Semantics, Operations, and Properties of P3109 Floating-Point Representations in Lean

Tung-Che Chang*, Sehyeok Park*, Jay P. Lim†, and Santosh Nagarakatte*
*Rutgers University, Piscataway, NJ, USA
tc.chang@rutgers.edu, sp2044@cs.rutgers.edu, santosh.nagarakatte@cs.rutgers.edu
†University of California, Riverside, Riverside, CA, USA
jlim@ucr.edu

***Abstract*—The upcoming IEEE P3109 standard for low-precision floating-point arithmetic can become the foundation of future machine learning hardware and software. Unlike IEEE-754, P3109 introduces a parametric framework defined by bitwidth, precision, signedness, and domain. This flexibility yields a vast combinatorial space of formats, some with as little as one bit of precision, alongside novel features such as stochastic rounding and saturation arithmetic. These deviations create a unique verification gap that this paper intends to address.**

This paper presents `FLoPS`, Formalization in Lean of the P3109 Standard, which is a formal model of P3109 in Lean. Our work serves as a rigorous, machine-checked specification that facilitates deep analysis of the standard. We demonstrate the model's utility by verifying foundational properties and analyzing key algorithms within the P3109 context. Specifically, we reveal that `FastTwoSum` exhibits a novel property of computing the exact "overflow error" under saturation using any rounding mode, whereas previously established properties of the `ExtractScalar` algorithm fail for formats with one bit of precision. We also provide a verified executable bit-level kernel, with refinement proofs for rounding, saturation, projection, and the core arithmetic operations. This work provides a verified foundation for reasoning about P3109 and enables formal verification of future numerical hardware and software. Our Lean development is open source and publicly available.

I. Introduction

The upcoming IEEE P3109 standard [21] introduces a flexible framework for low-precision, binary floating-point (FP) formats tailored for machine learning. In contrast to IEEE-754 [20], a P3109 format is defined by four parameters: bitwidth K (the total number of bits), precision P (how accurately real numbers can be represented), signedness Σ (whether negative numbers are included), and domain Δ (finite or extended, determining whether infinities are included). This parametric approach results in a large space of potential formats with distinct encoding schemes. Each format has 2^K bit-patterns, which are called code points. Each code point encodes a floating-point datum, which is a value from the mathematical object of closed extended reals (set of real numbers augmented with positive infinity, negative infinity, and not-a-number). Every format contains a single zero and a single `NaN` (not a number) datum, with infinities determined by the format. These features are designed to maximize the number of real numbers representable with low-bitwidth encodings. To address the limited precision and dynamic range of low-bitwidth formats, the P3109 standard provides diverse specifications for mapping real numbers to representable datums. P3109 extends the rounding modes supported by IEEE-754 (*i.e.*, round-down (RD), round-up (RU), round-toward-zero (RZ), round-to-nearest-ties-to-even (RNE), and round-to-nearest-ties-away (RNA)) with round-to-odd (RTO) and three variants of stochastic rounding (SR), which are used to enhance convergence and stability in machine learning training [17]. P3109 operations also use explicit saturation modes that determine whether out-of-range rounded results become a finite boundary datum, an infinity, or `NaN`.

Formal verification of IEEE-754 floating-point arithmetic is a well-established field [6], [7]. Mainstream FP verification literature assumes a "reasonably large" precision, an assumption that could fail in the context of machine learning formats where precision may drop to as little as one bit. Therefore, existing formalizations of floating-point arithmetic under IEEE-754 need to be significantly modified to handle various properties unique to P3109.

FLoPS. This paper develops a formalization bespoke to P3109 that supports verification of its parametric formats, operations, and future applications. We describe `FLoPS`, a **F**ormalization in **L**ean **o**f the **P**3109 **S**tandard. We chose Lean 4 for its extensive `Mathlib` libraries, including `EReal` for the real-and-infinity component of P3109's reference domain, and its expressive tactic language.

Our primary objective with `FLoPS` is to establish a mathematically precise ground truth for P3109 formats and projection, which can serve as a reference model for future P3109 implementations. As P3109-compliant hardware accelerators for machine learning are actively being developed, a verified specification is essential for ensuring that register-transfer-level (RTL) implementations faithfully realize the standard's semantics. To avoid the complexity of raw bit-level reasoning, we establish a three-way equivalence between code points, our algebraic model, and the P3109 datum set of each format (Section III), which serves as a natural bridge: properties proven at the mathematical level transfer directly to the concrete bit-patterns that hardware uses, and the verified projection pipeline can serve as a reference for equivalence checking against hardware designs.

To show our model's practical utility, we formalize the properties of the `FastTwoSum` [16] algorithm in the context of P3109 (Section IV). Under the round-to-nearest round-

https://doi.org/

ing mode, FastTwoSum computes the exact rounding error of FP addition using only FP operations. Through our formalization, we discovered that FastTwoSum exhibits a novel property in the presence of saturation. When an overflowing sum is clamped to the corresponding finite boundary datum, FastTwoSum computes exactly the difference between the real sum and that datum, regardless of the rounding modes used by its three operations. We extend our analysis of FastTwoSum to develop the first formal analysis of the FP splitting technique ExtractScalar [31]. Our analysis reveals that previously established properties of ExtractScalar do not extend to P3109 formats with one-bit precision (Section V).

The FLoPS framework is open source and publicly available [9], [10]. While developing FLoPS, we discovered multiple errors in the draft P3109 standard and reported them to the P3109 working group; these errors have subsequently been fixed.

II. Background

We provide a brief background on both the IEEE-754 standard and the P3109 standard highlighting the key differences in decoding a code point to a floating-point datum.

A. The IEEE 754 Standard

The binary IEEE-754 format [20] is characterized by its precision P and exponent bounds emin and emax. Let $\mathcal{V}_f \subseteq \mathbb{R}$ denote its finite representable numbers. A member $x \in \mathcal{V}_f$ has the form $x = (-1)^s m 2^{E-P+1}$, where $m, E \in \mathbb{Z}$ and $0 \leq m < 2^P$. A nonzero value is *normal* when $2^{P-1} \leq m < 2^P$ and $\texttt{emin} \leq E \leq \texttt{emax}$; it is *subnormal* when $0 < m < 2^{P-1}$ and $E = \texttt{emin}$.

A K-bit binary encoding consists of a sign bit, a $W = K - P$ bit exponent field, and $P - 1$ trailing significand bits. For a normal value, the real-value formula above uses the canonical (unbiased) exponent E. Encoding adds the format bias B and stores $E_{biased} = E + B$ in the exponent field; decoding subtracts the bias to recover $E = E_{biased} - B$. Normal encodings also omit the leading significand bit. An all-zero exponent field identifies zero or a subnormal, while an all-ones exponent field identifies infinities and NaNs. Because the sign is stored explicitly, IEEE-754 has distinct encodings for $+0$ and -0.

For rounding real results, we use the ordered numerical set $\overline{\mathcal{V}}_f = \mathcal{V}_f \cup \{-\infty, +\infty\}$; NaNs are IEEE-754 values but are excluded from this set because they are unordered. Arithmetic results generally require a rounding function $\circ : \mathbb{R} \to \overline{\mathcal{V}}_f$. For a real r, let $pred(r)$ and $succ(r)$ denote its neighboring elements in $\overline{\mathcal{V}}_f$. Rounding is *faithful* when it preserves representable inputs and otherwise selects one of these neighbors. A rounding function is *monotonic* when it preserves order: $r_1 \leq r_2$ implies $\circ(r_1) \leq \circ(r_2)$. IEEE-754 defines the five rounding directions RD, RU, RZ, RNE, and RNA. The directed modes select a neighbor according to sign and direction; the nearest modes select the closer neighbor, breaking midpoint ties toward an even significand (RNE) or away from zero (RNA).

TABLE I: Mappings from code points to their corresponding floating-point datums for the Binary4p2se signed extended format and the Binary4p2sf signed finite format. Each format has 4 bits in total and 2 bits of precision. For comparison, we also show an IEEE-754 format with one sign bit, two exponent bits, and two bits of precision. For each entry, the note column indicates whether it is zero (zero), subnormal (sub.), normal (norm.), infinity (± inf.), or NaN (nan).

Code	Binary4p2se		Binary4p2sf		IEEE-754	
point	Datum	Note	Datum	Note	Datum	Note
0000	0	zero	0	zero	$+0$	zero
0001	0.25	sub.	0.25	sub.	0.5	sub.
0010	0.5	norm.	0.5	norm.	1	norm.
0011	0.75	norm.	0.75	norm.	1.5	norm.
0100	1	norm.	1	norm.	2	norm.
0101	1.5	norm.	1.5	norm.	3	norm.
0110	2	norm.	2	norm.	$+\infty$	+inf.
0111	$+\infty$	+inf.	3	norm.	NaN	nan
1000	NaN	nan	NaN	nan	-0	zero
1001	-0.25	sub.	-0.25	sub.	-0.5	sub.
1010	-0.5	norm.	-0.5	norm.	-1	norm.
1011	-0.75	norm.	-0.75	norm.	-1.5	norm.
1100	-1	norm.	-1	norm.	-2	norm.
1101	-1.5	norm.	-1.5	norm.	-3	norm.
1110	-2	norm.	-2	norm.	$-\infty$	-inf.
1111	$-\infty$	-inf.	-3	norm.	NaN	nan

IEEE-754 resolves overflow, where the real value being encoded is greater than the largest finite representable value $M^{hi} = (2^P - 1)2^{\texttt{emax}-P+1}$, by rounding between the corresponding finite boundary and infinity. For example, $r > M^{hi}$ rounds to M^{hi} under RD or RZ and to $+\infty$ under RU.

B. The IEEE P3109 Standard

The upcoming P3109 standard [21] allows for the derivation of a format f via a tuple of four parameters: bitwidth (K), precision (P), signedness (Σ), and domain (Δ). The bitwidth K specifies the total storage size and must be at least 3 bits. The precision P must be at least 1. P is further constrained by the signedness parameter $\Sigma \in \{\texttt{Signed}, \texttt{Unsigned}\}$: P must be strictly less than K for $\Sigma = \texttt{Signed}$ and must be no greater than K otherwise.

Σ determines whether the datum set contains negative reals, while the domain $\Delta \in \{\texttt{Finite}, \texttt{Extended}\}$ determines whether it contains infinities. Finite formats contain finite real datums and NaN; Extended formats additionally contain both infinities when Signed, and only $+\infty$ when Unsigned.

Let $\mathbb{R}_\omega = \mathbb{R} \cup \{-\infty, +\infty, \texttt{NaN}\}$. Each format f has a datum set $\mathcal{D}_f \subseteq \mathbb{R}_\omega$ and a bijective encoding from its datums to 2^K integer code points. Thus, a code point represents a datum; it is not the datum itself. We write M^{lo} and M^{hi} for the minimum and maximum finite datums, respectively, of the format under discussion.

Table I presents the mapping between code points and decoded datums for two P3109 configurations: Binary4p2se ($\Sigma = \texttt{Signed}$, $\Delta = \texttt{Extended}$, $K = 4$ bitwidth, $P = 2$

bits of precision) and `Binary4p2sf` ($\Sigma =$ `Signed`, $\Delta =$ `Finite`, $K = 4$ bitwidth, $P = 2$ bits of precision). To facilitate comparison, we also provide the decoding for an IEEE-754 format with one sign bit, two exponent bits, and two bits of precision. As illustrated in the table, a code point is the bit-pattern (*i.e.*, an integer) encoding a particular floating-point datum representable by the P3109 format.

The P3109 standard uses one code point to represent each distinct datum. Unlike the IEEE-754 standard, P3109 explicitly defines one code point to represent the datum zero and a single code point for the `NaN` datum. The code points corresponding to special datums depend on the parameters K, Σ, and Δ, as shown in Table II. Finite configurations (*i.e.*, $\Delta =$ `Finite`) do not include $\pm\infty$, and unsigned configurations (*i.e.*, $\Sigma =$ `Unsigned`) do not include $-\infty$.

TABLE II: Code points for special datums at bitwidth K.

Datum	Signed Extended	Signed Finite	Unsigned Extended	Unsigned Finite
`NaN`	2^{K-1}	2^{K-1}	$2^K - 1$	$2^K - 1$
$+\infty$	$2^{K-1} - 1$	N/A	$2^K - 2$	N/A
$-\infty$	$2^K - 1$	N/A	N/A	N/A

In the case of $\Sigma =$ `Signed` formats, P3109 maps the `NaN` datum to the code point 2^{K-1}, the code point that IEEE-754 uses for -0. In Table I, this corresponds to the code point 1000. All code points not assigned to a special datum map to a finite datum. Hence, the code points for special datums can share the same exponent-field values with finite datums. Specifically, P3109 does not reserve the highest exponent field entirely for representing infinities and `NaNs`. In Table I, the code points 0110 and 0111 share the same exponent-field, 11. In the `Binary4p2se` format, these code points map to the datum 2 and $+\infty$, respectively. In contrast, IEEE-754 uses code points with the highest exponent field to represent $+\infty$ and `NaNs`, respectively.

These choices in P3109's encoding of special datums maximize the number of finite datums in low-precision formats. Each of `Binary4p2se` and `Binary4p2sf` represents 16 distinct datums. For comparison, the IEEE-754 encoding assigns two code points to zeros and two additional code points to `NaN`, resulting in 14 unique datums.

For a normal code point, P3109 stores $E_{biased} = E + B$ and recovers the canonical exponent E by subtracting the bias B. However, P3109's bias computation deviates from IEEE-754: IEEE-754 defines the exponent bias from the maximum representable exponent `emax`, whereas P3109 instead derives B directly from bitwidth, precision, and signedness. For `Signed` formats, $B = 2^{K-P-1}$; for `Unsigned` formats, $B = 2^{K-P}$. The exponent field width W is $K - P$ for `Signed` formats and $K - P + 1$ otherwise. Thus, in P3109, `emax` is derived from the bias. The specific value of `emax` depends on which code points are reserved for ∞ and `NaN`.

Consider the code point $n = 0011_2$ for the formats `Binary4p2se` and `Binary4p2sf` from the example in Table I. Its sign bit is 0 and the exponent field is 01_2, so $E_{biased} = 1$, and the trailing significand is $T = 1$. Since the bias $B = 2^{K-P-1} = 2^{4-2-1} = 2$, the exponent field is decoded to the canonical exponent, $E = E_{biased} - B = 1 - 2 = -1$. Finally, the floating-point datum encoded by the code point is

$$(1 + T2^{1-P})2^E = (1 + 2^{-1})2^{-1} = \tfrac{3}{4}$$

For a subnormal code point, the exponent field of the P3109 code point is all zeros, as in IEEE-754. The unbiased exponent for subnormals is $1 - B$.

For a `Signed` format, P3109 has an ordering pattern similar to IEEE-754: code points below 2^{K-1} increase from zero toward the positive boundary, while code points above it run in reverse numerical order from datums near zero toward the negative boundary; 2^{K-1} encodes `NaN`. Numerical comparison can therefore use similar sign-aware integer techniques, with special handling for `NaN`.

For arithmetic operations, P3109 provides the five IEEE-754 rounding directions RNE, RNA, RD, RU, and RZ, together with round-to-odd (RTO) and three finite-random-bit stochastic variants (SR) [17]. Ideal stochastic rounding chooses adjacent datums with distance-proportional probabilities and is point-wise unbiased. With a uniform N-bit random integer, the three finite-bit variants quantize the ideal round-up probability in different ways. Their bias properties also depend on the distribution and precision of the inputs; Section III-C states the probability spaces used in our proofs.

P3109 formats can also exhibit unique rounding behavior at very low precision. For $P = 1$, nonzero finite datums have no trailing significand bit, so parity-based modes use the parity of the stored biased exponent $E_{biased} = E + B$ instead. P3109 additionally defines three saturation modes for rounded results outside $[M^{lo}, M^{hi}]$. Every mode propagates `NaN` unchanged. `SatFinite` clamps every such result to a finite boundary. `SatPropagate` preserves representable infinities and otherwise clamps. `SatNone` returns a finite boundary, infinity, or `NaN` according to rounding direction, signedness, and domain.

III. The FLoPS Framework

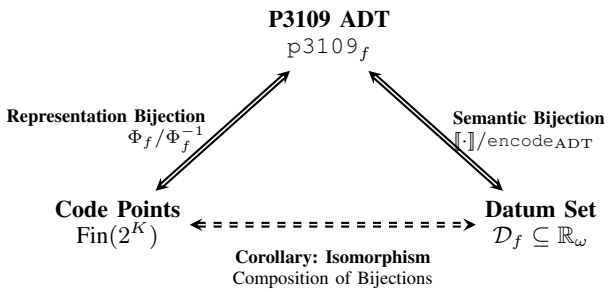


Fig. 1: The three-way isomorphism between code points, the algebraic ADT, and the datum set.

We describe our FLoPS framework for the formalization of low-precision representations in the P3109 standard. To bridge the gap between low-level code points used in the P3109 standard and high-level mathematical reasoning, FLoPS has three primary parts: (1) code points in $\mathrm{Fin}(2^K)$, (2) an algebraic model with inductive types, and (3) datum sets $\mathcal{D}_f \subseteq \mathbb{R}_\omega$. To reason about the projection pipeline, we use Mathlib's EReal, which provides the real-and-infinity component of $\mathbb{R}_\omega$ together with a library of order and arithmetic theorems; we represent NaN separately as described below. To facilitate rigorous proofs, we define the P3109 formats and algebraic terms as inductive types. This layer abstracts away from bit-level details to enforce the standard's parametric constraints on bitwidth, precision, and encodings of special datums directly in the type system. By treating P3109 representations as algebraic structures rather than as bit patterns, we simplify the formalization of operations and predicates.

For a fixed format f, we represent its code points by $\mathrm{Fin}(2^K)$ and factor decoding through the algebraic type:

$$\mathrm{Fin}(2^K) \underset{\Phi_f^{-1}}{\overset{\Phi_f}{\longleftrightarrow}} \texttt{p3109}_f \xleftarrow[\mathrm{encode}_{\mathrm{ADT}}]{[\![\cdot]\!]} \mathcal{D}_f.$$

Here, Φ_f maps a code point to an ADT term, and $[\![\cdot]\!]$ evaluates that term as a datum; the lower arrows are their inverses. We use the neutral name Φ_f because its codomain is our intermediate ADT rather than the datum set. Both maps are proved bijective in Section III-B.

Three-way isomorphism. To relate our model directly to the P3109 specification, we establish a three-way formal equivalence, as shown in Figure 1. Specifically, it involves showing (1) *representation equivalence* and (2) *semantic equivalence*. The representation equivalence step ensures a verified bijection between the code points and our algebraic model. The semantic equivalence step also provides a proof of bijection between the algebraic model and the datum set of a given P3109 format. These two steps guarantee that any property proven within our algebraic model automatically holds for the code point representation specified by the standard.

FLoPS also includes a formalization of the core projection pipeline from $\mathbb{R}_\omega$ to an algebraic term, which determines the target code point through Φ_f^{-1}. The pipeline uses a bespoke implementation of the rounding modes, including the unique "ties-to-even" logic required for $P = 1$ formats, and formalizes the saturation semantics. It supports verification of the foundational FP algorithms in Section IV.

A. The P3109 Algebraic Model

The P3109 algebraic type is indexed by a P3109 format as detailed in Section II. The core constructor of the ADT is a pair of integers m (significand) and e (exponent), supplemented with the constraints imposed by the indexed format. Note that because e is not explicitly scaled with respect to the precision, e represents the exponent of the significand's *least significant bit*.

Format parameters and derived values. We define a format f using a structure containing the four fields K, P, s (*i.e.*, Σ) and d (*i.e.*, Δ) along with their constraints.

```
structure p3109_format where
  -- Four format-defining parameters
  K : ℕ
  P : ℕ
  s : Signedness
  d : Domain
  -- Validity proofs for the parameter tuple
  h_K : 2 < K
  h_P : 0 < P ∧ (s = Signedness.signed → P < K
      ) ∧ (s = Signedness.unsigned → P ≤ K)
```

Listing 1: Definition of the p3109 format structure

To characterize finite terms within a format's dynamic range, we derive minimum and maximum representable exponents from the format's bias B. We distinguish the exponent e_{msb} assigned to a P-bit significand's most significant bit (MSB) from the exponent e_{lsb} assigned to its least significant bit (LSB). They satisfy $e_{\mathrm{msb}} = e_{\mathrm{lsb}} + P - 1$, equivalently $m2^{e_{\mathrm{lsb}}} = m2^{e_{\mathrm{msb}}-P+1}$. The standard's canonical exponent E uses the MSB convention, whereas the exponent e in p3109_finite uses the LSB convention. Accordingly, the **minimum exponent** for the MSB is uniformly defined as $\mathtt{emin} = 1 - B$, and $\mathtt{emin}_{\mathrm{lsb}} = \mathtt{emin} - P + 1$.

$$\mathtt{emax} = \begin{cases} 2^W - 3 - B & \text{if } P = 1, s = \mathtt{unsigned}, d = \mathtt{extended} \\ 2^W - 2 - B & \text{if } P = 1, s = \mathtt{unsigned}, d = \mathtt{finite} \\ 2^W - 2 - B & \text{if } P = 1, s = \mathtt{signed}, d = \mathtt{extended} \\ 2^W - 2 - B & \text{if } P = 2, s = \mathtt{unsigned}, d = \mathtt{extended} \\ 2^W - 1 - B & \text{otherwise} \end{cases}$$

Fig. 2: The definition of emax for all K, P, s, and d.

Unlike emin, the definition of **maximum exponent** varies across formats due to differences in the code points for special datums (see Table II). The maximum possible exponent for the MSB (emax) is defined via case analysis on the format parameters and their associated exponent field length W. The default definition $\mathtt{emax} = 2^W - 1 - B$, for which the encoding is $2^W - 1$, covers the majority of the formats. However, lower precision formats deviate from this definition. When $P = 1$, the trailing bits of the binary encoding form the exponent field (*i.e.*, no significand field). Moreover, the largest binary encodings are reserved for special datums. For an Unsigned Extended format with $P = 1$, we have $W = K$; NaN and ∞ occupy $2^W - 1$ and $2^W - 2$, respectively. The largest usable stored exponent-field value E_{biased} in this case is $2^W - 3$, so decoding gives $\mathtt{emax} = E_{biased} - B = 2^W - 3 - B$. A similar case arises for the Unsigned Extended formats when $P = 2$. Figure 2 provides the full definition of emax covering all cases. Given the definition for emax, we define the maximum exponent for the LSB as $\mathtt{emax}_{\mathrm{lsb}} = \mathtt{emax} - P + 1$.

Validity and canonical terms. We define algebraic terms with the inductive type p3109, indexed by a format f.

```
inductive p3109 (f : p3109_format) where
  -- Infinity validity proofs
  | p3109_infinity (h : f.d = .extended) (sign
      : Bool) (hm : sign=true → f.s=.signed)
  | p3109_nan
  -- Finite validity proofs
  | p3109_finite (m e : ℤ)
    (hm : f.s = .unsigned → 0 ≤ m)
    (hcan : @canonical_p3109 f ⟨m, e⟩)
```

Listing 2: Definition of the `p3109` inductive type

The `p3109` type has constructors `p3109_infinity`, `p3109_nan`, and `p3109_finite`. The infinity constructor requires f to be `Extended`; negative infinity additionally requires f to be `Signed`. In displayed constructor applications, including the worked example and Figure 4, we suppress the domain, signedness, and canonicality proof arguments; Lean constructs and checks them from the surrounding conditions.

A finite term $\texttt{p3109\_finite}(m, e)$ must satisfy a separate signedness constraint: m is nonnegative when f is `Unsigned`. Its canonicality predicate has two branches. The *normal* branch requires $2^{P-1} \le |m| < 2^P$ and $\texttt{emin}_{\text{lsb}} \le e \le \texttt{emax}_{\text{lsb}}$, together with a format-specific restriction at the top exponent that prevents collision with code points reserved for special datums. In our ADT's canonicality predicate, the *subnormal branch* requires $e = \texttt{emin}_{\text{lsb}}$ and $|m| < 2^{P-1}$. This branch deliberately includes $m = 0$, giving zero the unique ADT representative $(0, \texttt{emin}_{\text{lsb}})$. The P3109 standard classifies zero separately from normal and subnormal datums and gives it the canonical exponent $E = 0$; representing zero in this branch of our ADT changes neither the evaluated datum nor code point 0.

Lastly, we define a mapping $[\![\cdot]\!]$ from the P3109 algebraic model to $\mathbb{R}_\omega$. Mathlib's `EReal` represents $\mathbb{R} \cup \{-\infty, +\infty\}$, but does not contain `NaN`. We therefore represent $\mathbb{R}_\omega$ as $\texttt{EReal} \oplus \texttt{Unit}$. The left injection contains every real as well as Mathlib's $\top$ and $\bot$, representing $+\infty$ and $-\infty$; the unique right-injected element represents `NaN`. A finite P3109 term (m, e) maps to the left-injected $m \times 2^e$; the infinity constructors map to the left-injected $\top$ and $\bot$, and `p3109_nan` maps to the right-injected unit. We formally define the datum set $\mathcal{D}_f$ of a format f as the range of the evaluation function: $\mathcal{D}_f = \{v \mid \exists a \in \texttt{p3109}_f, [\![a]\!] = v\}$. Returning to the `Binary4p2se` example from Section II, the constructor stores the integer significand $m = 3$ and LSB exponent $e = E - P + 1 = -2$:

$$\Phi_f(3) = \texttt{p3109\_finite}(3, -2),$$
$$[\![\Phi_f(3)]\!] = 3 \cdot 2^{-2} = \tfrac{3}{4}.$$

We establish the invertibility of these mappings in the next section, in which we prove the correctness of the formalization.

B. Establishing the Three-Way Isomorphism

We now establish the isomorphism sketched in Figure 1.

Validating algebraic bounds. To establish the full isomorphism, we must first verify that the algebraic bounds defining our ADT are mathematically sound. The most delicate bound is the maximum exponent. As discussed in Section III-A, formats with precision $P \le 2$ reserve the highest exponent field encodings for special datums, dynamically lowering the maximum representable finite exponent. We decode the exponent directly from the code point n_{fmax} of the maximum finite datum (Figure 3) and prove its equality with the `emax` defined in Figure 2 under case analysis on the format parameters.

$$n_{fmax} = \begin{cases} 2^{K-1} - 2 & \text{if } \Sigma = \texttt{Signed} \wedge \Delta = \texttt{Extended} \\ 2^{K-1} - 1 & \text{if } \Sigma = \texttt{Signed} \wedge \Delta = \texttt{Finite} \\ 2^{K} - 3 & \text{if } \Sigma = \texttt{Unsigned} \wedge \Delta = \texttt{Extended} \\ 2^{K} - 2 & \text{if } \Sigma = \texttt{Unsigned} \wedge \Delta = \texttt{Finite} \end{cases}$$

Fig. 3: The definition of n_{fmax}, the code point of the maximum finite datum in a given format.

Bijection between code points and the ADT. We now make the code-point-to-ADT correspondence explicit. Fix a format f and write $\Phi = \Phi_f$ for the map introduced above. Figure 4a gives its piecewise definition. For code points in the finite domain, Φ extracts the trailing significand $T = n \bmod 2^{P-1}$ and the stored exponent field $E_{biased} = \lfloor n/2^{P-1} \rfloor$ by Euclidean division. For a normal code point, encoding stores $E_{biased} = E + B$, where E is the canonical exponent; decoding subtracts B to recover $E = E_{biased} - B$. The algebraic constructor then stores the LSB exponent $e = E - P + 1$. The all-zero exponent field is special: it denotes zero or a subnormal rather than a normal datum with exponent $-B$. A nonzero subnormal has canonical exponent $E = 1 - B = \texttt{emin}$, and the constructor stores $e = \texttt{emin}_{\text{lsb}}$. The helper function Φ_{fin} (Figure 4b) implements these two cases.

As an executable cross-check, we transcribed the standard's complete $K = 4$ tables into exact dyadic form: 14 formats and 224 code points. A standalone C decoder implements the bit-field equations independently of Lean. The artifact's table-comparison harness exhaustively compares both the C output and the executable `FLoPS fromBits` decoder with those tables.

We prove that Φ is a bijection.

Lemma 1 (Φ is injective)**.** $\forall n_1, n_2 \in \mathit{Fin}(2^K), \quad \Phi(n_1) = \Phi(n_2) \implies n_1 = n_2$

Lemma 2 (Φ is surjective)**.** $\forall b : \texttt{p3109}_f, \quad \exists a \in \mathit{Fin}(2^K) \quad \textit{s.t.} \quad \Phi(a) = b$

A challenge with these proofs, which distinguishes it from the corresponding IEEE-754 counterparts, is that P3109 dynamically shifts the boundary between code points for finite and special datums based on all four format parameters (K, P, Σ, Δ), whereas IEEE-754 reserves a fixed all-ones exponent field for infinities and `NaNs`. For injectivity, we decompose the encoding space into disjoint regions and use the uniqueness of Euclidean division within the finite region. For surjectivity, we construct explicit bit-pattern witnesses for each ADT constructor and verify they do not collide with adjacent

$$\Phi(n) = \begin{cases} \texttt{NaN} & \text{if } (n = 2^{K-1} \land \Sigma = \mathrm{S}) \lor (n = 2^K - 1 \land \Sigma = \mathrm{U}) \\ +\infty & \text{else if } \Delta = \mathrm{E} \land ((\Sigma = \mathrm{S} \land n = 2^{K-1} - 1) \lor (\Sigma = \mathrm{U} \land n = 2^K - 2)) \\ -\infty & \text{else if } n = 2^K - 1 \land \Sigma = \mathrm{S} \land \Delta = \mathrm{E} \\ -\Phi_{fin}(n - 2^{K-1}) & \text{else if } 2^{K-1} < n < 2^K \land \Sigma = \mathrm{S} \\ \Phi_{fin}(n) & \text{otherwise.} \end{cases}$$

(a) The ordered code-point-to-ADT decoding function.

$$\Phi_{fin}(n) = \begin{cases} \texttt{p3109\_finite}(T, \texttt{emin}_{\mathrm{lsb}}) & \text{if } E_{biased} = 0, \\ \texttt{p3109\_finite}(2^{P-1} + T, E_{biased} - B - P + 1) & \text{otherwise.} \end{cases}$$

(b) The helper for finite code points.

Fig. 4: The cases in (a) are cascading conditionals evaluated from top to bottom; each case after the first assumes that all preceding conditions are false. S, U, and E abbreviate `Signed`, `Unsigned`, and `Extended`. Constructor proof arguments are omitted; Lean constructs them from the branch conditions.

partitions. Because P3109 shifts partition boundaries dynamically, each format configuration requires separate verification; our mechanization certifies that the ADT invariants correctly partition the full encoding space.

Semantic soundness. To connect the algebraic model to the semantic domain, we define an ADT reification function, named `encode` in Lean, $\texttt{encode}_{\mathrm{ADT}} : \mathcal{D}_f \to \texttt{p3109}_f$, and establish the round-trip identity:

Theorem 1 (Round-Trip Identity)**.**

$$\forall a \in \mathit{p3109}_f, \quad \mathit{encode}_{\mathrm{ADT}}(\llbracket a \rrbracket) = a.$$

Theorem 1 confirms that $\llbracket \cdot \rrbracket$ is injective and $\texttt{encode}_{\mathrm{ADT}}$ is its left inverse. Since $\llbracket \cdot \rrbracket$ is surjective onto $\mathcal{D}_f$ by construction, it is a bijection. Composing Φ_f and $\llbracket \cdot \rrbracket$ yields the full isomorphism between $\mathrm{Fin}(2^K)$ and $\mathcal{D}_f$. In particular, define the datum-to-code-point map

$$\omega\mathrm{Encode}_f := \Phi_f^{-1} \circ \texttt{encode}_{\mathrm{ADT}} : \mathcal{D}_f \to \mathrm{Fin}(2^K).$$

The subscript makes the target format explicit, while $\texttt{encode}_{\mathrm{ADT}}$ names only the intermediate reification from a datum to its ADT term.

C. Projection

P3109 operations decode code point operands to $\mathbb{R}_\omega$, apply a mathematical operation, and project its result to a code point in the target format. Projection conceptually has three steps: *rounding* to precision P with an unbounded exponent, *saturating* with respect to the format's finite range, and *encoding* the resulting datum. In our algebraic layer, the last step reifies the datum as a `p3109` term; applying Φ_f^{-1} gives the standard's code point. The standard defines a projection specification π as a pair consisting of a rounding mode and a saturation mode: $\pi = (Rnd_\pi, Sat_\pi)$.

Rounding. The `RoundToPrecision` function converts $x \in \mathbb{R}_\omega$ to a rounded result whose finite cases have precision P and an unbounded exponent. P3109 includes RNE, RNA, RD, RU, RZ, RTO, and three variants of SR. Our implementation is parameterized by a rounding operator that rounds the scaled significand of x to an integer; for each operator, we provide an axiomatic definition and prove the correctness of the implementation. For the standard IEEE-754 modes, we formalize their defining properties (*e.g.*, RNE returns the nearest neighbor and even significand when at midpoint, RD returns the predecessor, RU returns the successor) and show they lift to the P3109 projection pipeline. For the three SR variants, we formalize the bias properties studied by Fitzgibbon and Felix [17]. Write the scaled significand as $y = m + \eta$, where $m = \lfloor y \rfloor$ and $0 \le \eta < 1$. With N random bits, R is uniform on $\{0, \ldots, 2^N - 1\}$, and rounding chooses either m or $m + 1$. Rounding down has signed error $-\eta$ output ULPs and rounding up has error $1 - \eta$. If $p_\uparrow(\eta)$ is the probability of rounding up, the *pointwise bias* is $\mathbb{E}_R[\mathrm{rnd}(y; R) - y] = p_\uparrow(\eta) - \eta$; the *aggregate bias* averages it over a specified distribution of input fractions η.

For idealized inputs, η is uniform on $[0, 1)$. Variant A rounds up with probability $\lfloor \eta 2^N \rfloor / 2^N$, yielding aggregate bias $-2^{-(N+1)}$. Variant B shifts the random threshold by half a step and has zero aggregate bias under this continuous distribution. Finite-precision inputs give a different distribution: if the source has D more significand bits than the target, the possible fractions form the grid $\eta = j/2^D$, $0 \le j < 2^D$, on which variant B can be biased. At an aligned position $\eta = k/2^N$, however, exactly k of its 2^N random choices round up, so its pointwise bias is zero.

Variant C first rounds η to the nearest point on the 2^{-N} grid, $\eta' = \mathrm{RNITE}(\eta 2^N)/2^N$, where RNITE rounds to the nearest integer with ties to even, and then applies variant A. For $N > 0$, the initial nearest-even rounding errors average to zero over a complete D-bit input grid, while variant A is unbiased at the resulting grid points; hence variant C has zero aggregate bias on that grid. We formalize a root-mean-square accumulated-error bound of $\sqrt{n}/2$ output ULPs for n independent aligned variant-B roundings. For variant C, we formalize the accumulated error relative to the original inputs and prove the same bound for the $n = 2^D$ points of a complete D-bit input grid.

Flocq [6], a library that formalizes floating-point arithmetic in Rocq, packages monotonicity and exactness on integers in

its `Valid_rnd` typeclass: $x \le y$ implies $\texttt{rnd}(x) \le \texttt{rnd}(y)$, and $\texttt{rnd}(n) = n$ for every integer n. `FLoPS` instead uses a `Faithful` typeclass, requiring $\texttt{rnd}(e, r)$ to be either $\lfloor r \rfloor$ or $\lceil r \rceil$, and a separate `Monotonic` typeclass. For each fixed exponent, `Faithful` and `Monotonic` together imply the Flocq-style obligations. All P3109 rounding modes are faithful, but the stochastic operators are not monotonic. Assuming only faithfulness, we can still establish a weaker monotonicity result when one side of the inequality is already representable (*i.e.*, $x \le y$) ; this is sufficient for the arithmetic proofs in the following sections.

To accommodate P3109's low-precision formats, our rounding operator has signature `rnd`: $\mathbb{Z} \to \mathbb{R} \to \mathbb{Z}$, where the first argument is the canonical exponent. This diverges from Flocq's [6] context-insensitive $\mathbb{R} \to \mathbb{Z}$ operator, which cannot implement P3109's exponent-dependent tie-breaking for $P = 1$ formats (*e.g.*, RNE relies on the parity of the stored biased exponent $E_{biased} = E + B$ rather than significand parity). We avoid partially applying the exponent, as doing so would make the operator binade-dependent and prevent the proof of global monotonicity. Here a binade is the interval between two consecutive powers of two; the concern is that the partially applied operator would change from one such interval to the next.

Saturation. Using the saturation modes described in Section II, the `Saturation` function handles inputs outside the format's finite range $[M^{lo}, M^{hi}]$. The output of applying `RoundToPrecision` and `Saturation` must be in the target datum set. We prove this through case analysis on whether the original input is already in the finite range. If it is, the result of `RoundToPrecision` and `Saturation` is guaranteed to be in the datum set due to the aforementioned monotonicity property. For inputs greater than M^{hi}, the saturated result is M^{hi}, $+\infty$, or `NaN`; below M^{lo}, it is M^{lo}, $-\infty$, or `NaN`, as prescribed by the standard's saturation table. A `NaN` input is propagated unchanged by every saturation mode.

ADT reification. The Lean function $\texttt{encode}_{\text{ADT}}$ takes a datum $x \in \mathcal{D}_f$ and returns the corresponding `p3109` algebraic term; its correctness is guaranteed by the bijection established in the three-way isomorphism (Theorem 1).

Projection. We assemble the three steps to form the `Projection` pipeline. We present key properties of `Projection` that are relevant to our analysis in Sections IV and V.

Lemma 3 (Projection equals rounding in range)**.** *If a real r is in the finite range $M^{lo} \le r \le M^{hi}$ then* `Projection` *and* `RoundToPrecision` *produce the same outputs in $\mathbb{R}_\omega$.*

Lemma 4 (Projection is "faithful")**.** *Projecting a real r in the finite range $[M^{lo}, M^{hi}]$ under any (Rnd_π, Sat_π) produces r if $r \in \mathcal{D}_f$, and $pred(r)$ or $succ(r)$ otherwise.*

Both lemmas follow because saturation is not triggered for inputs in the finite range and rounding selects either a representable input or one of its two representable neighbors.

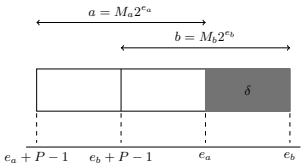
Interaction of round-to-odd with saturation. A key property of RTO is the "even implies exact" diagnostic: an even significand in the result certifies that no rounding occurred, which is vital for double-rounding avoidance [29]. We formalized the conditions under which this holds end-to-end through P3109's round-then-saturate pipeline. For values within the representable finite range $M^{lo} \le x \le M^{hi}$, saturation is not triggered (Lemma 3), and the diagnostic holds universally. However, when overflow triggers saturation, the diagnostic's reliability depends on the parity of the boundary values M^{hi} and M^{lo}. In configurations where these boundaries have odd significands (e.g., `Signed Finite` and $P > 1$), the diagnostic remains sound. Conversely, for formats like `Unsigned Finite` and `Signed Extended` ($P > 1$), M^{hi} has an even significand because the odd encoding is reserved for special values. Saturated out-of-range inputs in these formats can yield an even significand despite being inexact. Our mechanized proofs provide a complete characterization: the diagnostic is sound for all in-range values across all formats and for `Signed Finite` formats under any saturation. In other configurations, saturation can clip to a boundary value with an even significand.

IV. Analysis of FastTwoSum for P3109

`FastTwoSum` is a foundational error-free transformation (EFT) that computes the exact rounding error of floating-point addition. Introduced by Dekker [16], the algorithm computes a floating-point sum $s = \circ(a+b)$ and a correction term t. Under any round-to-nearest (RN) mode, if $e_a \ge e_b$, it is guaranteed that $s+t = a+b$ exactly. This property makes it ubiquitous in algorithms that emulate twice the working precision or require intermediate error compensation [30], [19].

FastTwoSum(a, b) :
$s = \circ(a + b)$
$z = \circ(s - a)$
$t = \circ(b - z)$
return s, t

(a) Original `FastTwoSum` algorithm



(b) Visualization of $\delta = a + b - s$

Fig. 5: (a) The expression $\circ(\cdot)$ denotes a floating-point operation that rounds the real arithmetic result via a given rounding mode $\circ$ (*e.g.*, RNE). The outputs s and t represent, respectively, the rounded sum of $a + b$ and an approximation of its rounding error $\delta = a + b - s$. (b) Suppose $a = M_a 2^{e_a}$ and $b = M_b 2^{e_b}$, where $e_a \ge e_b$. A P-bit significand for x spans exponent positions e_x through $e_x + P - 1$. The diagram shows the overlapping case $e_a \le e_b + P - 1$; $s = \circ(a + b)$ rounds away the trailing bits of b. The bits in the gray region form the rounding error $\delta = a + b - s$. If this region is exactly representable with P bits, δ is a datum in the same format as a and b.

Given the widespread use of FastTwoSum, we developed a mechanized formalization for P3109 to account for its extremely low precision ($P = 1$) and explicit saturation semantics, which prior IEEE-754 analyses exclude [5], [23], [6]. Using projections, we define FastTwoSum for finite P3109 inputs a and b as follows.

Definition 1 (FastTwoSum)**.** *Given two inputs $a, b \in \mathcal{D}_f \setminus \{\pm\infty, NaN\}$ such that f is Signed, FastTwoSum computes two outputs $s, t \in \mathcal{D}_f$ using three operations. Each operation is associated with projection specifications (Rnd_1, Sat_1), (Rnd_2, Sat_2), and (Rnd_3, Sat_3), respectively.*

$$s \leftarrow \text{Project}_{(Rnd_1, Sat_1)}(a + b)$$
$$z \leftarrow \text{Project}_{(Rnd_2, Sat_2)}(s - a)$$
$$t \leftarrow \text{Project}_{(Rnd_3, Sat_3)}(b - z)$$

The primary objective of FastTwoSum is to ensure that t is an accurate estimate of the error $\delta = a + b - s$. To analyze the accuracy of t under P3109 arithmetic, we first establish the conditions under which the second operation is exact (*i.e.*, no rounding error).

Lemma 5 (Exact Intermediate Step)**.** *Let $a, b \in \mathcal{D}_f \setminus \{\pm\infty, NaN\}$ for a Signed format f. If $|a + b| \leq M^{hi}$ and $e_a \geq e_b$, then $s - a \in \mathcal{D}_f$.*

The proof for the above lemma can be found in [5, Lemma 2.5]. Our formalization of this proof certifies that Lemma 5 holds for all Signed P3109 formats, irrespective of the available precision P and projection specifications (Rnd_1, Sat_1) and (Rnd_2, Sat_2). Because $s - a$ is representable, Lemma 5 yields $z = \text{Project}_{(Rnd_2, Sat_2)}(s - a) = s - a$. Subsequently, $t = \text{Project}_{(Rnd_3, Sat_3)}(b - (s - a)) = \text{Project}_{(Rnd_3, Sat_3)}(a + b - s)$: t is the projection of $\delta = a + b - s$ via Rnd_3 and Sat_3. As previously mentioned, if s is a representable datum *nearest* to $a + b$, then δ is also representable. We establish this property for all Signed P3109 formats.

Lemma 6 (Representable Error)**.** *Let $a, b \in \mathcal{D}_f \setminus \{\pm\infty, NaN\}$ for a Signed format f. If $|a + b| \leq M^{hi}$ and $s = \text{Project}_{(Rnd_1, Sat_1)}(a+b)$ for $Rnd_1 \in \text{RN}$, then $\delta = a+b-s \in \mathcal{D}_f$.*

Note that Lemma 6 only requires Rnd_1 to perform round-to-nearest (*i.e.*, $Rnd_1 \in \text{RN}$) without any specified tie-breaking rule. Because Rnd_1 is not restricted to RNE, we are able to prove Lemma 6 for all Signed formats without separately addressing cases where $P = 1$. Since Lemma 5 ensures t is a projection of δ and δ is representable, the final projection is exact.

Theorem 2 (Exact Error Computation)**.** *Let $a, b \in \mathcal{D}_f \setminus \{\pm\infty, NaN\}$ for a Signed format f. If $|a + b| \leq M^{hi}$, $e_a \geq e_b$, and $Rnd_1 \in \text{RN}$, then $t = a + b - s$.*

Theorem 2 implies that under RN, $|a+b| \leq M^{hi}$ and $e_a \geq e_b$ guarantee $s+t = a+b$ and that FastTwoSum is an EFT in P3109 arithmetic. We extend this analysis to the general case where the rounding mode is not RN, a scenario relevant to P3109's support for diverse rounding modes. Previous analyses have established that even when s is *not* the nearest neighbor of $a + b$, t is still a faithful rounding of δ so long as $z = s - a$ [5] [23]. As previously explained, Lemma 5 induces $z = s - a$ and $t = \text{Project}_{(Rnd_3, Sat_3)}(\delta)$. Through Lemma 4, which guarantees the faithfulness of projections, we establish that t is a faithful rounding of δ for all Signed P3109 formats irrespective of the rounding mode used to compute s.

Theorem 3 (Faithful Error Computation)**.** *Let $a, b \in \mathcal{D}_f \setminus \{\pm\infty, NaN\}$ for a Signed format f. If $|a + b| \leq M^{hi}$ and $e_a \geq e_b$, then t is a faithful rounding of $\delta = a + b - s$.*

To derive $t = \text{Project}_{(Rnd_3, Sat_3)}(b - z) = \text{Project}_{(Rnd_3, Sat_3)}(a + b - s)$ from Lemma 5, Theorems 2 and 3 must ensure that z is a finite datum (*i.e.*, $|z| \leq M^{hi}$). Note that Theorems 2 and 3 both require $|a + b| \leq M^{hi}$, which subsequently ensures $|s| \leq M^{hi}$ (*i.e.*, the first operation does not overflow to infinity). Therefore, under the standing assumption $e_a \geq e_b$, we must establish that $|s| \leq M^{hi}$ guarantees $|s - a| \leq M^{hi}$ and the second operation $z \leftarrow \text{Project}_{(Rnd_2, Sat_2)}(s - a)$ does not induce overflow. Boldo *et al.* prove this property in [5, Theorem 5.1]. However, the proof requires the target format to have at least 3 bits of precision. Through new proofs that separately address $P = 1$ and $P = 2$, we confirm that FastTwoSum is immune to any intermediate overflow for all Signed formats regardless of their precision.

Theorem 4 (Overflow Immunity)**.** *Let $a, b \in \mathcal{D}_f \setminus \{\pm\infty, NaN\}$ for a Signed format f. If $|a + b| \leq M^{hi}$ and $e_a \geq e_b$, then $|s - a| \leq M^{hi}$, so the projection computing z does not overflow.*

The properties $s - a \in \mathcal{D}_f$ (Lemma 5) and $\delta \in \mathcal{D}_f$ (Lemma 6) that underlie FastTwoSum's EFT guarantees both require s to be finite. Theorem 2 thus assumes $|a+b| \leq M^{hi}$ to ensure the first operation does not overflow to infinity regardless of Rnd_1 and Sat_1. Due to its explicit saturation semantics, addition in P3109 can keep s finite even when the real sum lies outside the finite range. Based on this observation, we identify a novel property of FastTwoSum: when an overflowing sum is clamped to the corresponding finite boundary, $e_a \geq e_b$ guarantees $t = a + b - s$.

Theorem 5 (Exact Overflow Error Computation)**.** *Let $a, b \in \mathcal{D}_f \setminus \{\pm\infty, NaN\}$ for a Signed format f. If $e_a \geq e_b$ and either $a + b > M^{hi}$ with $s = M^{hi}$ or $a + b < M^{lo}$ with $s = M^{lo}$, then $t = a + b - s$.*

Theorem 5 implies that when the real result $a + b$ is in the overflow range and s is clamped to the corresponding finite boundary, FastTwoSum can compute the *exact overflow error*. Furthermore, Theorem 5 holds irrespective of the rounding mode used to compute s. By addressing sums in the overflow range, Theorem 5 broadens FastTwoSum's applicable domain. Our findings thus highlight the utility of P3109's saturation features, as they preserve FastTwoSum's EFT guarantees in settings excluded by pure rounding.

V. Analysis of ExtractScalar

`FastTwoSum` is the foundation of various floating-point splitting algorithms, which decompose an input x across two numbers x_h and x_ℓ such that $x = x_h + x_\ell$. The most prominent example is Dekker's splitting [16], which splits a P-precision number across two values each with at most $\lfloor \frac{P}{2} \rfloor$ effective bits of precision (*i.e.*, available precision minus the number of trailing zeroes). When applied iteratively, splitting algorithms enable floating-point expansions, which represent numbers across multiple non-overlapping, lower-precision values. Given the low-precision nature of P3109 formats, formalizing floating-point splitting techniques for P3109 representations is of practical use to the design of future algorithms.

Unlike Dekker's splitting, which assigns a fixed number of bits, `ExtractScalar` [31] is an algorithm for which the distribution of precision is configurable. Using the projection specification, we define `ExtractScalar` for finite P3109 inputs as follows.

Definition 2 (`ExtractScalar`)**.** *Given two finite inputs $\sigma, x \in \mathcal{D}_f$ such that f is `Signed`, `ExtractScalar` computes two outputs $x_h, x_\ell \in \mathcal{D}_f$ using three operations performed under* RNE *and arbitrary saturation modes Sat_1, Sat_2, and Sat_3, which may differ.*

$$
\begin{aligned}
s &\leftarrow \mathrm{Project}_{(\mathrm{RNE}, Sat_1)}(\sigma + x) \\
x_h &\leftarrow \mathrm{Project}_{(\mathrm{RNE}, Sat_2)}(s - \sigma) \\
x_\ell &\leftarrow \mathrm{Project}_{(\mathrm{RNE}, Sat_3)}(x - x_h)
\end{aligned}
$$

`ExtractScalar` assumes $\sigma = 2^i$ for some $i \in \mathbb{Z}$ (*i.e.*, σ is an integer power of 2). `ExtractScalar` also assumes $|x| \leq 2^{-j}\sigma$ for some $j \in \mathbb{N}$ such that $|\sigma + x| \leq M^{hi}$ (*i.e.*, no overflow). Given these assumptions, Rump *et al.* prove that `ExtractScalar`'s outputs x_h and x_ℓ satisfy $x = x_h + x_\ell$: `ExtractScalar` is an EFT of x [31]. Moreover, they establish that the number of effective bits in x_h and x_ℓ are determined by the working precision P and the parameter j used to define σ (see Figure 6a).

While producing an EFT, `ExtractScalar` enforces several bounds on x_h and x_ℓ: $|x_h| \leq 2^{-j}\sigma$, $x_h \in 2^{-P}\sigma\mathbb{Z}$ (*i.e.*, x_h is an integer multiple of $2^{-P}\sigma$ for precision P), and $|x_\ell| \leq 2^{-P}\sigma$. Through $|x_h| \leq 2^{-j}\sigma$ and $x_h \in 2^{-P}\sigma\mathbb{Z}$, `ExtractScalar` ensures that the number of effective bits in x_h is determined by the *exponent difference* between σ and x. `ExtractScalar` thus allows σ to be configured with the appropriate j to achieve the desired distribution across x_h and x_ℓ. In conjunction with the property $|x_\ell| \leq 2^{-P}\sigma$, $x_h \in 2^{-P}\sigma\mathbb{Z}$ also ensures that x_h and x_ℓ are non-overlapping.

`ExtractScalar`'s guarantees of $|x_h| \leq 2^{-j}\sigma$ and $x_h \in 2^{-P}\sigma\mathbb{Z}$ also enable EFTs for floating-point vector operations. Given an n-element floating-point vector $v \in \mathcal{D}_f^n$, suppose σ is configured with respect to $\max_i |v_i|$ (*i.e.*, $\max_i |v_i| \leq 2^{-j}\sigma$). Applying `ExtractScalar` to each element v_i with the given σ subsequently produces vectors $v_h, v_\ell \in \mathcal{D}_f^n$ such that $v = v_h + v_\ell$. Furthermore, `ExtractScalar` ensures that for all elements $v_{h,i}$, $|v_{h,i}| \leq 2^{-j}\sigma$ and $v_{h,i} \in 2^{-P}\sigma\mathbb{Z}$. With their effective bits occupying the same range (see Figure 6b), the elements of v_h can be interpreted as fixed-point numbers. In [31], Rump *et al.* leverage this property to design an algorithm that adds all elements of v_h without inducing any rounding errors. This strategy has subsequently enabled accurate, vector-based algorithms for floating-point accumulation [31] [32] and dot products [26].

While Dekker's splitting is a well-studied algorithm that has been analyzed using mechanized proofs [3], `ExtractScalar` currently lacks such rigorous formalization. We have thus developed to our knowledge the first mechanized proof of `ExtractScalar` for P3109 formats. We present our key findings below.

Theorem 6 (`ExtractScalar` Properties)**.** *Let $\sigma, x \in \mathcal{D}_f \setminus \{\pm\infty, \mathtt{NaN}\}$ for a `Signed` format f such that $P > 1$. Let $\sigma = 2^i$ for some $i \in \mathbb{Z}$. Let $|x| \leq 2^{-j}\sigma$ for some $j \in \mathbb{N}$. If $|\sigma + x| \leq M^{hi}$, then for arbitrary saturation modes Sat_1, Sat_2, and Sat_3, the outputs x_h and x_ℓ satisfy the following conditions.*

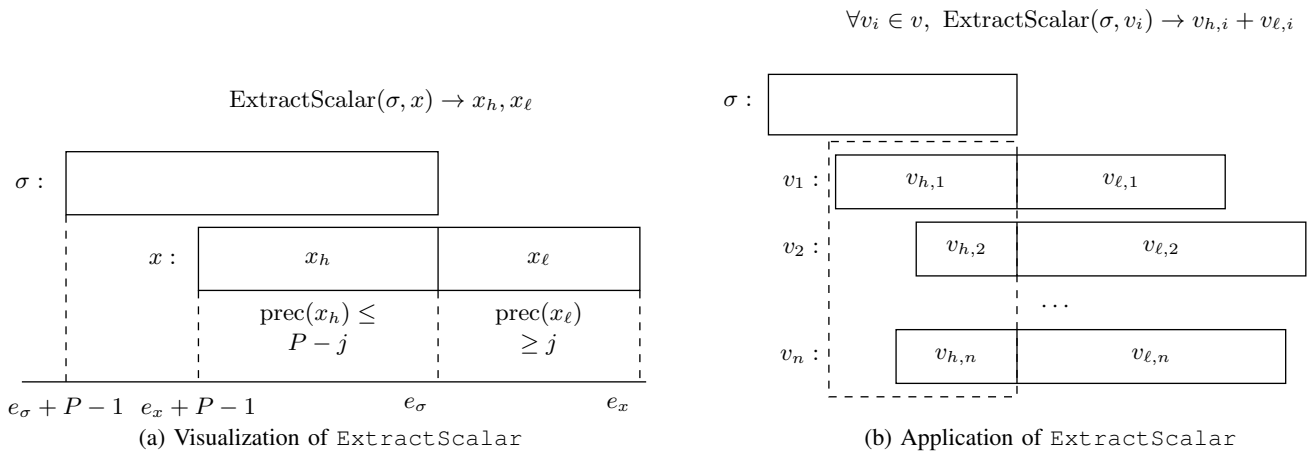
$$
\begin{aligned}
&(a)\ x = x_h + x_\ell \\
&(b)\ |x_\ell| \leq 2^{-P}\sigma \\
&(c)\ x_h \in 2^{-P}\sigma\mathbb{Z} \\
&(d)\ |x_h| \leq 2^{-j}\sigma
\end{aligned}
$$

Note that s and x_ℓ are the outputs of performing `FastTwoSum` (see Definition 1) on σ and x under RNE. Furthermore, `ExtractScalar`'s precondition $|x| \leq 2^{-j}\sigma$ implies $e_\sigma \geq e_x$. As such, Lemma 5 and Theorem 2 imply $x_h = s - \sigma$ and $x_\ell = \sigma + x - s$, respectively. Subsequently, $x = x_h + x_\ell$, and thus `ExtractScalar` is also an EFT under P3109 arithmetic. The core proofs for postconditions (b) through (d) are available in [31, Lemma 3.3, p. 201]. A direct application of these proofs to our model confirms these conditions for all `Signed` P3109 formats with precision at least two.

The original proof of the last condition $|x_h| \leq 2^{-j}\sigma$ leverages the properties of RNE. The proof asserts that when $\sigma + x$ is at the exact midpoint between σ and one of its floating-point neighbors, σ being an integer power of 2 induces $s = \sigma$ under RNE (see first operation in Definition 2). The equality $s = \sigma$ induces $x_h = 0$, which satisfies $|x_h| \leq 2^{-j}\sigma$ for the relevant values of j. However, the proof implicitly assumes all integer powers of 2 in the target format have even significands, which is not the case for P3109 formats where $P = 1$. As detailed in Section III-C, the tie-breaking for RNE when $P = 1$ uses the parity of the stored biased exponent. Thus, the floating-point sum of σ and x need not round to σ when $\sigma = 2^i$ for an odd integer i. Consequently, our analysis indicates that condition (d) does not hold for $P = 1$.

VI. Lean Development

We developed `FLoPS` using Lean 4. The development comprises approximately twenty-one thousand lines of code, covering the bit encodings, the algebraic model, the isomorphism proofs, the projection pipeline, and the analyses



(a) Visualization of `ExtractScalar` (b) Application of `ExtractScalar`

Fig. 6: (a) The horizontal axis shows exponent positions, with $e_\sigma \geq e_x$. `ExtractScalar`'s outputs x_h and x_ℓ satisfy $x = x_h + x_\ell$. Given $j \in \mathbb{N}$ such that $|x| \leq 2^{-j}\sigma$, the floating-point sum of σ and x rounds away at least j trailing bits of x, which are captured in x_ℓ. x_h, which captures x's leading bits, has at most $P - j$ effective precision bits. (b) Given a floating-point vector $v \in \mathcal{D}_f^n$, applying `ExtractScalar` to each element v_i produces vectors $v_h, v_\ell \in \mathcal{D}_f^n$ such that $v = v_h + v_\ell$. The elements of v_h collectively operate as *fixed-point* numbers.

of `FastTwoSum` and `ExtractScalar`. The mathematical model uses `Mathlib`'s `EReal` for reals and infinities, extended with `Unit` for `NaN`. The development also includes an executable bit-level kernel over Lean's `Fin`(2^K) type. Refinement proofs cover encoding, decoding, rounding, saturation, projection, and the core arithmetic operations; executable tests cover the standard's exceptional-value rules.

Errors discovered. During the process of formalizing the P3109 standard, we discovered multiple errors in the interim P3109 report [22]. We reported them to the working group, and the corrections have been incorporated into the draft standard. We highlight a few.

1) We discovered that the `emax` for unsigned formats was incorrectly specified. The interim report gave $2^W - 1 - B$ as `emax`, but this does not hold for representations with precision $P = 1$ and $P = 2$ as described in Section III-A.
2) We identified that the `encode` function had an error in the specification of infinity. The interim report had erroneously assigned $2^{K-1} - 2$ to $+\infty$ in the `Unsigned Extended` format; the correct code point is $2^K - 2$.
3) We also identified that a note in the report about projection was incorrect for RU; it has also been corrected.

VII. Related Work

The formalization of floating-point arithmetic [25] has a rich history spanning multiple proof assistants. One of the earliest efforts was Harrison's [18] formalization in HOL Light, which verified floating-point algorithms such as the exponential function. In PVS, Boldo and Muñoz [8] developed a parametric formalization that abstracts over format parameters. In Rocq, the PFF library [13], [3], [4] provided a high-level, axiomatic formalization of IEEE-754, but relied on rounding defined as a relation rather than a computable function, limiting its use for executable specifications. Flocq [6] addressed this by unifying PFF's axiomatic strengths with executability via the "Generic Format", and remains the state-of-the-art for IEEE-754 verification. Flocq is actively being ported to the `FloatSpec` [2] library in Lean. In Isabelle, Yu *et al.* [36] formalize IEEE-754 with a focus on executable representations. However, none of these frameworks can be directly adapted for P3109 due to structural mismatches in their rounding operator signatures and lack of saturation semantics (see Section III-C).

Closest to our work, Wintersteiger *et al.* [35] formalized the operational specification of P3109 using Imandra, producing an executable reference implementation with automatically extracted OCaml code. Their verification focuses on totality, proving that each operation always produces a valid P3109 code point rather than an error. Our work differs in scope: we establish a three-way isomorphism between bit encodings, an algebraic model, and the semantic domain; we formalize the full projection pipeline including all rounding and saturation modes; and we analyze the correctness properties of numerical algorithms (`FastTwoSum` and `ExtractScalar`) within P3109 arithmetic. SMT solvers such as Z3 [15] support IEEE-754 floating-point theory but do not model P3109's parametric formats or saturation modes.

Rounding-error analysis tools such as Gappa [14] and VCFloat2 [1] generate verification conditions tailored for IEEE-754's overflow-to-infinity semantics. P3109's saturation semantics can map overflow to a finite boundary, infinity, or `NaN`, behavior outside the scope of these tools. The RLibm project [24], [28] has recently analyzed `FastTwoSum` under round-to-odd [29], which is directly relevant to P3109 as it adopts round-to-odd as a rounding mode. Other efforts in correctly-rounded libraries [11], [33], automated error optimization [27], [12], and safety-critical analysis [34] address complementary aspects of floating-point verification but do not target P3109's parametric formats or saturation semantics.

VIII. Conclusion

The FLoPS framework provides a machine-checked specification of P3109's parametric formats and projection semantics in Lean, including stochastic rounding and saturation. We show a three-way equivalence between code points, our algebraic inductive type, and the datum set over $\mathbb{R}_\omega$. We show the usefulness of FLoPS by analyzing foundational FP algorithms such as FastTwoSum and the FP splitting technique ExtractScalar. Our verification of FastTwoSum revealed a novel property specific to P3109: when an overflowing sum is clamped to the corresponding finite boundary datum, the algorithm computes the exact overflow error regardless of the rounding modes used. Our analysis of ExtractScalar shows that previously known properties fail for formats with one-bit precision, highlighting the necessity of bespoke verification for ultra-low-precision formats.

As P3109-compliant hardware accelerators emerge, FLoPS's bit-level isomorphism provides a natural bridge for RTL equivalence checking: properties proven on our algebraic model transfer directly to the concrete bit encodings that hardware implements. The verified projection pipeline, including saturation and all rounding modes, can serve as a golden reference model against which hardware implementations are validated. Future work includes extending the algorithm analyses to Unsigned formats, which are relevant for machine learning operations such as ReLU that produce only nonnegative outputs, and connecting FLoPS to automated verification tools such as SMT-LIB's floating-point theory.

Acknowledgments

We thank the P3109 working group members especially Jeffrey Sarnoff and Christoph Wintersteiger for encouraging us to explore a Lean formalization. We would like to thank Ilya Sergey for his encouragement to explore Lean during Santosh's visit to NUS. This material is supported in part by the research gifts from the Intel corporation and National Science Foundation grants: NSF-2110861 and NSF-2312220. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Intel corporation or the National Science Foundation.

References

[1] Andrew Appel and Ariel Kellison. VCFloat2: Floating-point error analysis in Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2024, page 14–29, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3636501.3636953.

[2] Beneficial-AI-Foundation. FloatSpec: Formally verified float implementation with Lean 4, 2026. Ported from the Coq Flocq project. URL: https://github.com/Beneficial-AI-Foundation/FloatSpec.

[3] Sylvie Boldo. Pitfalls of a full floating-point proof: example on the formal proof of the Veltkamp/Dekker algorithms. In *Proceedings of the Third International Joint Conference on Automated Reasoning*, IJCAR'06, page 52–66, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11814771_6.

[4] Sylvie Boldo and Marc Daumas. Representable correcting terms for possibly underflowing floating point operations. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, ARITH '03, page 79, USA, 2003. IEEE Computer Society.

[5] 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. doi:10.1145/3054947.

[6] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic*, ARITH '11, page 243–252, USA, 2011. IEEE Computer Society. doi:10.1109/ARITH.2011.40.

[7] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. ISTE Press - Elsevier, 2017. URL: http://iste.co.uk/book.php?id=1238.

[8] Sylvie Boldo and César Muñoz. A high-level formalization of floating-point numbers in PVS. Technical Report CR-2006-214298, NASA, October 2006. NIA Report 2006-01.

[9] Tung-Che Chang, Sehyeok Park, Jay P. Lim, and Santosh Nagarakatte. Artifact for Semantics, Operations, and Properties of P3109 Floating-Point Representations in Lean . Zenodo, July 2026. doi:10.5281/zenodo.21660929.

[10] Tung-Che Chang, Sehyeok Park, Jay P Lim, and Santosh Nagarakatte. Flops: Formalization in the lean theorem prover of the p3109 standard, 2026. URL: https://github.com/rutgers-apl/FLoPS.

[11] Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. CR-LIBM: a correctly rounded elementary function library. In Franklin T. Luk, editor, *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, volume 5205, pages 458 – 464. International Society for Optics and Photonics, SPIE, 2003. doi:10.1117/12.505591.

[12] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. Daisy - framework for analysis and optimization of numerical programs (tool paper). In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 270–287, Cham, 2018. Springer International Publishing.

[13] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library for floating-point numbers and its application to exact computing. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, pages 169–184, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[14] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. Comput.*, 60(2):242–253, February 2011. doi:10.1109/TC.2010.128.

[15] 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.

[16] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18(3):224–242, June 1971. doi:10.1007/BF01397083.

[17] Andrew Fitzgibbon and Stephen Felix. On stochastic rounding with few random bits. In *2025 IEEE 32nd Symposium on Computer Arithmetic (ARITH)*, pages 133–140, 2025. doi:10.1109/ARITH64983.2025.00029.

[18] John Harrison. A machine-checked theory of floating point arithmetic. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '99, page 113–130, Berlin, Heidelberg, 1999. Springer-Verlag.

[19] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Arith 2001*, pages 155–162, 2001. doi:10.1109/ARITH.2001.930115.

[20] IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi:10.1109/IEEESTD.2019.8766229.

[21] IEEE SA P3109 Working Group. Draft Standard for Arithmetic Formats for Machine Learning, July 2026. Unapproved draft. URL: https://standards.ieee.org/ieee/3109/11165/.

[22] IEEE SA P3109 Working Group. Interim Report on Binary Floating-point Formats for Machine Learning, July 2026. URL:

https://github.com/P3109/Public/blob/main/IEEE%20P3109%20Interim%20Report.pdf.

[23] Claude-Pierre Jeannerod and Paul Zimmermann. FastTwoSum revisited. In *ARITH 2025*, pages 141–148, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/ARITH64983.2025.00030, doi:10.1109/ARITH64983.2025.00030.

[24] Jay P. Lim and Santosh Nagarakatte. One polynomial approximation to produce correctly rounded results of an elementary function for multiple representations and rounding modes. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498664.

[25] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[26] Katsuhisa Ozaki, Takeshi Ogita, Shin'Ichi Oishi, and Siegfried M. Rump. Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. *Numer. Algorithms*, 59(1):95–118, January 2012. doi:10.1007/s11075-011-9478-1.

[27] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11, June 2015. doi:10.1145/2813885.2737959.

[28] Sehyeok Park, Justin Kim, and Santosh Nagarakatte. Correctly rounded math libraries without worrying about the application's rounding mode. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi:10.1145/3729332.

[29] Sehyeok Park, Jay P. Lim, and Santosh Nagarakatte. Odd but error-free FastTwoSum: More general conditions for FastTwoSum as an error-free transformation for faithful rounding modes, 2026. URL: https://arxiv.org/abs/2601.17198, arXiv:2601.17198.

[30] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *ARITH 1991*, pages 132–143, 1991. doi:10.1109/ARITH.1991.145549.

[31] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation part i: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008. arXiv:https://doi.org/10.1137/050645671, doi:10.1137/050645671.

[32] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation part ii: Sign, k-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing*, 31(2):1269–1302, 2009. arXiv:https://doi.org/10.1137/07068816X, doi:10.1137/07068816X.

[33] 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. URL: https://inria.hal.science/hal-03721525, doi:10.1109/ARITH54963.2022.00014.

[34] Laura Titolo, Mariano Moscato, Marco A. Feliu, Paolo Masci, and César A. Muñoz. Rigorous floating-point round-off error analysis in PRECiSA 4.0. In *Formal Methods: 26th International Symposium, FM 2024, Milan, Italy, September 9–13, 2024, Proceedings, Part II*, page 20–38, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-71177-0_2.

[35] Christoph M. Wintersteiger. Formal verification of the IEEE P3109 standard for binary floating-point formats for machine learning. In *2025 IEEE 32nd Symposium on Computer Arithmetic (ARITH)*, pages 157–160, 2025. doi:10.1109/ARITH64983.2025.00032.

[36] Lei Yu. A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs*, July 2013. https://isa-afp.org/entries/IEEE_Floating_Point.html.