Automatic Verification of Floating-Point Accumulation Networks

David K. Zhang[0000-0003-3379-4890] and Alex Aiken[0000-0002-3723-9555]

Stanford University

Abstract. Floating-point accumulation networks (FPANs) are key building blocks used in many floating-point algorithms, including compensated summation and double-double arithmetic. FPANs are notoriously difficult to analyze, and algorithms using FPANs are often published without rigorous correctness proofs. In fact, on at least one occasion, a published error bound for a widely used FPAN was later found to be incorrect. In this paper, we present an automatic procedure that produces computer-verified proofs of several FPAN correctness properties, including error bounds that are tight to the nearest bit. Our approach is underpinned by a novel floating-point abstraction that models the sign, exponent, and number of leading and trailing zeros and ones in the mantissa of each number flowing through an FPAN. We also present a new FPAN for double-double addition that is faster and more accurate than the previous best known algorithm.

Keywords: Automatic theorem proving · Floating-point arithmetic · Error-free transformations · TwoSum algorithm · Double-double arithmetic · Quad-double arithmetic · Rounding error

1 Introduction

Many scientific and mathematical problems demand calculations that exceed the native precision limits of floating-point hardware (typically IEEE 64-bit or Intel x87 80-bit). To address this need, numerical programmers turn to *compensated algorithms*, such as Kahan–Babuška–Neumaier summation [37,5,48], and *floating-point expansions* [52], such as double-double and quad-double arithmetic [21,33]. These techniques enable a processor to execute floating-point operations with double, quadruple, or even higher multiples of its native precision. They are widely adopted in dense [42] and sparse [26] numerical linear algebra, high-precision quadrature [8], robust computational geometry [57], fluid dynamics [6,32], quantum chemistry [29,30], correctly rounded transcendental functions [19,58], and the discovery of new mathematical identities [9].

All of these techniques are based on common building blocks known as *error-free transformations* [51], which are floating-point algorithms that exactly compute their own rounding errors. For example, the Møller–Knuth TwoSum algorithm [47,39] takes a pair of floating-point numbers (x, y) and computes both

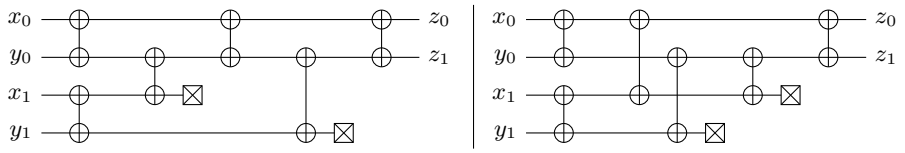

Fig. 1. Network diagram of the ddadd algorithm due to Li et al. (left) and madd, our new double-double addition algorithm (right). This graphical FPAN notation will be explained in detail in Section 3.

their rounded 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 true mathematical addition.

Although the TwoSum algorithm has been extensively studied, proven correct by both pen-and-paper [39,11] and formal methods [12,46], issues arise when multiple TwoSum operations are combined to accumulate three or more floating-point values. Multiple rounding error terms can interact in subtle and unexpected ways, significantly affecting the precision of the overall result when terms that are incorrectly assumed to be negligible are discarded.

In 2017, Joldes, Muller, and Popescu [35] discovered an error of this type in the ddadd algorithm designed by Li et al. for the XBLAS extended-precision linear algebra library [42]. Li et al. originally claimed that ddadd computes sums with relative error bounded by $2 \cdot 2^{-106} = 2\mathbf{u}^2$ when executed in IEEE binary64 (double precision) arithmetic, where $\mathbf{u} = 2^{-53}$ denotes the unit roundoff. Joldes, Muller, and Popescu refuted this claim by identifying a class of inputs in which two discarded error terms interfere constructively, producing a relative error of $2.25 \cdot 2^{-106} = 2.25\mathbf{u}^2$ in the final sum. While this weakened bound does not invalidate the usefulness of XBLAS, it demonstrates that even expert numerical analysts can make subtle mistakes in the analysis of rounding errors.

The ddadd algorithm is a prototypical example of a *floating-point accumulation network (FPAN)* — a term we introduce to describe a branch-free linear sequence of floating-point sum and TwoSum operations. Although this class of algorithms, to our knowledge, has never been explicitly named in the floating-point literature, FPANs pervade floating-point algorithms and occur in every single paper referenced in this introduction, along with dozens of software packages [27,53,19,58,33,43,25,36,56,61,7]. In addition to their obvious uses for high-precision addition and subtraction, FPANs also occur as subroutines in multiplication, divison, and square root algorithms [38,33], which in turn are used to implement transcendental functions, including the exponential, logarithm, and trigonometric functions, in many standard libraries [19,58,53].

In this paper, we present an automatic procedure for proving several FPAN correctness properties, including error bounds and nonoverlapping invariants. Our key technical insight is a one-sided reduction from a correctness property P of an FPAN F to a satisfiability problem $S_{P,F}$ in quantifier-free Presburger arithmetic (`QF_LIA`). Here, one-sidedness means that if $S_{P,F}$ is unsatisfiable, then $P(F)$ provably holds, but if $S_{P,F}$ is satisfiable, then $P(F)$ may or may not hold. In all cases we have tested, we find that the satisfiability of $S_{P,F}$ is far easier

for SMT solvers to determine than direct verification of $P(F)$ in a floating-point theory (`QF_FP`). Our reduced problems $S_{P,F}$ can be solved in less than one second, while direct $P(F)$ verifiers fail to terminate after multiple days of runtime.

Using our procedure, we discover and prove the correctness of a new algorithm for double-double addition — the same problem that ddadd solves — with strictly smaller relative error and lower circuit depth. Thus, our algorithm, named madd (for "More Accurate Double-Double addition," shown in Figure 1) is both faster and more accurate than the previous best known algorithm for this task. We expect our automatic decision procedure to enable additional novel algorithms to be found by computer search in future work.

Our reduction strategy uses a novel abstraction of floating-point arithmetic that does not consider the full bitwise representation of a floating-point number x, but tracks only its sign s_x, exponent e_x, and the number of leading and trailing zeros and ones in its mantissa ($\mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x$). Using this reduced representation, we construct abstract overapproximations of the floating-point sum and TwoSum operations, expressed as linear inequalities in the variables ($s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x$). Although we lose completeness by passing to an overapproximation, we show that our abstraction is nonetheless precise enough to prove best-possible error bounds, tight to the nearest bit, for several FPANs of practical interest, including ddadd and madd. Moreover, we use floating-point SMT solvers to directly verify the soundness of our abstraction.

In summary, this paper makes the following contributions:

1. We define *floating-point accumulation networks* (*FPANs*) and show that FPANs occur in many widely used floating-point algorithms.
2. We give a procedure for reducing correctness properties P of an FPAN F to satisfiability problems $S_{P,F}$ in quantifier-free Presburger arithmetic. Our strategy uses a novel formally verified abstraction of floating-point arithmetic that may be of independent interest.
3. We empirically demonstrate that $S_{P,F}$ is far easier for modern SMT solvers to reason about than $P(F)$, solving otherwise intractable verification problems in under one second.
4. We state a new double-double addition algorithm, madd, with lower circuit depth and strictly smaller relative error than the previous best known algorithm, as rigorously proven by our reduction procedure.

2 Background

2.1 Floating-Point Numbers

A *floating-point number* 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* $m = (m_0, m_1, \ldots, m_{p-1})$, which is an sequence of p *digits* $m_k \in \{0, 1, \ldots, b-1\}$. The value represented by (s, e, m) is defined as the following real number:

$$(-1)^s \times (m_0.m_1 m_2 \ldots m_{p-1}) \times b^e = (-1)^s \sum_{k=0}^{p-1} m_k b^{e-k} \tag{1}$$

Format	Base b	Precision p	Exponent range $\{e_{\min}, \ldots, e_{\max}\}$
binary16	$b = 2$	$p = 11$	$e \in \{-14, \ldots, +15\}$
bfloat16	$b = 2$	$p = 8$	$e \in \{-126, \ldots, +127\}$
binary32	$b = 2$	$p = 24$	$e \in \{-126, \ldots, +127\}$
binary64	$b = 2$	$p = 53$	$e \in \{-1022, \ldots, +1023\}$
binary128	$b = 2$	$p = 113$	$e \in \{-16382, \ldots, +16383\}$

Table 1. Parameters of the floating-point formats defined by the IEEE 754 standard and the nonstandard bfloat16 format commonly used in deep learning accelerators.

Floating-point hardware is almost always designed for base $b = 2$, and we assume $b = 2$ throughout this paper.

The IEEE 754 standard [2,3,4] defines a family of encodings of floating-point numbers (s, e, m) into fixed-width bit-vectors with a specified base b, precision p, and exponent range $e_{\min} \leq e \leq e_{\max}$. These standard encodings, listed in Table 1, have several additional features that are relevant to our analysis.

- A floating-point number with $m_0 = 1$ is said to be *normalized*. A nonzero floating-point number can be normalized without changing its represented real value by shifting its first nonzero bit m_k to position m_0 and adjusting its exponent e to $e - k$. IEEE 754 requires floating-point numbers to be normalized whenever possible so that the *implicit leading bit* $m_0 = 1$ does not need to be explicitly stored.
- The only floating-point numbers that cannot be normalized are zero and very small numbers whose adjusted exponent would fall below the minimum threshold $e_{\min}$. These small nonzero numbers are called *subnormal* and use a special alternative representation with no implicit leading bit.
- Zero has two distinct IEEE 754 encodings with opposite sign bits, denoted by `+0.0` (*positive zero*) and `-0.0` (*negative zero*). These two values are considered to be equal, i.e., `+0.0 == -0.0` evaluates to `true` in any IEEE 754-compliant programming environment.
- IEEE 754 defines three non-numeric floating-point values, called *positive infinity* ($+\infty$), *negative infinity* ($-\infty$), and *not-a-number* (`NaN`). These special values are returned from operations that would otherwise yield an unrepresentably large or indeterminate result. All floating-point values other than $\pm\infty$ and `NaN` are called *finite*.

We assume throughout this paper that all floating-point numbers are normalized or zero, ignoring subnormal numbers, $\pm\infty$, and `NaN`. We also identify `+0.0` with `-0.0`, writing $\pm$`0.0` to denote zero with either sign. These assumptions serve only to simplify exposition and do not affect the generality of our results. Indeed, we prove in Section 5 that our automated proof technique handles all finite floating-point values, including subnormal numbers. Non-finite inputs require no consideration because the algorithms discussed in this paper simply return `NaN` when given $\pm\infty$ or `NaN` inputs.

2.2 Rounding and Error-Free Transformations

In general, the sum or difference of two precision-p floating-point numbers may not be exactly representable as another precision-p floating-point number. To perform calculations at a fixed precision p, the result of every operation must be *rounded* by the following procedure:

1. Compute the exact sum or difference of the real values represented by the floating-point inputs, as if to infinite precision.
2. Find and return the closest precision-p floating-point number to the exact result. A *tie-breaking* rule must be used whenever the exact result is equidistant to two neighboring floating-point values.

As is standard in studies of extended-precision floating-point arithmetic [11,12], we assume throughout this paper that all floating-point operations are rounded to nearest with ties broken to even, denoted by RNE. This is the default rounding mode defined by IEEE 754 and is supported on virtually all general-purpose computing hardware. In fact, RNE is the only rounding mode available in many programming environments, including Python, JavaScript, and WebAssembly [24,1]. Following Knuth's convention [39], we distinguish between rounded and exact arithmetic operations using circled operators.

$$x \oplus y \coloneqq \mathsf{RNE}(x + y) \tag{2}$$

$$x \ominus y \coloneqq \mathsf{RNE}(x - y) \tag{3}$$

Rounding errors in precision-p floating-point arithmetic are characterized by the *unit roundoff*[1] constant $\mathbf{u} \coloneqq 2^{-p}$, which bounds the relative error of any individual rounded operation [55]. For example, for all floating-point numbers a and b such that $|a + b| < (2 - \mathbf{u})2^{e_{\max}}$, there exists $\delta \in \mathbb{R}$ satisfying

$$a \oplus b = (a + b)(1 + \delta) \quad \text{where} \quad |\delta| \leq \mathbf{u}. \tag{4}$$

Analogous results also hold for subtraction and other floating-point operations.

An *error-free transformation* is a floating-point algorithm that computes a rounded arithmetic operation together with the exact rounding error incurred by that operation. The oldest and most important example of an error-free transformation is the Møller–Knuth TwoSum algorithm, first discovered by Møller [47] and proven correct by Knuth [39], which computes the rounding error of a floating-point sum $s \coloneqq x \oplus y$ or difference $d \coloneqq x \ominus y$.

Although the algorithm itself is straightforward, the existence of TwoSum is a highly nontrivial result. It is not obvious *a priori* that the rounding error $e \coloneqq (x+y)-(x\oplus y)$ is always exactly representable as a floating-point number, let alone that it can be recovered using rounded floating-point operations. Despite the apparent simplicity of the pseudocode in Figure 2, proving the correctness of TwoSum requires lengthy case analysis [39, Sec. 4.2.2]. Analogous error-free transformations also exist for floating-point multiplication [59,60,21] and the fused multiply-add operation [14].

[1] The definition $\mathbf{u} \coloneqq 2^{-p}$ is appropriate when floating-point operations are rounded to nearest. For other rounding modes, the larger value $\mathbf{u} \coloneqq 2^{-(p-1)}$ is used instead.

Algorithm: $\mathsf{TwoSum}(x, y)$

Input: Two floating-point numbers (x, y).
Output: Two floating-point numbers (s, e) such that $s = x \oplus y$ and $e = (x + y) - (x \oplus y)$ 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)

Fig. 2. Pseudocode for the Møller–Knuth TwoSum algorithm. This algorithm can also be applied to subtraction by flipping the sign of y throughout.

2.3 Floating-Point Expansions

Floating-point expansion is a technique for representing high-precision numbers as sequences of machine-precision numbers. The basic idea is to represent a high-precision constant $C \in \mathbb{R}$ as a sequence of successive approximations:

$$\begin{aligned} x_0 &\coloneqq \mathsf{RNE}(C) \\ x_1 &\coloneqq \mathsf{RNE}(C - x_0) \\ x_2 &\coloneqq \mathsf{RNE}(C - x_0 - x_1) \\ &\vdots \\ x_{n-1} &\coloneqq \mathsf{RNE}(C - x_0 - x_1 - \cdots - x_{n-2}) \end{aligned} \tag{5}$$

Provided that no overflow or underflow occurs in this process, the final n-term expansion $(x_0, x_1, \ldots, x_{n-1})$ approximates C with precision $np + n - 1$, i.e.,

$$|C - (x_0 + x_1 + \cdots + x_{n-1})| \leq 2^{-(np+n-1)}|C| \tag{6}$$

where p denotes the underlying machine precision. To achieve the full precision of $np + n - 1$ bits, a floating-point expansion must be *nonoverlapping*, i.e.,

$$x_i = \mathsf{RNE}(x_i + x_{i+1}) \tag{7}$$

for each $i = 0, \ldots, n - 2$. This property, illustrated in Figure 3, ensures that no bit of C is redundantly covered by more than one component of the expansion. Intuitively, if there were such a redundant bit, then at least one bit would flip when adding x_i to x_{i+1}, causing $x_i \oplus x_{i+1}$ to differ from x_i.

Arithmetic with floating-point expansions is a delicate procedure that requires skillful use of error-free transformations to propagate rounding errors between terms. Moreover, the nonoverlapping property is easily broken and must be restored after every few operations to avoid loss of precision. Designing sequences of error-free transformations with correct error propagation and nonoverlapping

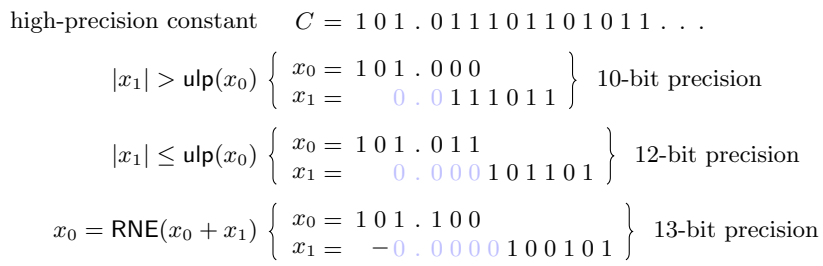

Fig. 3. Decomposition of a high-precision constant C into overlapping and nonoverlapping floating-point expansions with $p = 6$ mantissa bits per term. Light blue digits represent a shift stored in the exponent and are not part of the mantissa. Note that the final expansion rounds x_0 up instead of down. In this case, x_1 is negative, and the mantissa of x_1 contains the one's complement of the corresponding digits in C.

semantics is a remarkably difficult problem; the literature on this subject is punctuated by refutations, corrections, and corrections to those corrections [35,46]. Some general constructions are known, but these algorithms are far from optimal, particularly when the number of inputs is small [18,45].

In principle, a nonoverlapping floating-point expansion can contain up to $\lceil (e_{\max} - e_{\min} + p)/(p+1) \rceil$ terms before underflow occurs, causing all subsequent terms to round down to zero. However, in practice, it is more common to use fixed-length floating-point expansions consisting of two, three, or four terms [42,41,19,58,33]. These fixed-length expansions are called *double-word*, *triple-word*, *quadruple-word* or *double-double*, *triple-double*, *quad-double* representations. The latter names are used when the underlying machine format is IEEE binary64 (double precision), but most algorithms for double-double, triple-double, and quad-double arithmetic also work for other underlying formats.

3 Floating-Point Accumulation Networks

In this section, we formally define floating-point accumulation networks as a class of branch-free floating-point algorithms using a graphical notation inspired by sorting networks [40].

Definition 1. *A **floating-point accumulation network** (**FPAN**) is a diagram consisting of horizontal **wires** and vertical* TwoSum *__gates__ that connect exactly two wires. Each wire may optionally be terminated by the symbol* ⊠*, indicating that it is __discarded__.*

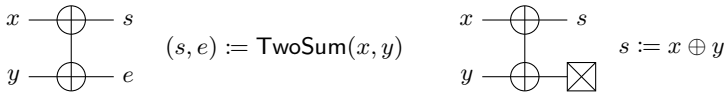

Fig. 4. Gate representations of the floating-point sum and TwoSum operations. In our notation, $x \oplus y$ is treated as a special case of $\mathsf{TwoSum}(x, y)$ with the error term discarded. Note that the inputs are unordered (i.e., $\mathsf{TwoSum}(x, y) = \mathsf{TwoSum}(y, x)$) but the order of the outputs is significant, with larger-magnitude outputs on top.

Algorithm: add2$((x_0, x_1), (y_0, y_1))$

Input: nonoverlapping expansions (x_0, x_1) and (y_0, y_1).
Output: nonoverlapping 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 $(z_0, z_1) \coloneqq \mathsf{TwoSum}(s_0, u)$
5 **return** (z_0, z_1)

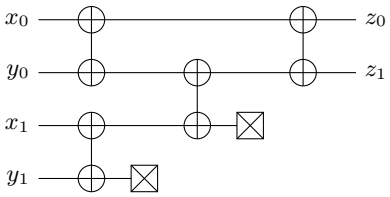

Fig. 5. Pseudocode and network diagram representations of Dekker's add2 algorithm. Note that the intermediate variables s_0, s_1, t, u are unnamed in the network diagram representation, implicitly represented by the wire segments in between TwoSum gates.

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 $(x_1, x_2, \ldots, x_n)$ 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 TwoSum gate, they are updated by $(x_i, x_j) \leftarrow \mathsf{TwoSum}(x_i, x_j)$ as specified in Figure 4. After all TwoSum gates have been executed, all values on wires that are not discarded are returned in top-to-bottom order. To illustrate this definition, Figure 5 presents equivalent pseudocode and network diagram representations of Dekker's add2 algorithm, the first algorithm ever proposed for double-double addition [21].

The purpose of an FPAN is to compute a nonoverlapping floating-point expansion of the exact value of the sum of its inputs. By the defining property of the TwoSum algorithm, this value is invariant to the application of a TwoSum gate to any two wires; it is only ever changed by discarding a wire. Therefore, to prove the correctness of an FPAN, it suffices to show the following:

1. The non-discarded outputs, denoted by $(a_0, a_1, \ldots, a_{n-k-1})$, must satisfy the *nonoverlapping invariant* $a_i = \mathsf{RNE}(a_i + a_{i+1})$ for all $i = 0, \ldots, n - k - 2$.
2. The discarded outputs, denoted by $(b_0, b_1, \ldots, b_{k-1})$, must satisfy an *error bound* of the form $|b_i| \leq C_i \mathbf{u}^{n-k} |a_0|$ for some small constant C_i for each $i = 0, \ldots, k - 1$.

Dekker's add2 algorithm is notable for violating these correctness properties. Assuming nonoverlapping inputs (x_0, x_1) and (y_0, y_1), Dekker proved [21] 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:

$$\frac{|(z_0 + z_1) - (x_0 + x_1 + y_0 + y_1)|}{|x_0 + x_1 + y_0 + y_1|} \leq 4\mathbf{u}^2 \frac{|x_0 + x_1| + |y_0 + y_1|}{|x_0 + x_1 + y_0 + y_1|} \tag{8}$$

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, in which case $|x_0 + x_1| + |y_0 + y_1|$ can be orders of magnitude

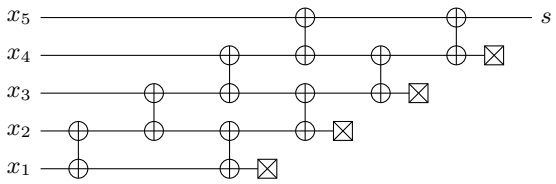

Fig. 6. Network diagram for Kahan–Babuška–Neumaier summation applied to five inputs. This double staircase accumulation pattern generalizes to any number of inputs.

larger than $|x_0+x_1+y_0+y_1|$. Joldes, Muller, and Popescu [35] 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 5 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 key differences that make floating-point accumulation harder than sorting. First, the outputs of an FPAN not only need to be sorted by magnitude; they also require a degree of mutual separation to satisfy nonoverlapping invariants and error bounds. 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.

Example: KBN Summation. Kahan–Babuška–Neumaier (KBN) summation is an algorithm that uses TwoSum to compute floating-point sums with a running compensation term to improve the accuracy of the final result [37,5,48]. 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 [27].

In our graphical notation, the KBN algorithm is written as an FPAN with a double staircase structure illustrated in Figure 6. 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.

4 Abstraction

Existing methods for floating-point verification are ill-suited for FPANs. On one hand, techniques based on interval analysis or projection from real arithmetic are too imprecise to reason about error-free transformations. On the other hand, bit-blasting produces enormous satisfiability problems that are only tractable for tiny mantissa widths. Verifying FPANs requires a technique that can precisely specify the magnitude and shape of a floating-point sum without reasoning through the value of every bit. To this end, we introduce a new abstract domain for floating-point reasoning, which we call the SELTZO abstraction.

Definition 2. *Let x be a 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 and exponent e_x of x;*
2. *the counts $(\mathsf{nlz}_x, \mathsf{nlo}_x)$ of leading zeros and ones, respectively, in the mantissa of x, ignoring the implicit leading bit; and*
3. *the counts $(\mathsf{ntz}_x, \mathsf{nto}_x)$ of trailing zeros and ones, respectively, in the mantissa of x, ignoring the implicit leading bit.*

For example, the SELTZO abstraction of $-1.0010011111_2 \times 2^7$ is $(1, 7, 2, 0, 0, 5)$, and the SELTZO abstraction of $+1.1111111111_2 \times 2^{-2}$ is $(0, -2, 0, 10, 0, 10)$. (Recall that the implicit leading bit is ignored when computing nlz_x and nlo_x.) When $x = \pm\mathsf{0.0}$, we set $e_x := e_{\min} - 1$ for consistency with IEEE 754 representation. Because we assume all floating-point numbers to be normalized or zero, $x = \pm\mathsf{0.0}$ is the only value we consider to have the property $e_x < e_{\min}$.

Our definition of the SELTZO abstraction is motivated by several design considerations that we will explore in the remainder of this section.

- The SELTZO abstraction precisely captures all of the FPAN correctness properties defined in Section 3 using linear inequalities in the precision p and the variables $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$.
- The variables of the SELTZO abstraction are particularly well-behaved under the floating-point sum and TwoSum operations, which interact in a highly predictable fashion with long stretches of leading or trailing zeros and ones.
- As shown by the worst-case inputs found by Joldes, Muller, and Popescu [35], FPANs tend to exhibit pathological behavior near powers of two, which mark the boundaries between different exponent values.

Note that distinct SELTZO abstract values correspond to disjoint sets of concrete floating-point values. 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 concrete values with *exactly* three trailing zeros, no more. As these bit counts increase, the number of concrete values that correspond to a particular abstract value drops exponentially, making the SELTZO abstraction more precise for floating-point numbers with many leading mantissa zeros or ones. These are precisely the floating-point numbers that lie near the boundaries between different exponent values.

4.1 SELTZO Correctness Properties

To verify FPANs, the SELTZO abstraction must be able to express necessary and sufficient conditions for the FPAN correctness properties defined in Section 3, i.e., nonoverlapping invariants of the form $x = \mathsf{RNE}(x + y)$ and relative error bounds of the form $|y| \leq C\mathbf{u}^k|x|$. The following propositions express these properties as linear inequalities in the SELTZO variables $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$ and the precision p of the underlying floating-point format.

Proposition 1. *Let x and y be finite precision-p floating-point numbers in the* binary16, bfloat16, binary32, binary64, *or* binary128 *formats (defined in Table 1). Using the notation of Definition 2, x and y satisfy the nonoverlapping property $x = \mathsf{RNE}(x + y)$ if and only if at least one of the following conditions holds:*

1. $y = \pm 0.0$.
2. $e_x - e_y > p + 1$.
3. $e_x - e_y = p + 1$ *and one or more of the following sub-conditions holds:*
   (a) $s_x = s_y$.
   (b) $\mathsf{ntz}_x < p - 1$ *(i.e., x is not a power of 2).*
   (c) $\mathsf{ntz}_y = p - 1$ *(i.e., y is a power of 2).*
4. $e_x - e_y = p$, $\mathsf{ntz}_y = p - 1$, $\mathsf{ntz}_x \geq 1$, *and one or more of the following sub-conditions holds:*
   (a) $s_x = s_y$.
   (b) $\mathsf{ntz}_x < p - 1$.

We prove Proposition 1 by direct verification with a floating-point SMT solver for each of the formats listed in Table 1. All such solvers we are aware of, namely, Z3 [20], CVC5 [10], MathSAT [17], and Bitwuzla [49], report that the existence of a counterexample to Proposition 1 is unsatisfiable in all listed formats.

Because the SELTZO abstraction works directly with the bitwise representation of a floating-point number, it most naturally expresses relative error bounds $|y| \leq C\mathbf{u}^k|x|$ where $C = 2^j$ is a power of two. We will restrict attention to error bounds in this form in the remainder of this paper.

Proposition 2. *Let x and y be finite precision-p floating-point numbers in any format. If $e_x - e_y > kp - j$ or $e_y = e_{\min} - 1$, then $|y| \leq 2^j\mathbf{u}^k|x|$.*

Proof. If $e_y = e_{\min} - 1$, then $y = \pm 0.0$ and the desired conclusion holds trivially. Otherwise, we have $|x| \in [2^{e_x}, (2 - 2\mathbf{u})2^{e_x}]$ and $|y| \in [2^{e_y}, (2 - 2\mathbf{u})2^{e_y}]$. From the lower bound on x and the upper bound on $|y|$, it follows that:

$$\frac{|y|}{|x|} \leq \frac{(2 - 2\mathbf{u})2^{e_y}}{2^{e_x}} = (1 - \mathbf{u})2^{-(e_x - e_y - 1)} \leq (1 - \mathbf{u})2^{j-kp} < 2^j\mathbf{u}^k \tag{9}$$

This proves $|y| < 2^j\mathbf{u}^k|x|$, as required. □

4.2 SELTZO Abstraction of TwoSum

In addition to expressing correctness properties, the SELTZO abstraction must also precisely capture the behavior of the TwoSum operation. In particular, it must be able to state relations between abstract inputs and outputs that rule out impossible input-output pairs. Relations of this type allow us to verify FPANs by ruling out the possibility of an output that violates a correctness property. The following proposition shows that the set of possible outputs of the TwoSum operation is highly constrained.

Proposition 3. *Let x and y be finite precision-p floating-point numbers in the* binary16, bfloat16, binary32, binary64, *or* binary128 *formats (defined in Table 1). The pair (x, y) is a possible output of* TwoSum *if and only if x and y satisfy at least one of the conditions listed in Proposition 1.*

Proof. First, observe that TwoSum is an idempotent operation, i.e., every output of TwoSum is a fixed point. To see this, let (x, y) be an arbitrary pair of finite precision-p floating-point numbers and set $(s_1, e_1) \coloneqq \mathsf{TwoSum}(x, y)$ and $(s_2, e_2) \coloneqq \mathsf{TwoSum}(s_1, e_1)$. By the defining property of TwoSum, we have $x+y = s_1 + e_1 = s_2 + e_2$, from which we deduce $s_2 = \mathsf{RNE}(s_1 + e_1) = \mathsf{RNE}(x + y) = s_1$ and $e_2 = x + y - s_2 = x + y - s_1 = e_1$.

Next, observe that $x = \mathsf{RNE}(x, y)$ if and only if $(x, y) = \mathsf{TwoSum}(x, y)$. Indeed, if $x = x \oplus y$, then $\mathsf{TwoSum}(x, y) = (x \oplus y,\ (x + y) - (x \oplus y)) = (x, y)$. Combining these two observations, we see that (x, y) is a possible output of TwoSum if and only if $x = \mathsf{RNE}(x + y)$, which is the desired result. □

Propositions 1 and 3 give a sharp characterization of the set of all possible outputs of TwoSum ranging over all possible inputs. However the inputs to a TwoSum gate inside an FPAN are typically constrained, either by assumed hypotheses (e.g., inputs y_0 and y_1 are assumed to be nonoverlapping) or as a consequence of previous TwoSum gates producing its inputs. Thus, the analysis of FPANs requires characterizing the set of possible TwoSum outputs not only over the full domain of all possible inputs, but also on restricted subdomains created by the interactions of these input constraints.

In the Appendix of this paper, we state a collection of several dozen lemmas that precisely characterize possible TwoSum outputs on various input subdomains that collectively cover the space of all possible inputs. We have formally verified the correctness of all of these lemmas for the binary16, bfloat16, binary32, binary64, and binary128 formats using a portfolio of floating-point SMT solvers, including Z3 [20], CVC5 [10], MathSAT [17], and Bitwuzla [49]. In many cases, we have also proven that these lemmas are the strongest possible in the sense that every abstract output not explicitly ruled out by the lemma is witnessed by some concrete input. We provide a full listing of these lemmas using the Z3 Python API at https://github.com/dzhang314/FPANVerifier.

5 Verification

With the SELTZO abstraction defined, we can now state our automatic FPAN verification procedure. Suppose we are given an FPAN F and a property P that is expressible as a logical combination of linear inequalities in the SELTZO varibles $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$. We construct a statement $S_{P,F}$ in quantifier-free Presburger arithmetic that expresses the existence of an abstract counterexample to $P(F)$. If this statement is unsatisfiable, then $P(F)$ has no abstract counterexamples. Hence, no concrete counterexample exists, and $P(F)$ is proven.

The first step of this procedure is to assign a unique label v_i to every wire segment in F. Every TwoSum gate delineates a new segment of the wires it

connects. 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 $6n + 12g$ variables.

We construct the statement $S_{P,F}$ as a logical conjunction of three types of conditions: (1) *consistency conditions* that enforce the validity of the abstract values $(s_v, e_v, \mathsf{nlz}_v, \mathsf{nlo}_v, \mathsf{ntz}_v, \mathsf{nto}_v)$; (2) *execution conditions* that constrain the possible outputs of each TwoSum gate; and (3) *counterexample conditions* that encode the negation of the property $P(F)$ that we wish to prove.

We first state the consistency conditions in Equations (10)–(17), which give a necessary and sufficient characterization of all valid SELTZO abstract values. One copy of these consistency conditions is instantiated for each label v_i.

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

$$(s_v = 0) \vee (s_v = 1) \qquad e_v \geq e_{\min} - 1 \tag{10}$$

   We use $e = e_{\min} - 1$ to encode zero. Note that we do *not* impose an upper bound on e to ease handing of subnormal values, as explained in Section 5.1.
2. If a floating-point variable is zero (i.e., $e_v = e_{\min} - 1$), then its mantissa must consist entirely of zeros.

$$(e_v = e_{\min} - 1) \to [(\mathsf{nlz}_v = \mathsf{ntz}_v = p - 1) \wedge (\mathsf{nlo}_v = \mathsf{nto}_v = 0)] \tag{11}$$

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

$$[(\mathsf{nlz}_v > 0) \wedge (\mathsf{nlo}_v = 0)] \vee [(\mathsf{nlz}_v = 0) \wedge (\mathsf{nlo}_v > 0)] \tag{12}$$
$$[(\mathsf{ntz}_v > 0) \wedge (\mathsf{nto}_v = 0)] \vee [(\mathsf{ntz}_v = 0) \wedge (\mathsf{nto}_v > 0)] \tag{13}$$

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

$$(\mathsf{nlz}_v = \mathsf{ntz}_v = p - 1) \vee (\mathsf{nlz}_v + \mathsf{ntz}_v < p - 1) \tag{14}$$
$$(\mathsf{nlo}_v = \mathsf{nto}_v = p - 1) \vee (\mathsf{nlo}_v + \mathsf{nto}_v < p - 1) \tag{15}$$
$$(\mathsf{nlz}_v + \mathsf{nto}_v = p - 1) \vee (\mathsf{nlz}_v + \mathsf{nto}_v < p - 2) \tag{16}$$
$$(\mathsf{ntz}_v + \mathsf{nlo}_v = p - 1) \vee (\mathsf{ntz}_v + \mathsf{nlo}_v < p - 2) \tag{17}$$

The upper bound of $p - 2$ in the last two conditions expresses the constraint that, in a bit string of the form $00 \cdots 0b11 \cdots 1$, the middle bit b either belongs to the group of leading zeros or trailing ones. Thus, $\mathsf{nlz}_v + \mathsf{nto}_v \neq p - 2$.

We then adjoin a set of execution conditions to $S_{P,F}$ to constrain the possible outputs of each TwoSum gate conditioned on its inputs. Each of these sets comprises hundreds of inequalities collectively stated in Propositions 1 and 3, along with the lemmas given in the Appendix. A separate copy of the execution conditions must be instantiated on each TwoSum gate in F.

Finally, we add counterexample conditions to encode the *negation* of the property $P(F)$ that we wish to prove. These conditions must be formulated such that any concrete counterexample to the desired property $P(F)$ would

have SELTZO abstract values that violate the counterexample conditions. Any property that is expressible in as a logical combination of linear inequalities in the SELTZO variables can be used to construct these conditions. By Propositions 1 and 2, this class includes both nonoverlapping invariants of the form $x = \mathsf{RNE}(x + y)$ and relative error bounds of the form $|y| \leq 2^j \mathbf{u}^k |x|$.

By combining our procedure with an SMT solver for the theory `QF_LIA` of quantifier-free Presburger arithmetic, we now have all necessary technical tools to automatically verify correctness properties of FPANs. Note that our procedure is sound but not complete; successful verification proves that no counterexamples exist, but failure to verify does not prove that a concrete counterexample exists. However, an abstract counterexample serves as an excellent starting point from which to either construct a concrete counterexample or to prove none exists by adding additional lemmas on new input subdomains to the execution conditions.

5.1 Handling Subnormal Values

Note that our consistency conditions do not impose an upper bound on the exponent e_x of any abstract value. This means that the unsatisfiability of $S_{P,F}$ actually proves a stronger result: there are no counterexamples to $P(F)$ even in a hypothetical floating-point format with infinite exponent range. This key observation enables all of the analysis we have presented so far to generalize to subnormal inputs, as long as the property P depends only on differences between exponents $e_x - e_y$ and not absolute exponent values. In particular, all of the properties stated in Propositions 1 and 2 satisfy this requirement.

Proposition 4. *If $S_{P,F}$ is unsatisfiable and the property P is expressible using only $(s_v, \mathsf{nlz}_v, \mathsf{nlo}_v, \mathsf{ntz}_v, \mathsf{nto}_v)$ and exponent differences of the form $e_v - e_w$, then $P(F)$ holds for all finite inputs.*

Proof. We prove the contrapositive. Suppose there exists a concrete counterexample to $P(F)$ consisting of finite (possibly subnormal) floating-point input values $(x_1, x_2, \ldots, x_n)$. Any property P expressible in this form is invariant under a global shift of all exponents. Hence, for any sufficiently large k, the SELTZO abstraction of $(2^k x_1, 2^k x_2, \ldots, 2^k x_n)$ yields an abstract counterexample to $P(F)$ consisting only of normalized abstract values. We may ignore the possibility of overflow in this construction because our abstraction uses an unbounded exponent range. This proves that $S_{P,F}$ is satisfiable. □

6 Results

We apply the machinery developed in this paper to prove tight error bounds for both the current state-of-the-art double-double addition algorithm (`ddadd`), used in many software libraries [42,19,58], 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. FPANs for both algorithms are shown in Figure 1.

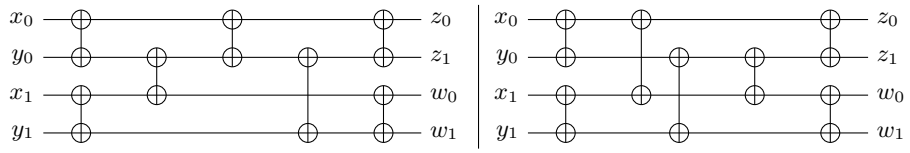

Fig. 7. Augmented network diagrams for ddadd (left) and madd (right), our new double-double addition algorithm, with error terms explicitly computed and named. The extra TwoSum gate used to compute these error terms serves only to facilitate our analysis and should not be included in an actual implementation of either algorithm.

Theorem 1. *Let* (x_0, x_1) *and* (y_0, y_1) *be floating-point expansions in the binary16, bfloat16, binary32, binary64, or binary128 format with* $x_0 = \mathsf{RNE}(x_0 + x_1)$ *and* $y_0 = \mathsf{RNE}(y_0 + y_1)$*. The ddadd algorithm, depicted in Figure 1 (left), computes a floating-point expansion* (z_0, z_1) *that approximates the exact sum* $x_0 + x_1 + y_0 + y_1$ *with relative error bounded above by* $(1 + 2\mathbf{u})2^{-(2p-2)} \approx 4\mathbf{u}^2$*.*

Theorem 2. *Let* (x_0, x_1) *and* (y_0, y_1) *be floating-point expansions in the binary16, bfloat16, binary32, binary64, or binary128 format with* $x_0 = \mathsf{RNE}(x_0 + x_1)$ *and* $y_0 = \mathsf{RNE}(y_0 + y_1)$*. The madd algorithm, depicted in Figure 1 (right), computes a floating-point expansion* (z_0, z_1) *that approximates the exact sum* $x_0 + x_1 + y_0 + y_1$ *with relative error bounded above by* $(1 + 2\mathbf{u})2^{-(2p-1)} \approx 2\mathbf{u}^2$*.*

We prove Theorems 1 and 2 by using the SELTZO abstraction to establish $P(F)$ for a suitably chosen property P given below. Some subsequent algebraic manipulation is then necessary to transform P into a true relative error bound that considers all discarded outputs.

Proof. Consider the augmented FPANs depicted in Figure 7, which include an extra TwoSum gate to compute the error terms (w_0, w_1). Using the SELTZO encodings stated in Propositions 1 and 2, we use an SMT solver to prove

$$P \coloneqq (x_0 = \mathsf{RNE}(x_0 + x_1)) \wedge (y_0 = \mathsf{RNE}(y_0 + y_1)) \rightarrow (|w_0| \leq 2^j \mathbf{u}^2 |z_0|) \tag{18}$$

where $j = 2$ for ddadd and $j = 1$ for madd. Moreover, by Proposition 3, we have $z_0 = \mathsf{RNE}(z_0, z_1)$ and $w_0 = \mathsf{RNE}(w_0, w_1)$ because (z_0, z_1) and (w_0, w_1) are TwoSum outputs. It follows that $w_0 + w_1 = (1 + \delta_w)w_0$ and $z_0 + z_1 = (1 + \delta_z)z_0$ for some $|\delta_w|, |\delta_z| \leq \mathbf{u}$. Hence, the relative error of the sums computed by ddadd and madd is bounded above by:

$$\begin{aligned} \frac{|(z_0 + z_1) - (x_0 + x_1 + y_0 + y_1)|}{|x_0 + x_1 + y_0 + y_1|} &= \frac{|(1 + \delta_w)w_0|}{|(1 + \delta_z)z_0 + (1 + \delta_w)w_0|} \\ &\leq \frac{(1 + \mathbf{u})|w_0|}{(1 - \mathbf{u})(|z_0| - |w_0|)} \leq \frac{(1 + \mathbf{u})2^j \mathbf{u}^2}{1 - (1 - \mathbf{u})2^j \mathbf{u}^2} \leq (1 + 2\mathbf{u})2^j \mathbf{u}^2 \end{aligned} \tag{19}$$

This expression is asymptotically equivalent to $4\mathbf{u}^2 + O(\mathbf{u}^3)$ for ddadd and $2\mathbf{u}^2 + O(\mathbf{u}^3)$ for madd. □

In Table 2, we compare the speed of directly verifying the property P defined above using a variety of floating-point theory solvers (QF_FP) to resolving

FPAN	Format	Z3	CVC5	MathSAT	Bitwuzla	Colibri2	SELTZO
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 2. Execution time for various SMT solvers to directly verify property P in a floating-point theory (`QF_FP`) compared to deciding the satisfiability of $S_{P,F}$ in quantifier-free Presburger arithmetic (`QF_LIA`). 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.

the satisfiability problems $S_{P,F}$ constructed by the SELTZO abstraction with a Presburger arithmetic solver (`QF_LIA`). We benchmark a portfolio of state-of-the-art SMT solvers using the latest software versions available at the time of writing, with the SELTZO abstraction implemented using the Z3 Python API. Our implementation is freely available at https://github.com/dzhang314/FPANVerifier. In all cases, the SELTZO abstraction produces problems that are many orders of magnitude faster to solve. We also note that our SELTZO solve times remain constant with respect to the precision p of the floating-point format, enabling scalability to wide formats that are intractable for bit-blasting.

Note that our bound of $4\mathbf{u}^2$ for ddadd is slightly looser than the $3\mathbf{u}^2$ bound proven by Joldes, Muller, and Popescu [35]. Our use of the SELTZO abstraction restricts us to proving bounds of the form $2^j\mathbf{u}^k + O(\mathbf{u}^{k+1})$, which we say are *tight to the nearest bit* if the leading constant factor is the smallest possible power of two (i.e., j is as small as possible). The following example due to Muller and Rideau [46] shows that our ddadd bound ($j = 2$) is tight to the nearest bit:

$$x_0 \coloneqq 1 \qquad x_1 \coloneqq \mathbf{u} - \mathbf{u}^2 \qquad y_0 \coloneqq -\frac{1}{2} + \frac{\mathbf{u}}{2} \qquad y_1 \coloneqq -\frac{\mathbf{u}^2}{2} + \mathbf{u}^3 \tag{20}$$

It is straightforward to verify that the result returned by ddadd has relative error $\approx 3\mathbf{u}^2$ on these inputs. Moreover, the following example shows that our madd bound ($j = 1$) is also tight to the nearest bit:

$$x_0 \coloneqq 1 + 2\mathbf{u} \qquad x_1 \coloneqq -\frac{\mathbf{u}}{2} - 2\mathbf{u}^2 \qquad y_0 \coloneqq -\mathbf{u} \qquad y_1 \coloneqq -\frac{\mathbf{u}^2}{2} - \mathbf{u}^3 \tag{21}$$

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

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

6.1 Ablated Abstractions

It is natural to ask whether simpler abstractions than SELTZO can prove Theorems 1 and 2. To investigate this question, we consider two ablated abstractions that use subsets of the SELTZO variables $(s_x, e_x, \mathsf{nlz}_x, \mathsf{nlo}_x, \mathsf{ntz}_x, \mathsf{nto}_x)$.

Definition 3. *Let x be a floating-point number. Using the notation of Definition 2, the **SE abstraction** of x is the 2-tuple (s_x, e_x), and the **SETZ abstraction** of x is the 3-tuple $(s_x, e_x, \mathsf{ntz}_x)$.*

The SE and SETZ abstractions are natural simplifications of SELTZO that omit some or all information about mantissa bit patterns. Notably, the SETZ abstraction is still expressive enough to capture all of the conditions of Proposition 1, none of which refer to leading bits or trailing ones.

To perform FPAN verification with these simpler abstractions, we construct SE and SETZ satisfiability problems $S_{P,F}$ using special sets of reduced lemmas given in the Appendix that make no reference to the omitted variables nlz_x, nlo_x, nto_x, and (for the SE abstraction) ntz_x. We then find the minimum value of j for which the property P defined above holds in each abstraction. These values are reported in Table 3. While SE and SETZ are unable to match the tight error bounds provable in the SELTZO abstraction, they still establish nontrivial bounds that are difficult to prove by hand. However, these results show that modeling leading mantissa bits is necessary to prove tight FPAN error bounds.

7 Related Work

Automatic floating-point verification. Most existing techniques for automatic verification of floating-point algorithms rely on abstractions derived from rational or real arithmetic, such as interval enclosures, polyhedra, and relational domains [16,44,15,54]. These abstractions are fundamentally inapplicable to FPANs. The notion of an error-free transformation like TwoSum only exists in finite-precision rounded arithmetic and has no semantically equivalent analogue in exact real arithmetic. Indeed, in real arithmetic, $\mathsf{TwoSum}(x, y) = (x + y, 0)$ is a trivial operation. As our results show, precise reasoning about FPANs requires explicitly modeling the interaction between the sum and error terms, taking into account the shape of the mantissa, which these abstractions cannot express.

Interactive floating-point verification. To the best of our knowledge, the only existing methods that can reason about error-free transformations and FPANs use

interactive, as opposed to automatic, theorem provers. Tools such as Flocq [13] and Gappa [22] have been used to verify algorithms involving FPANs [19,58], but they require a high degree of user expertise to construct sophisticated proof scripts. The SELTZO abstraction provides a complementary approach that could be integrated with these tools to provide a greater degree of automation.

Other approaches to high-precision arithmetic. Libraries for arbitrary-precision arithmetic, including GMP, MPFR, and Arb, make no internal use of floating-point operations [31,28,34]. 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 moderate precision levels (2–8 machine words), these algorithms are many times slower than FPANs, requiring complex branching logic and more operations per bit on average. While FPANs are hard to discover and prove correct, they enable high-performance branch-free arithmetic that massively accelerates high-precision scientific applications.

Scalable abstraction in other domains. Recent work on the Bitwuzla SMT solver [50] 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. One notable difference is that our approach does not require abstraction refinement tailored to a specific mantissa width or FPAN.

Sorting networks. FPANs are close analogues of sorting networks. Although they compute different operations, both are branch-free algorithms that accelerate sorting or accumulation of a fixed number of inputs in a data-parallel fashion. Because they both occupy a similar large combinatorial space, techniques for discovering and optimizing sorting networks, such as evolutionary algorithms [23], can also be applied to FPANs. Our work on the SELTZO abstraction provides an efficient correctness check that can be used to perform such a search.

8 Conclusion

In this paper, we defined *floating-point accumulation networks* (FPANs) and showed that FPANs are key building blocks used to construct extended-precision floating-point algorithms. To address the difficulty of analyzing floating-point rounding errors, which critically affects the correctness of FPANs, we introduced the SELTZO abstraction to enable efficient and precise automatic reasoning about FPANs using SMT solvers. Using the SELTZO abstraction, we developed computer-verified proofs of a number of FPAN correctness properties. In particular, we automatically and rigorously proved that madd, a novel FPAN for double-double addition, is simultaneously faster and more accurate than ddadd, the previous best known algorithm.

Acknowledgments. The authors thank Matthew Sotoudeh, Michael Paperr, Alexander J. Root, and the anonymous CAV reviewers for insightful comments on early drafts of this paper. We also thank Alan H. Karp and David H. Bailey for helpful discussions, references, and advice.

Disclosure of Interests. The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

1. WebAssembly Core Specification, https://www.w3.org/TR/wasm-core-2/
2. IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985 pp. 1–20 (1985). https://doi.org/10.1109/IEEESTD.1985.82928
3. IEEE standard for floating-point arithmetic. IEEE Std 754-2008 pp. 1–70 (2008). https://doi.org/10.1109/IEEESTD.2008.4610935
4. IEEE standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) pp. 1–84 (2019). https://doi.org/10.1109/IEEESTD.2019.8766229
5. Babuška, I.: Numerical stability in problems of linear algebra. SIAM Journal on Numerical Analysis **9**(1), 53–77 (1972). https://doi.org/10.1137/0709008, https://doi.org/10.1137/0709008
6. Bailey, D.H., Krasny, R., Pelz, R.: Multiple precision, multiple processor vortex sheet roll-up computation. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (United States) (12 1993), https://www.osti.gov/biblio/54379
7. Bailey, D.H.: High-precision software directory. https://www.davidhbailey.com/dhbsoftware/ (2024)
8. Bailey, D.H., Borwein, J.M.: Hand-to-hand combat with thousand-digit integrals. Journal of Computational Science **3**(3), 77–86 (2012). https://doi.org/https://doi.org/10.1016/j.jocs.2010.12.004, https://www.sciencedirect.com/science/article/pii/S1877750310000773, scientific Computation Methods and Applications
9. Bailey, D.: High-precision floating-point arithmetic in scientific computation. Computing in Science & Engineering **7**(3), 54–61 (2005). https://doi.org/10.1109/MCSE.2005.52
10. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) 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. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
11. Boldo, S., Graillat, S., Muller, J.M.: On the robustness of the 2sum and fast2sum algorithms. ACM Trans. Math. Softw. **44**(1) (Jul 2017). https://doi.org/10.1145/3054947, https://doi.org/10.1145/3054947
12. Boldo, S., Joldes, M., Muller, J.M., Popescu, V.: Formal verification of a floating-point expansion renormalization algorithm. In: Ayala-Rincón, M., Muñoz, C.A.

(eds.) Interactive Theorem Proving. pp. 98–113. Springer International Publishing, Cham (2017)
13. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic. pp. 243–252 (2011). https://doi.org/10.1109/ARITH.2011.40
14. Boldo, S., Muller, J.M.: Exact and approximated error of the FMA. IEEE Transactions on Computers **60**(2), 157–164 (2011). https://doi.org/10.1109/TC.2010.139
15. Chapoutot, A.: Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables. In: Cousot, R., Martel, M. (eds.) Static Analysis. pp. 184–200. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_12
16. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: Ramalingam, G. (ed.) Programming Languages and Systems. pp. 3–18. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_2
17. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 93–107. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
18. Collange, C., Joldes, M., Muller, J.M., Popescu, V.: Parallel floating-point expansions for extended-precision GPU computations. In: 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP). pp. 139–146 (2016). https://doi.org/10.1109/ASAP.2016.7760783
19. Daramy-Loirat, C., Defour, D., de Dinechin, F., Gallet, M., Gast, N., Lauter, C., Muller, J.M.: CR-LIBM A library of correctly rounded elementary functions in double-precision. Research report, LIP, (Dec 2006), https://ens-lyon.hal.science/ensl-01529804
20. De Moura, L., Bjørner, N.: 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. p. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
21. Dekker, T.J.: A floating-point technique for extending the available precision. Numerische Mathematik **18**(3), 224–242 (Jun 1971). https://doi.org/10.1007/BF01397083, https://doi.org/10.1007/BF01397083
22. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying floating-point implementations using Gappa (Dec 2007), https://ens-lyon.hal.science/ensl-00200830, working paper or preprint
23. Dobbelaere, B.: SorterHunter (2024), https://github.com/bertdobbelaere/SorterHunter
24. ECMA International: Standard ECMA-262 - ECMAScript Language Specification. 15 edn. (2024), https://ecma-international.org/publications-and-standards/standards/ecma-262/
25. Elrod, C., Févotte, F.: Accurate and Efficiently Vectorized Sums and Dot Products in Julia (Aug 2019), https://hal.science/hal-02265534, version submitted to the Correctness2019 workshop
26. Evstigneev, N., Ryabkov, O., Bocharov, A., Petrovskiy, V., Teplyakov, I.: 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). https://doi.org/https://doi.org/10.1016/j.cam.2022.114434, https://www.sciencedirect.com/science/article/pii/S0377042722002047
27. Python Software Foundation: The Python standard library: Built-in functions. https://docs.python.org/3/library/functions.html (2001–2025)

28. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: Mpfr: A multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**(2), 13–es (Jun 2007). https://doi.org/10.1145/1236463.1236468, https://doi.org/10.1145/1236463.1236468
29. Frolov, A.M.: High-precision, variational, bound-state calculations in coulomb three-body systems. Phys. Rev. E **62**, 8740–8745 (Dec 2000). https://doi.org/10.1103/PhysRevE.62.8740, https://link.aps.org/doi/10.1103/PhysRevE.62.8740
30. Frolov, A.M., Bailey, D.H.: 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 (apr 2003). https://doi.org/10.1088/0953-4075/36/9/315, https://dx.doi.org/10.1088/0953-4075/36/9/315
31. Granlund, T., Team, G.D.: GNU MP 6.0 Multiple Precision Arithmetic Library. Samurai Media Limited, London, GBR (2015)
32. He, Y., Ding, C.H.Q.: Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications. The Journal of Supercomputing **18**(3), 259–277 (Mar 2001). https://doi.org/10.1023/A:1008153532043, https://doi.org/10.1023/A:1008153532043
33. Hida, Y., Li, X., Bailey, D.: Algorithms for quad-double precision floating point arithmetic. In: Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001. pp. 155–162 (Jun 2001). https://doi.org/10.1109/ARITH.2001.930115, https://ieeexplore.ieee.org/document/930115, iSSN: 1063-6889
34. Johansson, F.: Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. IEEE Transactions on Computers **66**, 1281–1292 (2017). https://doi.org/10.1109/TC.2017.2690633
35. Joldes, M., Muller, J.M., Popescu, V.: Tight and rigorous error bounds for basic building blocks of double-word arithmetic. ACM Trans. Math. Softw. **44**(2) (Oct 2017). https://doi.org/10.1145/3121432, https://doi.org/10.1145/3121432
36. Joldes, M., Muller, J.M., Popescu, V., Tucker, W.: CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications. In: 5th International Congress on Mathematical Software (ICMS). Berlin, Germany (Jul 2016), https://hal.science/hal-01312858
37. Kahan, W.: Pracniques: further remarks on reducing truncation errors. Commun. ACM **8**(1), 40 (Jan 1965). https://doi.org/10.1145/363707.363723, https://doi.org/10.1145/363707.363723
38. Karp, A.H., Markstein, P.: High-precision division and square root. ACM Trans. Math. Softw. **23**(4), 561–589 (Dec 1997). https://doi.org/10.1145/279232.279237, https://doi.org/10.1145/279232.279237
39. Knuth, D.E.: The Art of Computer Programming, Volume II: Seminumerical Algorithms. Addison-Wesley (1969), https://www.worldcat.org/oclc/310551264
40. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)
41. Lauter, C.Q.: Basic building blocks for a triple-double intermediate format. Research Report RR-5702, LIP RR-2005-38, INRIA, LIP (Sep 2005), https://inria.hal.science/inria-00070314
42. Li, X.S., Demmel, J.W., Bailey, D.H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S.Y., Kapur, A., Martin, M.C., Thompson, B.J., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision blas. ACM Trans. Math. Softw. **28**(2), 152–205 (Jun 2002). https://doi.org/10.1145/567806.567808, https://doi.org/10.1145/567806.567808

43. Lu, M., He, B., Luo, Q.: Supporting extended precision on graphics processors. In: Proceedings of the Sixth International Workshop on Data Management on New Hardware. pp. 19–26. DaMoN '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1869389.1869392, http://doi.acm.org/10.1145/1869389.1869392
44. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Trans. Program. Lang. Syst. **30**(3) (May 2008). https://doi.org/10.1145/1353445.1353446, https://doi.org/10.1145/1353445.1353446
45. Muller, J.M., Brunie, N., De Dinechin, F., Jeannerod, C.P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., Torres, S.: Handbook of Floating-Point Arithmetic. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-76526-6, http://link.springer.com/10.1007/978-3-319-76526-6
46. Muller, J.M., Rideau, L.: 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) (Feb 2022). https://doi.org/10.1145/3484514, https://doi.org/10.1145/3484514
47. Møller, O.: Quasi double-precision in floating point addition. BIT Numerical Mathematics **5**(1), 37–50 (Mar 1965). https://doi.org/10.1007/BF01975722, https://doi.org/10.1007/BF01975722
48. Neumaier, A.: 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). https://doi.org/https://doi.org/10.1002/zamm.19740540106, https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.19740540106
49. Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 3–17. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_1, https://doi.org/10.1007/978-3-031-37703-7_1
50. Niemetz, A., Preiner, M., Zohar, Y.: Scalable bit-blasting with abstractions. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification. pp. 178–200. Springer Nature Switzerland, Cham (2024)
51. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM Journal on Scientific Computing **26**(6), 1955–1988 (2005). https://doi.org/10.1137/030601818, https://doi.org/10.1137/030601818
52. Priest, D.: Algorithms for arbitrary precision floating point arithmetic. In: [1991] Proceedings 10th IEEE Symposium on Computer Arithmetic. pp. 132–143 (1991). https://doi.org/10.1109/ARITH.1991.145549
53. The Julia Project: Julia standard library: Arrays. https://docs.julialang.org/en/v0.6/stdlib/arrays (2017)
54. Rivera, J., Franchetti, F., Püschel, M.: Floating-point tvpi abstract domain. Proc. ACM Program. Lang. **8**(PLDI) (Jun 2024). https://doi.org/10.1145/3656395, https://doi.org/10.1145/3656395
55. Rump, S.M., Lange, M.: On the definition of unit roundoff. BIT Numerical Mathematics **56**(1), 309–317 (Mar 2016). https://doi.org/10.1007/s10543-015-0554-0, https://doi.org/10.1007/s10543-015-0554-0
56. Sarnoff, J.: DoubleFloats.jl (2024), https://github.com/JuliaMath/DoubleFloats.jl
57. Shewchuk, J.R.: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Discrete & Computational Geometry **18**(3), 305–363 (Oct 1997). https://doi.org/10.1007/PL00009321, https://doi.org/10.1007/PL00009321

58. Sibidanov, A., Zimmermann, P., Glondu, S.: The CORE-MATH Project. In: 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH). pp. 26–34. IEEE, virtual, France (Sep 2022). https://doi.org/10.1109/ARITH54963.2022.00014, https://inria.hal.science/hal-03721525
59. Veltkamp, G.W.: ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie. Tech. Rep. 22, Technische Hogeschool Eindhoven (1968)
60. Veltkamp, G.W.: ALGOL procedures voor het rekenen in dubbele lengte. Tech. Rep. 21, Technische Hogeschool Eindhoven (1969)
61. Zhang, D.K.: MultiFloats.jl (2024), https://github.com/dzhang314/MultiFloats.jl

Appendix

Let x and y be precision-p floating-point numbers, and let $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$. Assume that x, y, s, and e are all normalized or zero. The following lemmas constrain the possible values of x, y, s, and e using the SE, SETZ, and SELTZO abstractions. Note that TwoSum is a commutative operation (i.e., $\mathsf{TwoSum}(x, y) = \mathsf{TwoSum}(y, x)$), so all of the following results also hold with the roles of x and y are reversed. We omit symmetric statements for brevity.

Lemma Family Z

Lemmas in Family Z apply when one or both of the inputs (x, y) are zero. These Lemmas support our SE, SETZ, and SELTZO results.

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

Lemma Family SE-I

Lemma SE-I gives a sufficient condition for the inputs (x, y) to be returned unchanged by TwoSum. This Lemma supports only our SE results and is superseded by a more precise Lemma in the SETZ and SELTZO abstractions. Note that, unlike the SETZ and SELTZO abstractions, the SE abstraction is not strong enough to state necessary and sufficient conditions for $(x, y) = \mathsf{TwoSum}(x, y)$.

Lemma SE-I: If $|e_x - e_y| < p + 1$ or $|e_x - e_y| = p + 1$ and $s_x = s_y$, then $(x, y) = \mathsf{TwoSum}(x, y)$.

Lemma Family SE-S

Lemmas in Family SE-S apply to SE abstract inputs with the same sign. These Lemmas support only our SE results and are superseded by more precise Lemmas in the SETZ and SELTZO abstractions. In this Lemma Family, we implicitly assume that x and y are nonzero with $s_x = s_y$ throughout.

Lemma SE-S1: If $e_x = e_y + p$, then one of the following statements holds:

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

Lemma SE-S2: If $e_x = e_y + (p - 1)$, then one of the following statements holds:

1. $s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, and $e = +0.0$.
2. $s_s = s_x$, $e_x \leq e_s \leq e_x + 1$, and $e_y - (p-1) \leq e_e \leq e_x - p$.

Lemma SE-S3: If $e_x = e_y + (p-2)$, then one of the following statements holds:

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

Lemma SE-S4: If $e_x > e_y$ and $e_x < e_y + (p-2)$, then one of the following statements holds:

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

Lemma SE-S5: If $e_x = e_y$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x + 1$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x + 1$, and $e_e = e_x - (p-1)$.

Lemma Family SE-D

Lemmas in Family SE-D apply to SE abstract inputs with different signs. These Lemmas support only our SE results and are superseded by more precise Lemmas in the SETZ and SELTZO abstractions. In this Lemma Family, we implicitly assume that x and y are nonzero with $s_x \neq s_y$ throughout.

Lemma SE-D1: If $e_x = e_y + (p+1)$, then one of the following statements holds:

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

Lemma SE-D2: If $e_x = e_y + p$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x - 1$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x - 1$, $s_e = s_y$, and $e_y - (p-1) \leq e_e \leq e_x - (p+2)$.
3. $s_s = s_x$, $e_s = e_x - 1$, $s_e \neq s_y$, and $e_y - (p-1) \leq e_e \leq e_x - (p+1)$.
4. $s_s = s_x$, $e_s = e_x$, $s_e \neq s_y$, and $e_y - (p-1) \leq e_e \leq e_x - p$.
5. $s = x$ and $e = y$.

Lemma SE-D3: If $e_x > e_y + 1$ and $e_x < e_y + p$, then one of the following statements holds:

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

Lemma SE-D4: If $e_x = e_y + 1$, then one of the following statements holds:

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

Lemma SE-D5: If $e_x = e_y$, then one of the following statements holds:

1. $s = +0.0$ and $e = +0.0$.
2. $e_x - (p-1) \leq e_s \leq e_x - 1$ and $e = +0.0$.

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. This Lemma supports both our SETZ and SELTZO results.

Lemma SETZ-I: Assume x and y are nonzero. $(s, e) = (x, y)$ 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

Some SETZ and SELTZO lemmas admit simpler statements by defining the *trailing exponent* $f_x \coloneqq e_x - (p - \mathsf{ntz}_x - 1)$. This is the place value of the last nonzero mantissa bit of x, which acts like the dual of e_x, bounding the mantissa from below.

Lemmas in Family SETZ-F apply to addends with the same trailing exponent (i.e., $f_x = f_y$). These Lemmas support our SETZ and SELTZO results. We assume throughout this Lemma Family that x and y are nonzero.

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

1. $s_s = s_x$, $e_s = e_x$, $f_x + 1 \le f_s \le e_x - 1$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_x + 1 \le f_s \le e_y$, 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-X: If $s_x = s_y$, $f_x = f_y$, and $e_x = e_y + 1$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x$, $f_x + 1 \le f_s \le e_x - 2$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_x + 1 \le f_s \le e_y$, 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 one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x + 1$, $f_x + 1 \le f_s \le e_x$, 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 one of the following statements holds:

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

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

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-X: If $s_x \neq s_y$, $f_x = f_y$, and $e_x = e_y + 1$, then one of the following statements holds:

1. For each k where $f_x + 1 \leq k \leq e_x - 1$: $s_s = s_x$, $e_s = k$, $f_x + 1 \leq f_s \leq k$, 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 one of the following statements holds:

1. $s = +0.0$ and $e = +0.0$.
2. For each k where $f_x + 1 \leq k \leq e_x - 1$: $f_x + 1 \leq f_s \leq k$, $e_s = k$, and $e = +0.0$.

Remaining SETZ and SELTZO Lemmas

To avoid occupying an excessive number of pages, we will only state a small sample of the remaining SETZ and SELTZO lemmas in this paper to provide a general idea of the results that can be proven in various subdomains of the SELTZO abstraction. The full list of lemmas used to prove the main results of this paper is provided in the files `se_lemmas.py`, `setz_lemmas.py`, and `seltzo_lemmas.py` in the GitHub repository https://github.com/dzhang314/FPANVerifier.

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 one of the following statements holds:

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 one of the following statements holds:

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

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

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

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 one of the following statements holds:

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 one of the following statements holds:

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-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 one of the following statements holds:

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-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 one of the following statements holds:

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 $f_e = f_y$.