Novel Aspects of IEEE SA P3109 Arithmetic Formats for Machine Learning

Andrew Fitzgibbon
Graphcore
United Kingdom
Email: awf@graphcore.ai

Christoph M. Wintersteiger
Imandra, Inc.
United Kingdom
Email: christoph@imandra.ai

Jeffrey Sarnoff †
IEEE
USA
Email: jeffrey.sarnoff@ieee.org

***Abstract*—The IEEE P3109 draft standard defines a parameterized family of binary floating-point formats and associated operations, with a focus on facilitating machine learning. These formats allow efficient and consistent representation of values in a small number of bits. The defined formats are parameterized over width and precision in bits, signedness, and the presence of infinities. Operations are defined by decoding floating-point values to the set $\mathbb{R}^\omega$ of *closed extended reals*: the reals augmented with positive and negative infinity and NaN (Not a Number). Explicit treatment of NaN and infinite operands ensures that only real arithmetic is invoked in operation definitions. Extensive rounding and saturation modes are defined; stochastic rounding is included. Operations are exception-free, accelerating throughput, with exceptional situations communicated through return values, e.g., NaN. Operations on blocks of values sharing a common scale factor are defined in terms of the underlying operations in a uniform manner. System vendors may describe approximate implementations via a novel scale-invariant measure, akin to units in the last place, called κ-approximation. Standard function definitions and various other properties are mechanically verified and generated using formal specifications.**

I. Introduction

Floating-point arithmetic has been central to modern successes in the machine interpretation of video, language, and real-world patterns of numerous other kinds. In particular, the training of deep compositions of neural networks, or deep learning, depends on gradient descent in spaces of values represented by floating-point encodings. Initial work used IEEE 754 single-precision (32-bit) values, and demonstrated the improved runtime was more important than the loss of precision relative to double-precision (64-bit). Recent years have seen the introduction of 16, 8, and even 4-bit representations, in each case yielding more capable models for given wall-clock training and inference times. At sixteen bits, early systems made use of IEEE 754's "half precision" datatype, with five exponent bits, and ten mantissa bits. It was found, however that training machine learning (ML) models benefits from greater dynamic range, leading to the introduction [1] of the "bfloat16" format, with 8 exponent bits. The proliferation continued with the introduction of 8-bit formats [2], [3], with at least three different hardware implementations having been produced [2], [4], [5]. These implementations differ in whether formats include infinities, not-a-number (NaN) and subnormal values. Interoperability between vendors was an increasing concern. Given this context, the IEEE Standards Association created IEEE SA Working Group P3109 "Standard for Arithmetic Formats for Machine Learning" [6], tasked with developing a standard characterized by the project authorization as follows:

> This standard defines a binary arithmetic and data format for machine learning-optimized domains. It also specifies the default handling of exceptions that occur in this arithmetic. This standard provides a consistent and flexible arithmetic framework optimized for Machine Learning Systems (MLS) in hardware and/or software implementations to minimize the work required to make MLS interoperable with each other, as well as other dependent systems. This standard is aligned with IEEE-754 for Floating-Point Arithmetic.

The present paper describes the consensus of this working group as detailed in the group's public interim report [7], focusing on innovations and novel approaches not available with extant standards. Here is a summary of contributions from the P3109 standardization effort:

- A coherent family of signed and unsigned floating-point formats focused on narrow-bitwidth computations used in machine learning systems.
- A family of arithmetic operations parameterized over operand formats which may include either P3109 formats or external binary formats such as those in IEEE-754 or bfloat16.
- Operation definitions are automatically generated from formal specifications, ensuring well-defined outputs for all formats and over all input values.
- Selectable projections of results into floating-point values through the pairing of rounding and saturation modes, including round-to-odd and multiple levels of stochastic rounding.
- The extension of these operations to blocks of values sharing a common scale factor.
- A new scale invariant measure of operation accuracy, named κ-approximation, that is computed entirely in representation space.

† We acknowledge the many contributions of the members of IEEE SA working group P3109.

TABLE I
EXAMPLE P3109 FORMATS: ALL 4-BIT SIGNED FORMATS. DATUMS 0, 1 AND NAN ARE CONSISTENTLY ENCODED FOR ALL PRECISIONS.

Code point		Binary4p1se	Binary4p2se	Binary4p3se	Binary4p1sf	Binary4p2sf	Binary4p3sf
0	0b0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
1	0b0001	0.1250	0.2500	0.2500	0.1250	0.2500	0.2500
2	0b0010	0.2500	0.5000	0.5000	0.2500	0.5000	0.5000
3	0b0011	0.5000	0.7500	0.7500	0.5000	0.7500	0.7500
4	0b0100	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
5	0b0101	2.0000	1.5000	1.2500	2.0000	1.5000	1.2500
6	0b0110	4.0000	2.0000	1.5000	4.0000	2.0000	1.5000
7	0b0111	Inf	Inf	Inf	8.0000	3.0000	1.7500
8	0b1000	NaN	NaN	NaN	NaN	NaN	NaN
9	0b1001	-0.1250	-0.2500	-0.2500	-0.1250	-0.2500	-0.2500
10	0b1010	-0.2500	-0.5000	-0.5000	-0.2500	-0.5000	-0.5000
11	0b1011	-0.5000	-0.7500	-0.7500	-0.5000	-0.7500	-0.7500
12	0b1100	-1.0000	-1.0000	-1.0000	-1.0000	-1.0000	-1.0000
13	0b1101	-2.0000	-1.5000	-1.2500	-2.0000	-1.5000	-1.2500
14	0b1110	-4.0000	-2.0000	-1.5000	-4.0000	-2.0000	-1.5000
15	0b1111	-Inf	-Inf	-Inf	-8.0000	-3.0000	-1.7500

II. BACKGROUND AND NOTATION

A floating-point format is a mapping from *code points* to a subset of the real numbers, and *special* values such as infinities, NaN, and zero. Table I shows examples of six formats where the code points are four-bit integers and the special values are as defined in the P3109 specification: zero, $\mathsf{Inf}, \mathsf{-Inf}, \mathsf{NaN}$.

We define the set of *closed extended reals*

$$\mathbb{R}^\omega = \mathbb{R} \cup \{-\infty, \infty, \mathsf{NaN}\}.$$

Functions which operate on the closed extended reals are prefixed ω. When it is necessary to distinguish quantities in $\mathbb{R}^\omega$ from code points, the term *datum* is used, and a format's *datum set* is the subset of $\mathbb{R}^\omega$ to which its code points map. The term *floating-point value* is defined as "a codepoint representing a floating-point datum in a given format".

$\mathsf{IsOdd}(I)$ is true if integer I is odd, false otherwise. The number of elements in a set S is denoted $\#S$. Integer division is denoted by $x \div y$, modulo by $x \bmod y$.

III. DATUM SETS

An important consideration for the working group was the choice of datum set for the floating-point formats. Existing practice [2], [4], [5], [8] has shown that different applications may require variation in both the format's storage size in bits ("bitwidth") and in the relative number of exponent and significand bits. Existing practice also includes unsigned formats, for example as scale factors in block operations [8]. Finally, formats are provided in variants with or without infinity (see §III-D). Hence P3109 formats are parameterized over:

- Bitwidth K, the total size of the format in bits stored.
- Precision P, the number of bits in the significand including the implicit leading bit.
- Signedness $\Sigma \in \{\mathsf{Signed}, \mathsf{Unsigned}\}$.
- Domain of the datum set $\Delta \in \{\mathsf{Finite}, \mathsf{Extended}\}$.

It is required that $\mathsf{K} \geq 3$ and that $\mathsf{P} < \mathsf{K}$ for signed formats (and $\mathsf{P} \leq \mathsf{K}$ for unsigned). It is suggested that $\mathsf{K} < 16$ to avoid overlap with IEEE-754 formats, while it is understood that P3109 systems might use larger bitwidths for internal computation. For that reason, leaving K unbounded in this specification is useful.

This parameterization may give rise to a large number of formats; it would be impractical for any vendor of optimized software or accelerator hardware to provide accelerated implementations of all variants of operations and formats. P3109 serves as a source of operation definitions, which system suppliers may map to function names or operation codes supplied in their systems. Fig. 6 shows an example of how such a mapping may be supplied.

A. Naming

A given P3109 format is identified by the parameterized name $\mathsf{Binary}\{\mathsf{K}, \mathsf{P}, \Sigma, \Delta\}$. A shortened notation is also used to refer to specific formats: $\mathsf{Binary}\langle\kappa\rangle p\langle\psi\rangle\langle\sigma\rangle\langle\delta\rangle$ The placeholders κ and ψ are decimal representations of the bitwidth K and precision P, respectively; signedness $\sigma \in \{\mathsf{s}, \mathsf{u}\}$, for signed and unsigned; and domain $\delta \in \{\mathsf{e}, \mathsf{f}\}$ for extended and finite.

For example, the format "Binary12p7se" is a 12-bit signed format in the extended domain, with 1 sign bit, 5 exponent bits, and 7 bits of precision (of which only 6 are explicitly represented). As another example, the format "Binary6p1uf", is a 6-bit unsigned format in the finite domain, with no sign bit, 6 exponent bits, and 1 bit of precision (hence a zero-bit mantissa).

B. Not a Number (NaN)

P3109 formats specify a single NaN value, with rationale as follows.

Many existing floating-point formats define multiple NaN values which are returned from operations whose results lie outside the set of representable values, e.g. division of zero by zero, or addition of positive and negative infinities. P3109 formats define a single NaN, with the following rationale.

In machine learning systems, NaN is valuable for debugging code running on accelerator hardware, where exceptions may

TABLE II
ENCODINGS OF SELECTED VALUES FOR GIVEN BITWIDTH K, INDEPENDENT OF PRECISION.

Datum	Symbol	Signed extended	Signed finite	Unsigned extended	Unsigned finite
Zero	0.0	0	0	0	0
One	1.0	$2^{K-2}-0$	$2^{K-2}-0$	$2^{K-1}-0$	$2^{K-1}-0$
Not a Number	NaN	$2^{K-1}-0$	$2^{K-1}-0$	$2^{K-0}-1$	$2^{K-0}-1$
Positive Infinity	Inf	$2^{K-1}-1$	N/A	$2^{K-0}-2$	N/A
Negative Infinity	-Inf	$2^{K-0}-1$	N/A	N/A	N/A

be difficult or expensive to convey back to the user, hence at least one NaN value is required.

NaNs also have utility as a sentinel value. In some datasets, for example, individual element values may be missing or out of range; a sentinel may be used to record the positions of these values. Infinities can serve as a missing value indicator, but given the restricted range of P3109 formats, infinity is likely to become used as a separate saturation indicator.

Multiple NaN values are known in some statistical computation systems (e.g., the R system has NaN and NA), but these features are not widely used. In the context of P3109, supporting multiple NaNs would reduce the already limited encoding space.

C. Zero

P3109 formats specify a single zero value, with no sign. The inclusion of negative zero would incur the cost of an additional code point. Given the decision to encode only a single NaN, placing that NaN at the code point where IEEE-754 encodes negative zero enables the strictly positive and strictly negative number ranges to be symmetric for signed formats.

A key rationale for including -0 in IEEE-754 was the consistent implementation of branch cuts in the ArcTan2 function and the complex trigonometric functions [9], [10]. In P3109, these functions return NaN for inputs undefined in the reals, unless the limit approaching these inputs is path-independent. For example, $\mathsf{ArcTan2}(0,0) = \mathsf{NaN}$, since the limit depends on the path of approach, as with $\mathsf{ArcTan2}(\mathsf{Inf},\mathsf{Inf}) = \mathsf{NaN}$.

Similarly $\mathsf{Divide}(X,0) = \mathsf{NaN}$ for all X, since the limit depends on the sign of the approach to zero. This also avoids inconsistency of the form $1/(1/-\infty) = \infty$.

It was considered that the use of integer comparisons in sorting would weigh against placing NaN at the negative zero code point. For example, the JAX machine learning framework is known to sort using integer comparison [11]. However, such sorting still requires $O(n)$ preprocessing and postprocessing steps to enable the use of two's-complement integer comparison, and already has special treatment of NaN and -0, so the cost of eliminating -0 and placing NaN in the -0 position is negligible.

D. Infinities

Formats are parameterized over the extended and finite domains, that is the inclusion or exclusion of infinite values.

Infinite values are used widely in machine learning systems, for example mask values in transformer models, or to represent overflow to adjust dynamic loss scaling factors [12]. Representing infinite values requires two code points in a signed format, and for narrow formats, the reduction in the number of finite values may be significant, hence the choice is parameterized.

E. Exponent bias

In IEEE-754, the exponent bias of a format with bitwidth K and precision P is determined by consideration of the largest finite value emax, and bias is chosen so that emax has an exponent of $2^{K-P-1}-1$. Because P3109 formats are parameterized over the presence of infinities, it is more consistent to use bias, rather than emax, as the characterizing parameter of the format, since otherwise the presence of infinities would affect the bias and hence the code point-to-value mapping for finite values. The IEEE-754 behavior is maintained for the majority of signed formats by choosing a bias of 2^{K-P-1}. For unsigned formats, the bias is doubled: 2^{K-P}.

A valuable consequence of the specifications is that the value 1.0 encodes to the midway code point, that is 2^{K-2} for signed formats and 2^{K-1} for unsigned.

Table II shows the code point mappings for selected values as a function of K.

F. Subnormals

Subnormal numbers extend the dynamic range of floating-point values and induce equal quantization steps close to zero. Machine learning accelerators have historically chosen either to implement subnormals, or to flush them to zero, but recent practice has tended towards their inclusion. At present, all known implementations of 8-bit floating-point implement subnormals. Hence, P3109 formats with precisions greater than 1 include subnormals. Note that under κ-approximation (§VI), a system vendor may supply additional implementations of some *operations* which flush subnormals to zero, but this does not change the fact that the *format* still includes code points in the subnormal regime.

IV. OPERATIONS

Operations are defined via conversion to closed extended real values, on which the mathematical operation is performed, before conversion back to the appropriate datum set. In general, operation results will not be members of the datum set

Signature

$\omega\mathsf{RoundToPrecision}_{\mathsf{P},\mathsf{B},\mathsf{Rnd}}(X) \to Z$

Parameters

P : integer precision
B : exponent bias
Rnd : rounding mode

Operands

X : closed extended real value

Result

Z : closed extended real value, of the form $M \times 2^E$ if finite

Behavior

$\omega\mathsf{RoundToPrecision}(X \in \{0, -\infty, \infty, \mathsf{NaN}\}) \to X$
$\omega\mathsf{RoundToPrecision}(X) \to Z$

where

Compute exponent as integer.

$\hat{E} = \lfloor \log_2(|X|) \rfloor$

Truncate exponent to subnormal range.

$E = \max(\hat{E}, 1 - \mathsf{B}) - \mathsf{P} + 1$

$S = |X| \times 2^{-E}$

$$I = \begin{cases} \lfloor S \rfloor + 1 & \textbf{if } \mathsf{RoundAway}(\mathsf{Rnd}) \\ \lfloor S \rfloor & \textbf{otherwise} \end{cases}$$

$Z = \mathrm{sign}(X) \times I \times 2^E$

Fig. 1. Rounding is expressed as a function $\mathbb{R}^\omega \mapsto \mathbb{R}^\omega$, with mode-specific behavior defined by the function RoundAway (Fig. 2).

and hence will be *projected* into the datum set via rounding and saturation.

As an example, the specification of the exponential function $\mathsf{Exp}(x) \to r$ involves the following steps:

$$\begin{aligned} X &= \omega\mathsf{Decode}(x) && \text{where } X \in \mathbb{R}^\omega \\ R &= \omega\mathsf{Exp}(X) && \text{where } R \in \mathbb{R}^\omega \\ r &= \omega\mathsf{Project}_{f,\rho}(R) && \text{result } r \text{ as integer code point} \end{aligned}$$

where ωDecode converts a code point (represented as an integer) x to a value in the extended reals X, and ωProject maps the extended real result R to a value r in the datum set of the target format f. The rounding mode and saturation are described in a *projection specification* ρ.

The ωProject function is defined as the composition of rounding to the target precision, followed by saturation to the target format's datum set. It takes the projection specification ρ, the target format f, and a closed extended real value R, and returns the encoded result r:

$$\begin{aligned} \omega\mathsf{Project}_{f,\rho}(R) \to r, \text{ where} \\ R_r &= \omega\mathsf{RoundToPrecision}(R) \\ R_s &= \omega\mathsf{Saturate}(R_r) \\ r &= \omega\mathsf{Encode}(R_s) \end{aligned}$$

A. Rounding modes

The supported rounding modes are: round to nearest, with ties to even or away from zero; round toward positive, negative, or zero; round inexact to odd; and three variants of stochastic rounding [13]. The precise behavior of these modes is defined in the function $\omega\mathsf{RoundToPrecision} : \mathbb{R}^\omega \mapsto \mathbb{R}^\omega$, (Fig. 1) which takes a closed extended real and returns a closed extended real which, if finite, is of the form $M \times 2^E$ where M and E are integers.

$\mathsf{RoundAway} : \mathsf{Rnd} \to \mathsf{Boolean} =$

$\mathsf{TowardZero} \to \mathsf{False}$

$\mathsf{TowardPositive} \to \eta > 0 \textbf{ and } X > 0$

$\mathsf{TowardNegative} \to \eta > 0 \textbf{ and } X < 0$

$\mathsf{NearestTiesToAway} \to \eta \geq 0.5$

$\mathsf{NearestTiesToEven} \to \eta > 0.5 \textbf{ or} (\eta = 0.5 \textbf{ and } \mathsf{CodeIsOdd})$

$\mathsf{ToOdd} \to \eta > 0 \textbf{ and not } \mathsf{CodeIsOdd}$

$\mathsf{StochasticA}_{N,R} \to \lfloor \eta \times 2^N \rfloor + R \geq 2^N$

$\mathsf{StochasticB}_{N,R} \to \lfloor \eta \times 2^{N+1} \rfloor + (2 \times R + 1) \geq 2^{N+1}$

$\mathsf{StochasticC}_{N,R} \to \mathsf{RNITE}(\eta \times 2^N) + R \geq 2^N$

where

$$\mathsf{CodeIsOdd} = \begin{cases} \mathsf{IsOdd}(\lfloor S \rfloor) & \textbf{if } \mathsf{P} > 1 \\ \mathsf{IsOdd}(E + \mathsf{B}) \textbf{ and } (\lfloor S \rfloor \neq 0) & \textbf{if } \mathsf{P} = 1 \end{cases}$$

$$\mathsf{RNITE}(X) = \begin{cases} \lfloor X \rfloor & \textbf{if } (X < \lfloor X \rfloor + 0.5) \\ \lfloor X \rfloor + 1 & \textbf{if } (X > \lfloor X \rfloor + 0.5) \\ \lfloor X \rfloor + \mathsf{IsOdd}(\lfloor X \rfloor) & \end{cases}$$

Fig. 2. Auxiliary function RoundAway takes the rounding mode and truncated fraction $\eta = S - \lfloor S \rfloor$ to determine whether to round away from zero. Stochastic rounding modes are explicitly supplied with random bits $0 \leq R < 2^N$.

B. Saturation modes

The ωSaturate function takes a closed extended real, and the maximum and minimum finite values of the target format ($\{M^{\mathrm{lo}}, M^{\mathrm{hi}}\} \subset \mathbb{R}$) and returns a closed extended real datum, according to the following modes:

SatFinite: All return values are clamped to the representable finite range, so e.g. $-\infty$ yields M^{lo}. This is the only defined saturation mode when the operation result's format domain is Finite.

SatPropagate: Finite return values are clamped to the representable finite range. Infinite return values are preserved, so e.g. $-\infty$ yields $-\infty$.

SatNone: Values outside $[M^{\mathrm{lo}}, M^{\mathrm{hi}}]$ are mapped to $\pm\infty$, following IEEE-754. For unsigned formats, values below 0 are clamped to 0.

Both SatPropagate and SatNone are specified to map any value greater than M^{hi} to $+\infty$, but because rounding precedes saturation, values which round to M^{hi} will be projected to M^{hi}. Hence the IEEE-754 property that values between M^{hi} and the number half a unit in the last place above M^{hi} will project to M^{hi} is maintained.

C. Operation specification

Operation definitions follow the template shown in Fig. 3. An operation is *parameterized* by the formats of its operands and result, and the projection specification. The combination of

Signature

$\mathsf{Divide}_{f_x,f_y,f_r,\rho}(x,y) \to r$

Parameters

f_x : format of x
f_y : format of y
f_r : format of r
ρ : projection specification

Operands

x : floating-point value, format f_x
y : floating-point value, format f_y

Result

r : floating-point value, format f_r

Behavior

$\omega\mathsf{Divide}(\mathsf{NaN}, *) \to \mathsf{NaN}$
$\omega\mathsf{Divide}(*, \mathsf{NaN}) \to \mathsf{NaN}$
$\omega\mathsf{Divide}(+\infty, \pm\infty) \to \mathsf{NaN}$
$\omega\mathsf{Divide}(-\infty, \pm\infty) \to \mathsf{NaN}$
$\omega\mathsf{Divide}(*, 0) \to \mathsf{NaN}$
$\omega\mathsf{Divide}(+\infty, Y)$ **if** $Y > 0 \to +\infty$
$\omega\mathsf{Divide}(+\infty, Y)$ **if** $Y < 0 \to -\infty$
$\omega\mathsf{Divide}(-\infty, Y)$ **if** $Y > 0 \to -\infty$
$\omega\mathsf{Divide}(-\infty, Y)$ **if** $Y < 0 \to +\infty$
$\omega\mathsf{Divide}(*, \pm\infty) \to 0$
$\omega\mathsf{Divide}(X, Y) \to X/Y$
$\mathsf{Divide}(x, y) \to \mathsf{Project}_{f_r,\rho}(\omega\mathsf{Divide}(X, Y))$
where
$X = \mathsf{Decode}_{f_x}(x)$
$Y = \mathsf{Decode}_{f_y}(y)$

Fig. 3. Operation definitions follow a standard template: *Parameters* specify static quantities such as formats and rounding modes; *Operands* and *Result* specify value-dependent quantities; Behavior is defined by a sequence of pattern-matching rules, considered in order, and returning the right-hand-side value of the first matching pattern. A floating-point operation Op may be defined in terms of a closed extended real operation ωOp, whose properties are proved by formal verification, and from which this specification is mechanically generated (§V).

an operation and its parameters is an *operation specialization*. The *operands* and *result* are values in the specified formats. The *behavior* is defined by a sequence of pattern-matching rules, considered in order, and returning the right-hand-side value of the first matching pattern.

V. Formal verification

To ensure the consistent and unsurprising behavior of operation definitions with arguments of any formats, we use formal specifications that enable us to prove properties of the standard formally. The operation definitions are generated from formal specifications in the Imandra Modelling Language (IML) [14]; these are mechanically checked for completeness and consistency. Most properties we prove are of the shape as shown in Fig. 4, which is a formal specification of the Divide operation, from which the definition in Fig. 3 is later automatically generated. The formal specification is also used to prove properties of the operations, for example that no combination of inputs to Divide will reach the `Error` clause, for all formats.

For internal functions such as ωEncode and ωDecode, a number of further proofs are necessary to support verification

```
(** 4.11.5 Division *)

let wDivide (x : CER.t) (y : CER.t) :
            (CER.t, string) Result.t =
  let open CER in
  match x, y with
  | NaN, _               -> Ok NaN
  | _, NaN               -> Ok NaN
  | PInf, (PInf | NInf) -> Ok NaN
  | NInf, (PInf | NInf) -> Ok NaN
  | _, R 0.0             -> Ok NaN
  | PInf, R y when y >. 0.0 -> Ok PInf
  | PInf, R y when y <. 0.0 -> Ok NInf
  | NInf, R y when y >. 0.0 -> Ok NInf
  | NInf, R y when y <. 0.0 -> Ok PInf
  | _, (PInf | NInf) -> Ok (R 0.0)
  | R x, R y -> Ok (R (x /. y))
  | _ -> Error "undefined"

let divide f_x f_y f_r rho x y =
  match
    wDivide (wDecode f_x x) (wDecode f_y y)
  with
  | Ok r -> wProject f_r rho r
  | Error e -> Error e

(** Theorem: "divide never returns an error" *)
theorem divide_ok f_x f_y f_r rho x y  =
  Result.is_ok (divide f_x f_y f_r rho x y)
```

Fig. 4. The standards text for the behavior of Divide in Fig. 3 is automatically generated from the formal specification shown here. The example theorem proves that the `Error` clause in `wDivide` is not reached for any combination of inputs, of any combination of formats.

of other operations. For instance, ωEncode requires the input value to be in the datum set of the target format, a property that must be established by ωRoundToPrecision and ωSaturate. As another example, ωDecode must correctly produce a value within the format range of the input format. Proofs of correctness of block operations rely on the correctness of the underlying scalar operations, but require numerous proofs of correctness of the control flow of various intermediate functions.

Overall, the formalization currently contains around 500 theorems, which also includes some more experimental properties like the convergence of square-root computations and some proofs of equivalence between different specifications of the same operation. The latter were particularly helpful during development of the standard, as they allowed changes to be made without introducing regressions.

While most useful during development of the standard, the formal specification also serves as a reference for further mathematical analysis of the standard after its publication. Since it is executable, the standard enables efficient test vector generation and serves as a test oracle for implementations. It is freely available online [15] and an earlier version is described in more detail in [16]. A recent reformulation in Lean [17] has established additional properties, for example regarding `FastTwoSum`.

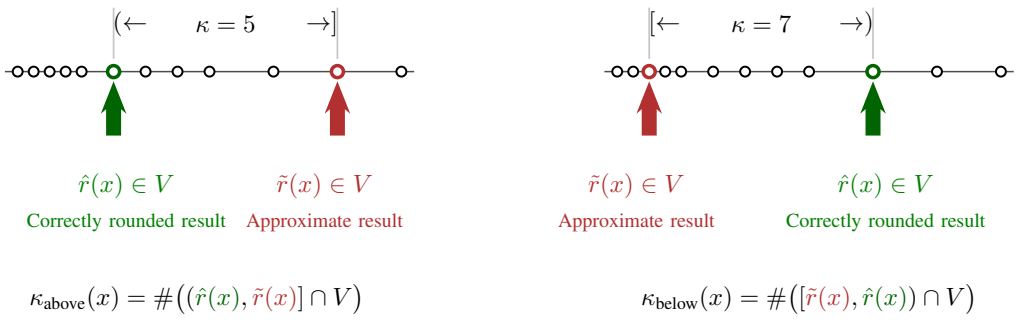

Fig. 5. Kappa-approximation. An approximate implementation $\tilde{r}$ of an operation a produces results differing from the correctly rounded result $\hat{r}$ by at most κ value steps. The calculation is in terms of the interval between $\tilde{r}(x)$ and $\hat{r}(x)$, inclusive of the former, exclusive of the latter, expressed as $\big(\hat{r}(x), \tilde{r}(x)\big] \cup \big[\tilde{r}(x), \hat{r}(x)\big)$ in §VI.

VI. Approximate Implementations

For numeric operations, in addition to an exact implementation, which must be provided, a system may additionally provide κ-*approximate* implementations. The κ-approximation is a measure of the accuracy of an approximate implementation, akin to units in the last place, but precisely defined for all results.

A numeric operation a, for a given set of parameters, has a defined correctly rounded result $\hat{r}(x) = a(x)$ for operand x (which may be a tuple of values). A κ-approximate implementation produces a floating-point value $\tilde{r}(x)$, which for some operand x has $\tilde{r}(x) \neq \hat{r}(x)$.

Let the set of operands producing finite results be X, so that for all $x \in X$ we have $\hat{r}(x) \in V$, where V is the result format's finite value set. For all $x \in X$, assume the approximation $\tilde{r}(x) \in V$, i.e. $\tilde{r}(x)$ "agrees with" $\hat{r}(x)$ on returned infinities or NaN for all inputs. For other cases, see [7].

The value of κ for an operation specialization will be the maximum over all operands $x \in X$ of the number of values in V between $\tilde{r}(x)$ and $\hat{r}(x)$ inclusive of the former, exclusive of the latter, as illustrated in Fig. 5. Formally,

$$\kappa = \max_{x \in X} \#\left(\Big(\big(\hat{r}(x), \tilde{r}(x)\big] \cup \big[\tilde{r}(x), \hat{r}(x)\big) \Big) \cap V \right)$$

Such specifications may be over X or over a covering union of disjoint subsets of X, as in the example in Fig. 6, where κ is defined over three input intervals.

For each κ-approximate implementation of an operation specialization, a different κ will apply in general. For example a system may supply implementations of $\mathsf{Add}_{f_x,f_y,f_r,\rho}$ where κ depends on f_x, f_y, f_r and ρ.

VII. Encoding

A P3109 datum in a K-bit format is encoded by an integer in the range 0 to $2^K - 1$. A detailed specification of the encoding and decoding operations is given in Fig. 7, following the same pattern as operation definitions in Fig. 3. These operations also accept external formats such as binary16, binary32, and bfloat16, enabling all P3109 operations to be specialized over mixed P3109 and external formats.

The instruction `exp8.px.sat` has the following variants: `px` in {`3`,`4`} and `sat` in {`finite`,`inf`}, implementing P3109 Exp operation variants as follows:

```
exp8.3.finite: Exp{Binary8p3se, Binary8p4se, RNS}
exp8.4.finite: Exp{Binary8p4se, Binary8p4se, RNS}
exp8.3.inf: Exp{Binary8p3se, Binary8p4se, RNI}
exp8.4.inf: Exp{Binary8p4se, Binary8p4se, RNI}  where
RNS = (NearestTiesToEven, SatFinite)
RNI = (NearestTiesToEven, OvfInf)
```

In addition the operation `fastexp8.3.inf(x)` approximates `Exp{Binary8p3se, Binary8p4se, RNI}(x)`

$$\text{with } \kappa(x) = \begin{cases} 0 & \text{if } \omega\mathsf{Decode}(x) \in (-\infty, -12.3) \\ 1 & \text{if } \omega\mathsf{Decode}(x) \in (-12.3, 1.2) \\ 2 & \text{otherwise} \end{cases}$$

Fig. 6. Operation mapping. An example showing how system-supplied operation names may be defined in terms of P3109 specifications. Here `Exp{f_x, f_r, rho}` is a textual representation of the P3109 operation $\mathsf{Exp}_{f_x,f_r,\rho}$. Such mappings may be generated from configuration files.

All formats contain a single zero, encoded by the integer 0. All formats contain a single NaN. For signed formats, NaN is encoded at the code point which IEEE-754 uses for negative zero: 2^{K-1}. For unsigned formats, NaN is encoded at $2^K - 1$. Formats in the extended domain contain one or two infinities. For signed formats, Inf is encoded at $2^{K-1} - 1$ and -Inf is encoded at $2^K - 1$. For unsigned formats, Inf is encoded at $2^K - 2$.

VIII. Block Operations

Recent accelerator hardware has introduced *block operations*, in which sequences of values sharing a common scale factor are processed together. Existing systems [8] allow the scale factor to be a floating-point value, or to be a power of two. In P3109, a block is a pair $(s, [x_1, ..., x_B])$ comprising scale factor s in format f_s and a sequence of one or more elements x_i, each in format f_x. This implies nothing about the representation of the block in memory or on a communications channel; it purely defines the behavior of arithmetic operations

on blocks. Because P3109 includes unsigned formats, and formats which are pure powers of two (P = 1), existing hardware implementations are represented in a consistent manner.

Operations on blocks are defined on the closed extended reals, using the same ω functions, with lifting to $\mathbb{R}^\omega$ via ωDecode and projection via ωRoundToPrecision and ωSaturate as for scalar operations. Following the example of Exp in §IV, the operation BlockExp is defined as follows:

$$\begin{aligned}&\mathsf{BlockExp}((s,[x_1,...,x_B]),s_r) \to (s_r,[r_1,...,r_B])\\ &\quad\textbf{where}\\ &\qquad S = \omega\mathsf{Decode}_{f_s}(s)\\ &\qquad X_i = \omega\mathsf{Multiply}(S,\omega\mathsf{Decode}_{f_x}(x_i))\\ &\qquad Z_i = \omega\mathsf{Exp}(X_i)\\ &\quad [r_1,...,r_B] = \mathsf{BlockProject}(s_r,[Z_1,...,Z_B])\end{aligned}$$

The definition of BlockProject follows the same pattern as for scalar operations, applying ωProject to each element of the block:

$$\begin{aligned}&\mathsf{BlockProject}(s,[X_1,...,X_B]) \to [r_1,...,r_B] \quad \text{where}\\ &\quad S = \omega\mathsf{Decode}_{f_s}(s)\\ &\quad Z_i = \begin{cases}\mathsf{NaN} & \textbf{if } S \text{ is } \mathsf{NaN} \textbf{ or } X_i \text{ is } \mathsf{NaN}\\ 0 & \textbf{if } S = 0\\ \mathrm{sgn}(X_i)\times\mathrm{sgn}(S) & \textbf{if } S = \pm\infty\\ \omega\mathsf{Divide}(X_i,S) & \textbf{otherwise}\end{cases}\\ &\quad r_i = \omega\mathsf{Project}_{f_r,\rho_r}(Z_i)\end{aligned}$$

The definition in terms of ωDivide is equivalent to multiplication by the reciprocal as the operation is in the closed extended reals. An implementation may implement this in any convenient manner, typically avoiding explicit division.

A key feature of this definition is that the scale factor s_r of the *result* block is supplied as a parameter to the operation. At first glance, this may appear incorrect: the scale factor of the result block should be determined by the operation, for example to maximize the precision of the result. However, in practice, existing implementations of block operations offer a wide variety of schemes to determine the result scale factor, and it is impractical to define a single scheme which meets all needs. To define the *behavior* of the operation, the result scale factor is supplied as a parameter, even though an implementation will generally choose to compute it internally. Such an operation $\mathsf{BlockOp}\mathit{ScalingAlgorithm}((s,[x_1,...,x_B])) \to (s_r,[r_1,...,r_B])$, which computes and returns s_r, shall have the property that its result is identical to that of $\mathsf{BlockOp}((s,[x_1,...,x_B]),s_r)$. One provided example in the standard is ConvertToBlock, with a corresponding ConvertToBlockMaxAbsFinite scaling algorithm, which chooses the result scale factor to be the smallest scale factor such that all results are representable finite values.

The draft standard also defines block reduction operations such as BlockReduceAdd and BlockDotProduct.

IX. Next float above or below

Operations analogous to IEEE-754's **nextUp** and **nextDown** operations are defined for P3109 formats. In IEEE-754, the operation **nextUp**(x) is described as returning the "least floating-point number that compares greater than x". However the definition therein specifies that $\mathbf{nextUp}(+\infty) = +\infty$, which is inconsistent with that textual description.

In P3109 this inconsistency would have further implications in the presence of finite-domain formats, so the wording is changed to "least floating-point number that compares greater than x, or NaN", and the operations are renamed to NextLessThan and NextGreaterThan, with behavior identical to IEEE-754 for non-extreme values, and adjusted definitions for extreme values and zero. For reference, Table III shows the values of NextGreaterThan for some special values for various combinations of signedness and domain.

TABLE III
NEXT FLOAT ABOVE OR BELOW

Value	754	SE	SF	UE	UF
-Inf	-MAX	=			
-MAX	=	=	=		
-MIN	-0	0	0		
-0	0				
0	MIN	=	=	=	=
MAX	Inf	=	NaN	=	NaN
Inf	Inf	NaN		NaN	

In this table, entries "=" indicate the behavior is identical to 754, while blank entries mean that the input value is not present in the P3109 format, and MAX is the largest finite value, and MIN the smallest subnormal. The columns SE, SF, UE, UF indicate Signed Extended, Signed Finite, Unsigned Extended, and Unsigned Finite, respectively.

X. Discussion and Limitations

This paper has presented an overview of the key aspects of the P3109 floating-point standard for machine learning systems. A number of other topics in the standard have not been covered here, such as format-level operations such as $\mathsf{MaxFiniteOf}(f)$ and $\mathsf{ExponentBitwidthOf}(f)$.

The specification defines scaled operations, for example of the form $\mathsf{ScaledMultiply}(s,a,b) = 2^s \times a \times b$. These are specified in terms of single-element block operations.

The choice of bias means that the value ranges of some existing 8-bit formats differ from P3109 formats by approximately a factor of two. While this has not proven to be an issue in the training of machine learning models, direct conversion of existing inference models might involve unacceptable loss of accuracy. However, the widespread use of block scaling and operation scaling will generally allow for this loss to be compensated, at the possible cost of additional code complexity.

Conversely, P3109 operations may be instantiated entirely with external formats. For example, an implementation might supply Add with all formats as binary16 and $\rho = (\mathsf{StochasticB}_{16}, \mathsf{SatNone})$, which describes stochastic rounding in binary16.

Signature

$\omega\mathsf{Decode}_f(x) \to X$

Parameters

f : format of x

Operands

x : floating-point value, in format f

Result

X : closed extended real value

Behavior

$$\omega\mathsf{Decode}(\mathsf{Signed}, *, 2^{\mathsf{K}-1}) \to \mathsf{NaN}$$
$$\omega\mathsf{Decode}(\mathsf{Unsigned}, *, 2^{\mathsf{K}} - 1) \to \mathsf{NaN}$$
$$\omega\mathsf{Decode}(\mathsf{Signed}, \mathsf{Extended}, 2^{\mathsf{K}-1} - 1) \to +\infty$$
$$\omega\mathsf{Decode}(\mathsf{Signed}, \mathsf{Extended}, 2^{\mathsf{K}} - 1) \to -\infty$$
$$\omega\mathsf{Decode}(\mathsf{Unsigned}, \mathsf{Extended}, 2^{\mathsf{K}-1} - 2) \to +\infty$$
$$\omega\mathsf{Decode}(\mathsf{Signed}, \Delta, 2^{\mathsf{K}-1} < x < 2^{\mathsf{K}}) \to -\omega\mathsf{Decode}(\mathsf{Signed}, \Delta, x - 2^{\mathsf{K}-1})$$
$$\omega\mathsf{Decode}(*, *, x) \to \begin{cases} (0 + T \times 2^{1-\mathsf{P}}) \times 2^{1-\mathsf{B}} & \textbf{if } E = 0 \\ (1 + T \times 2^{1-\mathsf{P}}) \times 2^{E-\mathsf{B}} & \textbf{otherwise} \end{cases}$$

where

$$T = x \bmod 2^{\mathsf{P}-1}$$
$$E = x \div 2^{\mathsf{P}-1}$$

Signature

$\omega\mathsf{Encode}_f(X) \to r$

Parameters

f : result format

Operands

X : closed extended real value, in the value set of f

Result

r : floating-point value, format f

Behavior

$$\omega\mathsf{Encode}(\mathsf{NaN}) \to \begin{cases} 2^{\mathsf{K}-1} & \textbf{if } \sigma = \mathsf{Signed} \\ 2^{\mathsf{K}} - 1 & \textbf{if } \sigma = \mathsf{Unsigned} \end{cases}$$
$$\omega\mathsf{Encode}(+\infty) \to \begin{cases} 2^{\mathsf{K}-1} - 1 & \textbf{if } \sigma = \mathsf{Signed} \\ 2^{\mathsf{K}} - 2 & \textbf{if } \sigma = \mathsf{Unsigned} \end{cases}$$
$$\omega\mathsf{Encode}(X < 0) \to \omega\mathsf{Encode}_f(-X) + 2^{\mathsf{K}-1}$$
$$\omega\mathsf{Encode}(0) \to 0$$
$$\omega\mathsf{Encode}(X > 0) \to \begin{cases} T & \textbf{if } S < 2^{\mathsf{P}-1} \\ T + (E + \mathsf{B}) \times 2^{\mathsf{P}-1} & \textbf{otherwise} \end{cases}$$

where

$$E = \max(\lfloor \log_2(X) \rfloor, 1 - \mathsf{B})$$
$$S = X \times 2^{-E} \times 2^{\mathsf{P}-1}$$
$$T = S \bmod 2^{\mathsf{P}-1}$$

Fig. 7. Specification of the $\omega\mathsf{Decode}$ and $\omega\mathsf{Encode}$ operations, lightly paraphrased from the standard (which uses auxiliary operations $\omega\mathsf{DecodeExternal}$ and $\omega\mathsf{DecodeAux}$, and more carefully extracts bitwidth K, precision P, and bias B from f). In contrast to IEEE-754, explicit integer arithmetic is used to extract the exponent and significand fields, but implementing hardware does not need to implement integer arithmetic if bitfield extraction is more convenient. $\omega\mathsf{Encode}$ has the precondition that X is in the datum set of format f, which is formally verified for all standard operations (§V). Hence it follows that $S \in \mathbb{N}$, so there is no floor or round operation in $\omega\mathsf{Encode}$, and that X is finite in finite formats.

The definitions are coherent for bitwidths of three and above. It is feasible to extend these to two-bit formats, but a number of inconsistencies arise (see the appendix in the interim report [7]), which suggest that such formats may better be seen as application-specific.

Encodings are defined in terms of integers, rather than machine bit patterns. This allows for a precise presentation and does not require that an implementation offer any form of integer arithmetic, but transfers machine endianness to floats, which may conflict with other uses on some hardware.

Only operations common in the machine learning literature are defined. It is hoped that the principles of definition are clear from these examples, so that future vendors defining newly-required operations will naturally arrive at compatible definitions. It would be expected that such definitions would be incorporated in any updated release of the standard.

REFERENCES

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "Cloud TPU: Machine learning accelerators for training and inference," *IEEE Micro*, vol. 38, no. 2, pp. 39–47, 2018.

[2] B. Noune, P. Jones, D. Justus, D. Masters, and C. Luschi, "8-bit numerical formats for deep neural networks," *arXiv:2206.02915*, 2022.

[3] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "FP8 formats for deep learning," *arXiv:2209.05433*, 2022.

[4] P. Micikevicius, S. Oberman, P. Dubey, M. Cornea, A. Rodriguez, I. Bratt, R. Grisenthwaite, N. Jouppi, C. Chou, A. Huffman, M. Schulte, R. Wittig, D. Jani, and S. Deng, "OCP 8-bit floating point specification (OFP8) revision 1.0," tech. rep., opencompute.org, 2023.

[5] Tesla, Inc., "Tesla Dojo Technology: A guide to Tesla's configurable floating point formats and arithmetic," 2023.

[6] IEEE, "P3109 standard for arithmetic formats for machine learning." https://standards.ieee.org/ieee/3109/11165/.

[7] P3109 Working Group, "Interim report," 2026. https://github.com/P3109/Public.

[8] Open Compute Project, "OCP microscaling formats (mx) specification version 1.0." Open Compute Project Foundation, 2023.

[9] W. Kahan, "Branch cuts for complex elementary functions or much ado about nothing's sign bit," *Institute of Mathematics and its Applications Conference*, 1987.

[10] W. Kahan and J. W. Thomas, "Augmenting a programming language with complex arithmetic," tech. rep., EECS Department, University of California, Berkeley, 1991.

[11] Google, "Jax: Lax function `_float_to_int_for_sort`." https://github.com/google/jax path jax/_src/lax/lax.py#L3934, Commit fc5960f2 (accessed 2026-02-13), 2023.

[12] R. Zhao, B. Vogel, and T. Ahmed, "Adaptive loss scaling for mixed precision training," *arXiv:1910.12385*, 2019.

[13] A. W. Fitzgibbon and S. Felix, "On stochastic rounding with few random bits," in *32nd Symp. on Comput. Arithmetic, ARITH 2025*, pp. 133–140, IEEE, 2025.

[14] Imandra, Inc., "ImandraX documentation: The IML Language." https://imandrax.dev/docs/language (accessed 2026-05-17).

[15] C. M. Wintersteiger, "Formal verification of the IEEE P3109 standard," 2025. https://github.com/imandra-ai/ieee-p3109.

[16] C. M. Wintersteiger, "Formal verification of the IEEE P3109 standard for binary floating-point formats for machine learning," in *32nd Symp. on Comput. Arithmetic, ARITH 2025*, IEEE, 2025.

[17] T.-C. Chang, S. Park, J. P. Lim, and S. Nagarakatte, "FLoPS: Semantics, operations, and properties of P3109 floating-point representations in Lean," *arXiv:2602.15965*, 2026.