETZ-ESP0: If $s_x = s_y$, either $(e_x > e_y > f_x > f_y)$ or $(e_x > e_y + 1 > f_x > f_y)$, and $e_x < f_y + (p - 1)$, then $s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, $f_s = f_y$, and $e = +0.0$.

Lemma SETZ-ESP1: If $s_x = s_y$, $e_x = e_y + 1$, $e_y = f_x > f_y$, and $e_x < f_y + (p - 1)$, then $s_s = s_x$, $e_s = e_x + 1$, $f_s = f_y$, and $e = +0.0$.

Lemma SETZ-ESC: If $s_x = s_y$, $e_x > e_y$, $f_x < f_y$, and $e_x < f_x + (p - 1)$, then $s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, $f_s = f_x$, and $e = +0.0$.

Lemma SETZ-ESS: If $s_x = s_y$, $e_x = e_y$, $f_x < f_y$, $e_x < f_x + (p - 1)$, and $e_y < f_y + (p - 1)$, then $s_s = s_x$, $e_s = e_x + 1$, $f_s = f_x$, and $e = +0.0$.

Lemma SETZ-EDP0: If $s_x \neq s_y$, $e_x > e_y + 1 > f_x > f_y$, and $e_x < f_y + p$, then $s_s = s_x$, $e_x - 1 \leq e_s \leq e_x$, $f_s = f_y$, and $e = +0.0$.

Lemma SETZ-EDP1: If $s_x \neq s_y$, $e_x = e_y + 1$, $e_y > f_x > f_y$, and $e_x < f_y + p$, then $s_s = s_x$, $f_x \leq e_s \leq e_x$, $f_s = f_y$, and $e = +0.0$.

Lemma SETZ-EDP2: If $s_x \neq s_y$, $e_x = e_y + 1 = f_x$, and $f_x > f_y + 1$, then $s_s = s_x$, $f_y \leq e_s \leq e_x - 2$, $f_s = f_y$, and $e = +0.0$.

Lemma SETZ-EDP3: If $s_x \neq s_y$, $e_x = e_y + 1 = f_x = f_y + 1$, then $s_s = s_x$, $f_y \leq e_s \leq e_x - 1$, $f_s = f_y$, and $e = +0.0$.

Lemma SETZ-EDC0: If $s_x \neq s_y$, $e_x > e_y + 1$, and $f_x < f_y$, then $s_s = s_x$, $e_x - 1 \leq e_s \leq e_x$, $f_s = f_x$, and $e = +0.0$.

Lemma SETZ-EDC1: If $s_x \neq s_y$, $e_x = e_y + 1$, and $f_x < f_y$, then $s_s = s_x$, $f_y \leq e_s \leq e_x$, $f_s = f_x$, and $e = +0.0$.

Lemma SETZ-EDC2: If $s_x \neq s_y$, $e_x = e_y = f_y$, and $f_x < f_y$, then $s_s = s_x$, $f_x \leq e_s \leq e_x - 1$, $f_s = f_x$, and $e = +0.0$.

Lemma SETZ-EDS0: If $s_x \neq s_y$, $e_x = e_y$, $f_x < f_y$, $e_x > f_x + 1$, and $e_y > f_y + 1$, then $f_x \leq e_s \leq e_x - 1$, $f_s = f_x$, and $e = +0.0$.

Lemma SETZ-EDS1: If $s_x \neq s_y$, $e_x = e_y$, $e_x > f_x + 1$, and $e_y = f_y + 1$, then $f_x \leq e_s \leq e_x - 2$, $f_s = f_x$, and $e = +0.0$.

Lemma Family SETZ-O

Lemmas in Family SETZ-O (for "overlap") apply to addends with completely overlapping mantissas whose floating-point sum has nonzero error.

Lemma SETZ-O0: If $s_x = s_y$, $e_x = f_x + (p-1)$, and $e_x > e_y > f_y > f_x$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_s = f_x$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-3) \leq f_s \leq e_y$, $f_x \leq e_e \leq e_x - (p-1)$, and $f_e = f_x$.
3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_x \leq e_e \leq e_x - (p-1)$, and $f_e = f_x$.

Lemma SETZ-O1: If $s_x = s_y$, $e_x = f_x + (p-1)$, and $e_x > e_y = f_y > f_x + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_s = f_x$, and $e = +0.0$.

2. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-3) \leq f_s \leq e_y - 1$, $f_x \leq e_e \leq e_x - (p-1)$, and $f_e = f_x$.

3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_y$, $s_e \neq s_y$, $f_x \leq e_e \leq e_x - (p-1)$, and $f_e = f_x$.

4. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_x \leq e_e \leq e_x - (p-1)$, and $f_e = f_x$.

Lemma SETZ-O2: If $s_x = s_y$, $e_x = f_x + (p-1)$, and $e_y = f_y = f_x + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_s = f_x$, and $e = +0.0$.

2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_x \leq e_e \leq e_x - (p-1)$, and $f_e = f_x$.

Lemma Family SETZ-1

Lemma SETZ-1: If $e_x < e_y + p$, $e_x > f_y + p$, $f_x > e_y + 1$, and $e_x > f_x$ or $s_x = s_y$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-1) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $f_s = e_y + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-1A: If $e_x = e_y + p$, $e_x > f_y + p$, $f_x > e_y + 1$, and $e_x > f_x$ or $s_x = s_y$, then $s_s = s_x$, $e_s = e_x$, $f_s = e_y + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-1B0: If $e_x < e_y + (p-1)$, $e_x = f_y + p$, $f_x > e_y + 1$, and $e_x > f_x$ or $s_x = s_y$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $f_s = e_y + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-1B1: If $e_x = e_y + (p-1)$, $e_x = f_y + p$, $f_x > e_y + 1$, and $e_x > f_x$ or $s_x = s_y$, then $s_s = s_x$, $e_s = e_x$, $f_s = e_y + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma Family SETZ-2

Lemma SETZ-2: If $s_x = s_y$, $e_x > f_y + p$, and $f_x < e_y$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-1) \leq f_s \leq e_x - 1$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

4. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2A0: If $s_x = s_y$, $e_x = f_y + p$, $f_x < e_y$, and $e_y < f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_x - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2A1: If $s_x = s_y$, $e_x = f_y + p$, $f_x + 1 < e_y$, and $e_y = f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_x - 2$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2A2: If $s_x = s_y$, $e_x = f_y + p$, $f_x + 1 = e_y$, and $e_y = f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 2$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

4. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2B0: If $s_x = s_y$, $e_x > f_y + p$, $f_x = e_y$, and $e_x < f_x + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-1) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
4. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
5. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
6. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2B1: If $s_x = s_y$, $e_x > f_y + p$, $f_x = e_y$, and $e_x = f_x + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2C0: If $s_x = s_y$, $e_x = f_y + (p-1)$, $f_x < e_y$, $e_x < f_x + (p-1)$, and $e_y < f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_s = f_y$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-3) \leq f_s \leq e_y$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.

Lemma SETZ-2C1: If $s_x = s_y$, $e_x = f_y + (p-1)$, $f_x < e_y$, $e_x < f_x + (p-1)$, and $e_y = f_y + (p-1)$, then $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-3) \leq f_s \leq e_y$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.

Lemma SETZ-2D0: If $s_x = s_y$, $e_x > f_y + p$, $f_x = e_y + 1$, and $e_x < f_x + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-1) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x$, $e_y + 2 \leq f_s \leq e_x - 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
4. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-2D1: If $s_x = s_y$, $e_x > f_y + p$, $f_x = e_y + 1$, and $e_x = f_x + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_y + 2 \leq f_s \leq e_x - 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-2AB0: If $s_x = s_y$, $e_x = f_y + p$, $f_x = e_y$, $e_x < f_x + (p-1)$, and $e_y < f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
4. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
5. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
6. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2AB1: If $s_x = s_y$, $e_x = f_y + p$, $f_x = e_y$, and $e_x = f_x + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2AB2: If $s_x = s_y$, $e_x = f_y + p$, $f_x = e_y$, and $e_y = f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2BC0: If $s_x = s_y$, $e_x = f_y + (p-1)$, $f_x = e_y$, $e_y > f_y + 1$, and $e_y < f_y + (p-2)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_s = f_y$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-3) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.
4. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.

Lemma SETZ-2BC1: If $s_x = s_y$, $e_x = f_y + (p-1)$, $f_x = e_y$, and $e_y > f_y + (p-3)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-3) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.

Lemma SETZ-2BC2: If $s_x = s_y$, $e_x = f_y + (p-1)$, $f_x = e_y$, and $e_y = f_y + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $f_s = f_y$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p-1)$, and $f_e = f_y$.

Lemma SETZ-2AD0: If $s_x = s_y$, $e_x = f_y + p$, $f_x = e_y + 1$, and $e_x < f_x + (p-2)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $e_y + 2 \leq f_s \leq e_x - 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

4. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2AD1: If $s_x = s_y$, $e_x = f_y + p$, $f_x = e_y + 1$, and $e_x > f_x + (p - 3)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_y + 2 \leq f_s \leq e_x - 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma Family SETZ-3

Lemma SETZ-3: If $s_x \neq s_y$, $e_x > f_y + (p + 1)$, and $f_x < e_y$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $e_x - p \leq f_s \leq e_y$, $f_y \leq e_e \leq e_x - (p + 2)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x$, $e_x - (p - 1) \leq f_s \leq e_x - 1$, $f_y \leq e_e \leq e_x - (p + 1)$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $f_s = e_x$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p + 2)$, and $f_e = f_y$.

4. $s_s = s_x$, $e_s = e_x$, $f_s = e_x$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p + 1)$, and $f_e = f_y$.

Lemma SETZ-3A: If $s_x \neq s_y$, $e_x = f_y + (p + 1)$, and $f_x < e_y$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $e_x - (p - 1) \leq f_s \leq e_y$, $f_y \leq e_e \leq e_x - (p + 1)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x$, $e_x - (p - 1) \leq f_s \leq e_x$, $f_y \leq e_e \leq e_x - (p + 1)$, and $f_e = f_y$.

Lemma SETZ-3B: If $s_x \neq s_y$, $e_x > f_y + (p + 1)$, and $f_x = e_y$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $e_x - p \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p + 2)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x - 1$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p + 2)$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $e_x - (p - 1) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p + 1)$, and $f_e = f_y$.

4. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

5. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

6. $s_s = s_x$, $e_s = e_x$, $f_s = e_x$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p+2)$, and $f_e = f_y$.

Lemma SETZ-3C0: If $s_x \neq s_y$, $e_x = f_y + p$, $f_x < e_y$, and $e_y < f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $f_s = f_y$, and $e = +\mathsf{0.0}$.

2. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_x - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $f_s = e_x$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-3C1: If $s_x \neq s_y$, $e_x = f_y + p$, $f_x + 1 < e_y$, and $e_y = f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $f_x \leq e_s \leq e_x - 1$, $f_s = f_y$, and $e = +\mathsf{0.0}$.

2. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_x - 2$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $f_s = e_x$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-3C2: If $s_x \neq s_y$, $e_x = f_y + p$, $f_x + 1 = e_y$, and $e_y = f_y + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_x - 2 \leq e_s \leq e_x - 1$, $f_s = f_y$, and $e = +\mathsf{0.0}$.

2. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 2$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $f_s = e_y - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

4. $s_s = s_x$, $e_s = e_x$, $f_s = e_x$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-3D0: If $s_x \neq s_y$, $e_x > f_y + p$, $f_x = e_y + 1$, and $e_x < f_x + (p-1)$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-1) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e = s_y$, $f_y \le e_e \le e_x - (p+1)$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $e_y + 2 \le f_s \le e_x$, $s_e \ne s_y$, $f_y \le e_e \le e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-3D1: If $s_x \ne s_y$, $e_x > f_y + p$, $f_x = e_y + 1$, and $e_x = f_x + (p-1)$, then $s_s = s_x$, $e_s = e_x$, $e_y + 2 \le f_s \le e_x$, $s_e \ne s_y$, $f_y \le e_e \le e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-3AB: If $s_x \ne s_y$, $e_x = f_y + (p+1)$, and $f_x = e_y$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $e_x - (p-1) \le f_s \le e_y - 1$, $f_y \le e_e \le e_x - (p+1)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x - 1$, $f_s = e_y$, $s_e \ne s_y$, $f_y \le e_e \le e_x - (p+1)$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $e_x - (p-1) \le f_s \le e_y - 1$, $f_y \le e_e \le e_x - (p+1)$, and $f_e = f_y$.

4. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e \ne s_y$, $f_y \le e_e \le e_x - (p+1)$, and $f_e = f_y$.

5. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \le f_s \le e_x$, $s_e = s_y$, $f_y \le e_e \le e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-3BC0: If $s_x \ne s_y$, $e_x = f_y + p$, $f_x = e_y$, $e_x > f_x + 1$, and $e_y > f_y + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $f_s = f_y$, and $e = +0.0$.

2. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \le f_s \le e_y - 1$, $f_y \le e_e \le e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e \ne s_y$, $f_y \le e_e \le e_x - p$, and $f_e = f_y$.

4. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \le f_s \le e_x - 1$, $s_e = s_y$, $f_y \le e_e \le e_x - p$, and $f_e = f_y$.

Lemma SETZ-3BC1: If $s_x \ne s_y$, $e_x = f_y + p$, $f_x = e_y$, and $e_y = f_y + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $f_s = f_y$, and $e = +0.0$.

2. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \le f_s \le e_x - 1$, $s_e = s_y$, $f_y \le e_e \le e_x - p$, and $f_e = f_y$.

Lemma SETZ-3CD0: If $s_x \ne s_y$, $e_x = f_y + p$, $f_x = e_y + 1$, $e_x > f_x$, and $e_y > f_y + 1$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x$, $e_y + 2 \leq f_s \leq e_x$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-3CD1: If $s_x \neq s_y$, $e_x = f_y + p$, $f_x = e_y + 1$, and $e_y < f_y + 2$, then $s_s = s_x$, $e_s = e_x$, $e_y + 2 \leq f_s \leq e_x$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma Family SETZ-4

Lemma SETZ-4: If $s_x \neq s_y$, $e_x > f_y + (p+1)$, $f_x < e_y + (p+1)$, and $e_x = f_x$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $e_x - p \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p+2)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x - 1$, $f_s = e_y$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p+2)$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x - 1$, $f_y + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+2)$, and $f_e = f_y$.

Lemma SETZ-4A0: If $s_x \neq s_y$, $e_x = f_y + (p+1)$, $f_x < e_y + p$, and $e_x = f_x$, then exactly one of the following statements is true:

1. $s_s = s_x$, $e_s = e_x - 1$, $e_x - (p-1) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

2. $s_s = s_x$, $e_s = e_x - 1$, $f_s = e_y$, $s_e = s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

3. $s_s = s_x$, $e_s = e_x - 1$, $f_s = e_y + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-4A1: If $s_x \neq s_y$, $e_x = f_y + (p+1)$, $f_x = e_y + p$, and $e_x = f_x$, then $s_s = s_x$, $e_s = e_x - 1$, $e_x - (p-1) \leq f_s \leq e_y + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+1)$, and $f_e = f_y$.

Lemma SETZ-4B: If $s_x \neq s_y$, $e_x > f_y + (p+1)$, $f_x = e_y + (p+1)$, and $e_x = f_x$, then $s_s = s_x$, $e_s = e_x - 1$, $e_x - p \leq f_s \leq e_y + 1$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - (p+2)$, and $f_e = f_y$.

Bibliography

[1] Ryan Abbott, William Detmold, Fernando Romero-López, Zohreh Davoudi, Marc Illa, Assumpta Parreño, Robert J. Perry, Phiala E. Shanahan, and Michael L. Wagman. Lattice quantum chromodynamics at large isospin density. *Phys. Rev. D*, 108:114506, Dec 2023.

[2] Ivo Babuška. Numerical stability in problems of linear algebra. *SIAM Journal on Numerical Analysis*, 9(1):53–77, 1972.

[3] David H. Bailey. High-precision floating-point arithmetic in scientific computation. *Computing in Science & Engineering*, 7(3):54–61, 2005.

[4] David H. Bailey. Reproducibility and variable precision computing. *The International Journal of High Performance Computing Applications*, 34(5):483–490, 2020.

[5] David H. Bailey. High-precision software directory. https://www.davidhbailey.com/dhbsoftware/, 2024.

[6] David H. Bailey and Jonathan M. Borwein. Hand-to-hand combat with thousand-digit integrals. *Journal of Computational Science*, 3(3):77–86, 2012. Scientific Computation Methods and Applications.

[7] David H. Bailey, R. Krasny, and R. Pelz. Multiple precision, multiple processor vortex sheet roll-up computation. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (United States), December 1993.

[8] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard – version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT '10)*, July 2010. Edinburgh, Scotland.

[10] Dimitris Bertsimas and John Tsitsiklis. Simulated annealing. *Statistical science*, 8(1):10–15, 1993.

[11] Sylvie Boldo, Stef Graillat, and Jean-Michel Muller. On the robustness of the 2Sum and Fast2Sum algorithms. *ACM Trans. Math. Softw.*, 44(1), July 2017.

[12] Sylvie Boldo, Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. Formal verification of a floating-point expansion renormalization algorithm. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 98–113, Cham, 2017. Springer International Publishing.

[13] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252, 2011.

[14] Sylvie Boldo and Jean-Michel Muller. Exact and approximated error of the FMA. *IEEE Transactions on Computers*, 60(2):157–164, 2011.

[15] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, page 150–153, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] Mythile C, Jaisiva S, Arunadevi A, and Sudhapriya K. Examining floating point precision in contemporary FPGAs. In *2024 Third International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT)*, pages 1–7, 2024.

[17] Alexandre Chapoutot. Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, pages 184–200, Berlin, Heidelberg, 2010. Springer.

[18] Liqian Chen, Antoine Miné, and Patrick Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In G. Ramalingam, editor, *Programming Languages and Systems*, pages 3–18, Berlin, Heidelberg, 2008. Springer.

[19] Stephen Chrisomalis. *Numerical Notation: A Comparative History.* Cambridge University Press, 2010.

[20] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2013. Springer.

[21] Caroline Collange, Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. Parallel floating-point expansions for extended-precision GPU computations. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 139–146, 2016.

[22] Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter, and Jean-Michel Muller. CR-LIBM: A library of correctly

rounded elementary functions in double-precision. Research report, LIP, December 2006.

[23] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying floating-point implementations using Gappa. working paper or preprint, December 2007.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, June 1971.

[26] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 737–744, Cham, 2014. Springer International Publishing.

[27] Ecma International. *Standard ECMA-262 - ECMAScript Language Specification.* 15 edition, 2024.

[28] Chris Elrod and François Févotte. Accurate and Efficiently Vectorized Sums and Dot Products in Julia. Version submitted to the Correctness2019 workshop, August 2019.

[29] N. M. Evstigneev, O. I. Ryabkov, A. N. Bocharov, V. P. Petrovskiy, and I. O. Teplyakov. Compensated summation and dot product algorithms for floating-point vectors on parallel architectures: Error bounds, implementation and application in the Krylov subspace methods. *Journal of Computational and Applied Mathematics*, 414:114434, 2022.

[30] Nicolas Fabiano, Jean-Michel Muller, and Joris Picot. Algorithms for triple-word arithmetic. *IEEE Transactions on Computers*, 68(11):1573–1583, 2019.

[31] Giovanni Fantuzzi, David Goluskin, and Jean-Bernard Lasserre. Polynomial optimization for nonlinear dynamics: Theory, algorithms and applications. *Oberwolfach Reports*, 21(3):1975–2032, February 2025.

[32] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13–es, June 2007.

[33] Alexei M. Frolov. High-precision, variational, bound-state calculations in Coulomb three-body systems. *Phys. Rev. E*, 62:8740–8745, December 2000.

[34] Alexei M. Frolov and David H. Bailey. Highly accurate evaluation of the few-body auxiliary functions and four-body integrals. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 36(9):1857, April 2003.

[35] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013.

[36] Anthony Garreffa. NVIDIA spent $10 billion on developing its next-generation Blackwell GPU. https://www.tweaktown.com/news/96987/nvidia-spent-10-billion-on-developing-its-next-generation-blackwell-gpu/index.html, March 2024.

[37] GCC Team. *libquadmath: The GCC Quad-Precision Math Library.* GNU Compiler Collection, 2023. Accessed: 2025-04-14.

[38] Torbjrn Granlund and GMP Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library.* Samurai Media Limited, London, GBR, 2015.

[39] Hilbert Hagedoorn. Nvidia Blackwell GPU: R&D costs and pricing $30,000 to $40,000 for just the GPU. https://www.guru3d.com/story/nvidia-blackwell-gpu-costs-and-pricing-to-for-just-the-gpu/, March 2024.

[40] William Hart, Fredrik Johansson, and Sebastian Pancratz. *FLINT: Fast Library for Number Theory.* FLINT Project, 2023. Accessed: 2025-04-14.

[41] Yun He and Chris H. Q. Ding. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. *The Journal of Supercomputing*, 18(3):259–277, Mar 2001.

[42] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, pages 155–162, June 2001. ISSN: 1063-6889.

[43] Nicholas J. Higham and Theo Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 31:347–414, 2022.

[44] Institute of Electrical and Electronics Engineers. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985.

[45] Institute of Electrical and Electronics Engineers. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[46] Institute of Electrical and Electronics Engineers. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[47] Konstantin Isupov, Vladimir Knyazkov, and Alexander Kuvaev. Design and implementation of multiple-precision BLAS level 1 functions for graphics processing units. *Journal of Parallel and Distributed Computing*, 140:25–36, 2020.

[48] Manish Kumar Jaiswal and Ray C.C. Cheung. Area-efficient FPGA implementation of quadruple precision floating point multiplier. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 376–382, 2012.

[49] Manish Kumar Jaiswal and Hayden K.-H So. Architecture for quadruple precision floating point division with multi-precision support. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 239–240, 2016.

[50] Manish Kumar Jaiswal and Hayden K.-H. So. An Unified Architecture for Single, Double, Double-Extended, and Quadruple Precision Division. *Circuits, Systems, and Signal Processing*, 37(1):383–407, January 2018.

[51] Claude-Pierre Jeannerod and Paul Zimmermann. FastTwoSum revisited. In *2025 IEEE 32nd Symposium on Computer Arithmetic (ARITH)*, pages 141–148, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[52] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[53] Fredrik Johansson. Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66:1281–1292, 2017.

[54] Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. Tight and rigorous error bounds for basic building blocks of double-word arithmetic. *ACM Trans. Math. Softw.*, 44(2), October 2017.

[55] Mioara Joldes, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker. CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications. In *5th International Congress on Mathematical Software (ICMS)*, Berlin, Germany, July 2016.

[56] Mioara Joldeş, Olivier Marty, Jean-Michel Muller, and Valentina Popescu. Arithmetic algorithms for extended precision using floating-point expansions. *IEEE Transactions on Computers*, 65(4):1197–1210, 2016.

[57] Christophe Junke and François Bobot. Visualization of execution traces in the Colibri 2 SMT solver. In Martin Bromberger and Antti Hyvärinen, editors, *Proceedings of the 23rd International Workshop on Satisfiability Modulo Theories (SMT 2025)*, volume 4008 of *CEUR Workshop Proceedings*, pages 126–135, August 2025.

[58] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, January 1965.

[59] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Trans. Math. Softw.*, 23(4):561–589, December 1997.

[60] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[61] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[62] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. Cryptopt: Verified compilation with randomized program search for cryptographic primitives. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[63] Christoph Quirin Lauter. Basic building blocks for a triple-double intermediate format. Research Report RR-5702, LIP RR-2005-38, INRIA, LIP, September 2005.

[64] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, June 2002.

[65] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[66] Ding Ma and Michael A. Saunders. *Solving Multiscale Linear Programs Using the Simplex Method in Quadruple Precision*, page 223–235. Springer International Publishing, 2015.

[67] Ding Ma, Laurence Yang, Ronan M. T. Fleming, Ines Thiele, Bernhard O. Palsson, and Michael A. Saunders. Reliable and efficient solution of genome-scale models of metabolism and macromolecular expression. *Scientific Reports*, 7(1), January 2017.

[68] John Maddock and Christopher Kormanyos. *Boost.Multiprecision: A Multiprecision Arithmetic Library for C++*. Boost C++ Libraries, 2023. Accessed: 2025-04-14.

[69] Henry Massalin. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126, 1987.

[70] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix. *SIAM Review*, 20(4):801–836, 1978.

[71] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), May 2008.

[72] Jean-Michel Muller, Nicolas Brunie, Florent De Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Springer International Publishing, Cham, 2018.

[73] Jean-Michel Muller and Laurence Rideau. Formalization of double-word arithmetic, and comments on "Tight and rigorous error bounds for basic building blocks of double-word arithmetic". *ACM Trans. Math. Softw.*, 48(1), February 2022.

[74] Ole Møller. Quasi double-precision in floating point addition. *BIT Numerical Mathematics*, 5(1):37–50, March 1965.

[75] Alessio Netti, Yang Peng, Patrik Omland, Michael Paulitsch, Jorge Parra, Gustavo Espinosa, Udit Agarwal, Abraham Chan, and Karthik Pattabiraman. Mixed precision support in HPC applications: What about reliability? *Journal of Parallel and Distributed Computing*, 181:104746, 2023.

[76] A. Neumaier. Rundungsfehleranalyse einiger verfahren zur summation endlicher summen. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 54(1):39–51, 1974.

[77] Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023.

[78] Aina Niemetz, Mathias Preiner, and Yoni Zohar. Scalable bit-blasting with abstractions. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification*, pages 178–200, Cham, 2024. Springer Nature Switzerland.

[79] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[80] Pavel Panchekha. FastTwoSum is Faster Than TwoSum. Blog post, May 2025.

[81] Jessica Pointing, Oded Padon, Zhihao Jia, Henry Ma, Auguste Hirth, Jens Palsberg, and Alex Aiken. Quanto: Optimizing quantum circuits with automatic generation of circuit identities. *Quantum Science and Technology*, 9(4):045009, 2024.

[82] D.M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*, pages 132–143, 1991.

[83] Python Software Foundation. The Python standard library: Built-in functions. https://docs.python.org/3/library/functions.html, 2001–2025.

[84] Joao Rivera, Franz Franchetti, and Markus Püschel. Floating-point TVPI abstract domain. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[85] Andreas Rossberg. WebAssembly Core Specification.

[86] Siegfried M. Rump and Marko Lange. On the definition of unit roundoff. *BIT Numerical Mathematics*, 56(1):309–317, March 2016.

[87] Jeffrey Sarnoff. DoubleFloats.jl. https://github.com/JuliaMath/DoubleFloats.jl, 2024. Version 1.4.2.

[88] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[89] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[90] Anton Shilov. Apple spent $1 billion to tape out new M3 processors: Analyst. https://www.tomshardware.com/software/macos/apple-spent-dollar1-billion-to-tape-out-new-m3-processors-analyst, November 2023.

[91] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. The CORE-MATH Project. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 26–34, virtual, France, September 2022. IEEE.

[92] Erich Strohmaier, Jack Dongarra, Horst D. Simon, and Martin Meuer. TOP500 List - June 2025. https://top500.org/lists/top500/2025/06/, June 2025.

[93] The Julia Project. Julia standard library: Arrays. https://docs.julialang.org/en/v0.6/stdlib/arrays, 2017.

[94] Lloyd N. Trefethen. The definition of numerical analysis. Technical report, USA, 1992.

[95] Gerhard W. Veltkamp. ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie. Technical Report 22, Technische Hogeschool Eindhoven, 1968.

[96] Gerhard W. Veltkamp. ALGOL procedures voor het rekenen in dubbele lengte. Technical Report 21, Technische Hogeschool Eindhoven, 1969.

[97] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice-Hall, Englewood Cliffs, N.J., 1963.

[98] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. Quartz: superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 625–640, 2022.

[99] David K. Zhang. MultiFloats.jl. https://github.com/dzhang314/MultiFloats.jl, 2024. Version 2.3.0.

[100] David K. Zhang and Alex Aiken. Automatic verification of floating-point accumulation networks. In Ruzica Piskac and Zvonimir Rakamarić, editors, *Computer Aided Verification*, pages 215–237, Cham, 2025. Springer Nature Switzerland.

[101] David K. Zhang and Alex Aiken. High-performance branch-free algorithms for extended-precision floating-point arithmetic. In *SC25: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2025.

[102] David Kai Zhang. An explicit 16-stage Runge–Kutta method of order 10 discovered by numerical search. *Numerical Algorithms*, 96(3):1243–1267, Jul 2024.

[103] Kasia Świrydowicz, Eric Darve, Wesley Jones, Jonathan Maack, Shaked Regev, Michael A. Saunders, Stephen J. Thomas, and Slaven Peleš. Linear solvers for power grid optimization problems: A review of gpu-accelerated linear solvers. *Parallel Computing*, 111:102870, 2022.