Posits: the good, the bad and the ugly

Florent de Dinechin, Luc Forget, Jean-Michel Muller, Yohann Uguen

Posits: the good, the bad and the ugly

Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI
Lyon, France
Florent.de-Dinechin@insa-lyon.fr

Luc Forget
Univ Lyon, INSA Lyon, Inria, CITI
Lyon, France
Luc.Forget@insa-lyon.fr

Jean-Michel Muller
Univ Lyon, CNRS, ENS-Lyon, UCBL, Inria, LIP
Lyon, France
Jean-Michel.Muller@ens-lyon.fr

Yohann Uguen
Univ Lyon, INSA Lyon, Inria, CITI
Lyon, France
Yohann.Uguen@insa-lyon.fr

ABSTRACT

Many properties of the IEEE-754 floating-point number system are taken for granted in modern computers and are deeply embedded in compilers and low-level software routines such as elementary functions or BLAS. This article reviews such properties on the posit number system. Some are still true. Some are no longer true, but sensible work-arounds are possible, and even represent exciting challenges for the community. Some represent a danger if posits are to replace floating point completely. This study helps framing where posits are better than floating-point, where they are worse, what is the cost of posit hardware, and what tools are missing in the posit landscape. For general-purpose computing, using posits as a storage format could be a way to reap their benefits without losing those of classical floating-point.

CCS CONCEPTS

• **General and reference** → **Cross-computing tools and techniques**; • **Hardware** → *Emerging technologies*; • **Mathematics of computing** → *Mathematical software*.

KEYWORDS

Posits, floating-point, numerical analysis

ACM Reference Format:
Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. 2019. Posits: the good, the bad and the ugly. In *Proceedings of Conference on Next Generation Arithmetic (CoNGA 2019).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Efficient low-level floating-point code, such as high-performance elementary functions [20, 23], uses all sorts of tricks [1] accumulated over the years. These tricks address *performance*, but also and mostly *accuracy* issues: they often exploit a class of floating-point operations which are exact, such as multiplications by powers of two, Sterbenz subtractions, or multiplications of small-mantissa number [24]. Some of these tricks will be detailed in the course of this article. Based on them, more complex tricks can be designed. Examples for elementary functions include exact or accurate range reduction techniques [23] (e.g. Cody and Waite [5], or Payne and Hanek [25]). Examples for more generic codes include doubled precision and more generally floating-point expansions, which have become a common basic block in accuracy-extending techniques in linear algebra [9, 16, 28, 29] and others [6, 7, 18, 20].

Section 2 argues on a concrete example why posit developers of low-level code will need to develop a similar bag of tricks. The quire, a Kulisch-like long accumulator [17] that is part of the posit standard [11], enables new tricks, and could make some floating-point tricks redundant if the quire offers the same service at a lower cost. Otherwise (e.g. when the quire is not supported in hardware, or when it has a performance overhead) a good starting point is to evaluate the floating-point tricks, and see if they can be transposed to posits. Section 3 shows that most addition-related tricks work with posits almost as well as with floats. Section 4 shows that multiplication-related tricks don't.

However, floating-point computing is not just using tricks: actually, most programmers ignore them. They simply rely on operations and toolboxes that have been designed using a body of numerical analysis techniques, similarly accumulated over the years [13]. Posit will eventually need the equivalent. However, it will have to be rebuilt from scratch, because current numerical analysis is built upon a property of floating-point (the constant relative error of operations whatever the inputs) that is lost with posits. This is also discussed in Section 4.

Section 5 then attempts to balance existing posit literature with a review of applicative situations where posits can be expected to behave worse than floats, also discussing remedies.

Finally, section 6 attempt to quantitatively compare the hardware cost of floats, posits and the quire. It also suggests that posits could be first implemented as a storage format only, complementing (not replacing) the classical floating-point stack.

Section 7 concludes by listing some of the developments that are needed to assist developers of numerical software based on posits.

1.1 Notations and conventions

This article assumes basic knowledge of the posit format [12], and refers to the draft posit standard [11] for a detailed description of the format itself. It also assumes basic knowledge of the IEEE-754-2008 floating-point standard [14]. The 16-bit, 32-bit, and 64-bit

CoNGA 2019, March 2019, Singapore

formats defined by this standard are respectively noted FP16, FP32 and FP64.

For posits, we use the following notations, matching the Python `sfpy` package[1] which is a reference implementation of the draft posit standard[11].

- Posit8 is 8 bits with es=0;
- Posit16 is 16 bits with es=1;
- Posit32 is 32 bits with es=2.
- Posit64 is 64 bits with es=3.
- PositN represents any of the previous
- PositN(i) with $i \in \mathbb{N}$ denotes the positN number whose encoding is the integer i.

We denote the posit operations for addition, subtraction, and multiplication as $\oplus$, $\ominus$ and $\otimes$.

1.2 Casting posits to floats

The following lemma is quite useful when experimenting with posits:

Lemma 1.1 (Exact cast to FP64). *Any Posit8, Posit16 or Posit32 except NaR is exactly representable as an IEEE-754 FP64 (double precision) number.*

Proof. Table 2 shows that for these sizes, the exponent range as well as the mantissa range of the posits are included in the corresponding ranges of FP64. Therefore, any posit exponent and any posit mantissa fraction will be converted exactly to an FP64 exponent and mantissa. □

2 MOTIVATION

The first motivation of this work was the implementation of elementary functions such as exponential, logarithms, etc. in a posit-based system.

For Posit8 and Posit16, most functions are probably best implemented by clever tabulation (clever meaning that the tables can be limited in size thanks to symmetries of the function and other redundancies).

For larger precisions, the function has to be evaluated. Since posits are a kind of floating-point representation, algorithms designed for floating-point are a good starting point. Let us take the example of the exponential function. The skeleton of a typical algorithm to evaluate in floating-point the exponential of a number X is as follows.

(1) Compute the integer $E \approx \lfloor X/\log(2) \rceil$ which will be the tentative exponent.
(2) Compute the float $Y \approx X - E \times \log(2)$, a reduced argument in the interval $I \approx [\frac{-\log(2)}{2}, \frac{\log(2)}{2}]$. Remark that $Y \approx X - E \times \log(2)$ can be rewritten $e^X \approx 2^E e^Y$.
(3) Evaluate a polynomial $Z \approx P(Y)$ where the polynomial P is a good approximation of e^Y on I
(4) Construct a float of value 2^E, and multiply it by Z to obtain the result $R \approx e^X$.

Actual implementations may use a more elaborate, table-based, range reduction [31] but we can ignore this for the sake of our motivation.

[1] https://pypi.org/project/sfpy/

The posit standard wisely mandates correct rounding of the result. Because of the Table Maker's Dilemma [23], this sometimes requires to perform the intermediate computations with an accuracy that is almost three times the target precision (about 150 bits for FP64). This computation will be expensive, therefore the current technique is to first evaluate the function (using the previous algorithm) with an accuracy just slightly larger than the format precision (7 to 10 bits more accurate). Most of the times, rounding this intermediate result is the correct rounding of e^X. When it is not, we can detect it and launch an expensive accurate computation (same algorithm, but with larger-precision data and a higher-degree polynomial). The idea is that this expensive computation will be rare enough so that its average cost remains small.

Still, both steps need to compute with a precision larger than the floats themselves: a few bits larger for the first tentative step, three times larger for the rare accurate step. Extended-precision data is manipulated as the unevaluated sum of two floats in the first step, as the sum of three floats in the accurate step. Note that since log(2) is an irrational number, Y needs to be computed in the extended precision.

All this translates fairly well to posits. The quire can be used to evaluate (2) very simply and accurately (representing log(2) to the required accuracy as a sum of posits). However, Y must then be *multiplied* several times in the polynomial evaluation, all with extended precision. You cannot multiply the quire, therefore Y must be extracted from the quire. For the first step, it will be extracted as a sum of two posits (first round the quire to a posit, then subtract this posit from the quire, then round the result to a second posit).

But rounding the quire to a posit is an expensive operation, due to the size of the quire itself. This will be quantified in Section 6. This cost is easily amortized for the large dot products of linear algebra. Here, since we will need to extract Y as a sum of two posits, we tend to believe that it will be much more efficient to avoid the quire altogether and compute it directly as a sum of two posits, using an adaptation to posits of the Cody and Waite technique [5, 23]. This motivates the techniques studied in Section 3 and 4.

3 THE GOOD: SUMMATIONS, ROUNDING PROPERTIES AND DOUBLED PRECISION

3.1 Accurate summations without a quire

A fair rule of thumb is the following: *in a summation of smaller terms yielding a result with an exponent close to zero, posits are expected to be more accurate than floats of the same size.*

In such situations, even when the smallest summands suffer loss of precision due to tapered arithmetic, the bits thus lost would have been shifted out of the mantissa anyway in the summation process. In other words, in these situations, trading off the accuracy of these smaller terms for accuracy of the final sum is a win. IEEE-754 floating point, in this case, computes in vain bits that are going to be shifted out.

The condition that exponent should be close to zero is a relatively soft one: for instance, Posit32 has more fraction bits than IEEE FP32 for numbers whose magnitude ranges from 10^{-6} to 10^{6}. The "golden zone" thus defined (and highlighted on Figure 1) is fairly comfortable to live with, and may easily hide the fact that out of this zone, posits become less accurate than floats of the same size.

This rule of thumb explains for instance why posits are a very good choice for current machine learning applications: there, convolutions and other neuron summations are scaled at each level by activation functions, and the number of terms summed in each level is small enough for the sum to remain in the golden zone.

This property could be considered useless, since the summation could be performed in the quire anyway (and must, as soon as the result can not be predicted to lie in the golden zone). Again, it is a matter of cost. Although an exact accumulator has been shown to be a cost-effective way of implementing an IEEE-compliant FP16 format [3], for larger formats hardware quire support is not expected to be cheap, as section 6 will show. Indeed, at the time of writing, none of the leading posit hardware developments supports the quire [4, 27].

Let us now depart from this "macro" view and study useful properties of one posit operation in isolation.

3.2 Is the rounding error in the addition of two posits a posit?

The rounding error of the addition of two IEEE-754 floating-point numbers of the same format can itself be represented as a floating-point number of the same format. This is an important property (and the main motivation of subnormal numbers), for at least two reasons. 1/ it enables many other properties, and 2/ it has the following corollary which programmers will assume and compilers can exploit:

$a \ominus b == 0$ is equivalent to $a = b$ (except when a or b is NaN)

Here are the results of two exhaustive tests using `sfpy`. Lack of space prevents giving the test programs here, but they are straightforward, based on Lemma 1.1. A slightly more subtle test program will be shown in Section 3.4.

Lemma 3.1. *The rounding error in the addition of two Posit8 is always a Posit8.*

Lemma 3.2. *The rounding error in the addition of two Posit16 is a Posit16 except for the unique case of Posit16(0x0001) + Posit16(0x0001).*

The generalization of this observation is the following:

Conjecture 3.3. *The rounding error in the addition of two posits of same format is a posit of the same format, except in the "twilight zone" where geometric rounding is used instead of arithmetic rounding.*

In Posit32, we found only two situations where the property does not hold: $x \ominus x$ when x is Posit32(0x00000003), or when x is Posit32(0x7ffffffd). Let us detail the first (the second is similar). The value of x is 2^{-114}, therefore the exact sum $x+x$ is 2^{-113}. The two neighbouring Posit32 values are Posit32(3) itself and Posit32(4)=2^{-112}. As we are in the twilight zone, we have to round the exponent to the nearest. It is a tie, which is resolved by rounding to the even encoding, which is Posit32(4). Now the rounding error is $2^{-112} - 2^{-113} = 2^{-113}$, which is not a Posit32.

Remark that if rounding was simply specified as "round to the nearest Posit", the result would have been 2^{-114}, as 2^{-113} is much closer to 2^{-114} than to 2^{-112}. The rounding error would have been 2^{-114}, which is a Posit32. It general, one could argue that it would make more sense to resolve exponent ties by rounding to the nearest posit. The current motivation for this choice, according to one of our reviewers, is that the hardware cost would be higher.

A proof of this conjecture in the general case remains to be written.

3.3 Posit variant of the Sterbenz Lemma

The following lemma has been exhaustively tested for Posit8 and Posit16, and is proven here in the general case:

Lemma 3.4. *For any two PositN of the same format a and b, where a and b are different from NaR,*

$$\frac{a}{2} \leq b \leq 2a \implies a \ominus b = a - b \quad .$$

The proof in the general case is based on the following lemma :

Lemma 3.5. *For a given PositN format, if the fraction field length of 2^l is $p \geq 1$ bits, then for all d in $[\![0, p]\!]$, the fraction field length of 2^{l-d} is at least $p - d$.*

Proof. By induction over d:

If $d = 0$, then the property is true.

Now, assuming that the property holds for a given $d < p$, we can encode 2^{l-d} with at least $p - d$ significand field bits.

If the exponent field of the posit encoding of 2^{l-d} is different from zero, then decrementing it will express 2^{l-d-1} with the same significand length $p - d \geq p - d - 1$.

If it is equal to zero, then we need to decrement the regime and the exponent field becomes $2^{es} - 1$. As $p - d \geq 1$, even in the case when decrementing the range increases its run length, es bit still remains so it is possible to set the exponent to its maximum value. In the worst case, the run length of the range needs to be incremented by one, so the significand field length is reduced by at most 1.

The property therefore holds for $d + 1$. □

Proof of Lemma 3.4. For the rest of the proof, $\mathbb{P}_{w,es}$ will denote the set of posits of width w and exponent size es, and a and b will be two elements of $\mathbb{P}_{w,es}$ such that

$$\frac{a}{2} \leq b \leq 2 \times a$$

As $\mathbb{P}_{w,es}$ is closed under negation, the proof can be restricted without loss of generality to the case when $0 \leq b \leq a$. In this case the following relation holds :

$$\frac{a}{2} \leq b \leq a$$

On the limit cases, $a - b$ is equal to either b or zero, both of which are posits so the relation can be further restricted to $\frac{a}{2} < b < a$, which also implies that b has at least one significand bit.

Noting $\mathrm{ulp}(\cdot)$ the function that associates to a posit the weight of the last bit of its significand, and $\mathrm{ufp}(\cdot)$ the function that associates the weight of the first bit of the significand, we have $\mathrm{ulp}(a) \geq \mathrm{ulp}(b)$. The equality $\mathrm{ulp}(a) = \mathrm{ulp}(b)$ occurs either when $\mathrm{ufp}(a) = \mathrm{ufp}(b)$, or when $\mathrm{ufp}(a) = 2\mathrm{ufp}(b)$ and a has one extra bit of precision. We have $a - b = k_a \times \mathrm{ulp}(b) - k_b \times \mathrm{ulp}(b) = (k_a - k_b) \times \mathrm{ulp}(b) = k \times \mathrm{ulp}(b)$, with $k < k_b$ as $b > \frac{a}{2}$. Therefore k can be written with the precision of b, possibly with leading zeros. Thanks to Lemma 3.5, $a - b$ can be represented as a posit. □

3.4 TwoSum works in posit arithmetic

In the following, TwoSum denotes the algorithm introduced by Møller in [22]. This algorithm inputs two floats a and b. It returns two floats s and r such that $s = RN(a + b)$ and $a + b = s + r$ exactly under the condition that $a + b$ does not overflow (underflow to subnormals is not a problem [24]). Here, RN denotes the standard IEEE-754 round to nearest even.

```
1 def TwoSum(a, b):
2     s = a + b
3     aapprox = s - b
4     bapprox = s - aapprox
5     da = a - aapprox
6     db = b - bapprox
7     t = da + db
8     return s, t
```

The following lemma states that TwoSum almost works in posits, at least on the two smallest formats.

Lemma 3.6. *If neither a nor b belongs to the twilight zone, the posits* `s` *and* `t` *computed by the TwoSum sequence of posit operations verifies* `s+t=a+b` *exactly.*

Proof. The proof is by exhaustive test on Posit8 and Posit16. This test deserves to be detailed. For clarity, we call FloatTwoSum the TwoSum algorithm using only IEEE-754 numbers, and PositTwoSum the same algorithm using only posit numbers

In order to check that PositTwoSum gives the valid output s, t for a given input a, b, i.e. $s = a \oplus b$ and $s + t = a + b$, only the second equality has to be verified, as the first one is true by construction of s.

The following algorithm (in Python syntax with `sfpy`) inputs two posits (on 8, 16 or 32 bits) `a` and `b`, and returns True if TwoSum returns their exact sum.

```
1 def CheckTwoSumExact(a, b):
2     fa, fb = float(a), float(b)
3     fs1, ft1 = FloatTwoSum(fa, fb)
4     ps, pt = PositTwoSum(a, b)
5     fps, fpt = float(ps), float(pt)
6     fs2, ft2 = FloatTwoSum(fps, fpt)
7     return fs1 == fs2 and ft1 == ft2
```

In this algorithm, `float(`p`)` returns the IEEE-754 FP64 format number having the same value than the posit p, thanks to Lemma 1.1.

With Lemma 1.1, and using the known property of the FloatTwoSum algorithm, we have in `(fs1,ft1)` a canonical representation of the exact sum `a+b`. Then PositTwoSum, line 4, computes in `(ps, pt)` the PositTwoSum of the two input posits. Due to different number formats, usually `(ps,pt)`$\neq$`(fs,ft)`. However, lines 5 and 6 ensure that `fs2+ft2=ps+pt`. Therefore, if the value returned by line 7 is true, then `ps+pt=a+b`. □

It now remains to attempt to prove this property in the general case of arbitrary posit systems.

3.5 There is a FastTwoSum that works

The FastTwoSum algorithm below [24] compute the same decomposition as TwoSum, but needs fewer operations, at the cost of an additional constraint on the inputs: When applied on IEEE numbers, it requires the first operand's exponent to be greater or equal to the second's. This is typically enforced by checking that the first operand has a greater magnitude than the second.

```
1 def FastTwoSum(a, b):
2     s = a + b
3     bapprox = s - a
4     t = b - bapprox
5     return s, t
```

Exhaustive tests on Posit8 and Posit16 gives the following results

Lemma 3.7. *If neither a nor b belongs to the twilight zone, and* $|a| > |b|$ *the posits* `s` *and* `t` *computed by the FastTwoSum sequence of posit operations verifies* `s+t=a+b` *exactly.*

On posits, the inequality needs to be strict for the result to hold, as shown by the following counter-example. When computing the FastTwoSum sequence of operation over Posit8(0x5F), which represents the binary number 1.11111, with itself, the addition of the two outputs is no longer equal to the exact sum. With no limit on the width, the binary encoding of the sum would have been 011011111 which is rounded to 01110000. Then the difference between the rounded sum and the operand is 10.00001, which would normally be encoded as 0110000001, rounded to 01100000 (10.000). Finally, the difference gives -0.00001, which is exactly encoded and differs from the exact error which is -0.0001.

4 THE BAD: THE LOSS OF EXACT PRODUCTS, TWOMULT, AND THE STANDARD MODEL

4.1 Multiplication by a power of two is not always exact

This property is also true in floating-point in the absence of overflow and underflow. It is no longer true in posits, as the significand of the result may lack the bits to represent the exact result.

Here are a few extreme examples of inexact multiplications by 2 or 0.5 in Posit8 (similar example can be found in Posit16 and Posit32)

- 1.03125 * 2.0 returns 2.0
- 10.0 * 2.0 returns 16.0
- 64.0 * 2.0 returns 64.0
- 0.015625 * 0.5 returns 0.015625
- 0.984375 * 0.5 returns 0.5

The extreme case of wrong multiplication by a power of two is (still in Posit8) 64.0*64.0 which returns 64.0. However this is a consequence of the current choice of saturated arithmetic in posits (which could be disputed, but it is not the object of the present article). In this kind of situation, floating-point will overflow, so the result will not be exact either.

We still have in posits the following lemma:

Lemma 4.1 (Exact multiplications by a power of two). *If* $2^k x$ *is closer to* 1 *than* x *is, then the PositN product* $x \otimes 2^k$ *is exact.*

Proof. The result has at least as many significand bits as x itself, and the condition also takes the result away from the twilight zone. □

Unfortunately, in the algorithm sketched in Section 2 for the exponential function, we need the opposite: the argument is reduced to a small value whose exponential is around 1, then scaled back to a larger value by a multiplication by a power of two. This multiplication was exact with floats, while with posits it will introduce an error term. This makes a correctly rounded implementation more challenging, but of course not impossible.

4.2 The rounding error in the product of two posits is not always a posit

This property is true for floating-point numbers of the same format. This enables the very useful TwoMult operation, which writes the product of two floating-point numbers as a sum of two floating-point numbers [24]. TwoMult is the core of the doubled-precision multiplications needed in the polynomial evaluation in our exponential example.

This property is **not** true on the posits, there is therefore no hope of designing the equivalent of TwoMult as we did for TwoSum.

Here are but a few counter-examples in Posit8.

- 3.75 * 12.0 = 32.0 + 13.0
- 3.75 * 14.0 = 64.0 + -11.5

and neither 13.0 nor -11.5 are representable as Posit8.

It should be noted that the situation where the product of two posits cannot be represented as the sum of two posits is the majority of the case: exhaustive test shows that out of 65535 possible non-NaR products in Posit8, only 21759 have their error representable as a Posit8.

To be fair, among these, there are about 3000 cases where the product saturates, for instance (still in Posit8)

- 3.75 * 24.0 = 64.0 + 26.0
- 3.75 * 32.0 = 64.0 + 56.0

These correspond to cases where IEEE-754 floating-point would overflow, hence out of the domain where the product of two floats can be represented as a sum of two floats.

Still, the cases where the product of two Posit8 cannot be represented exactly as the sum of two Posit8 remains the majority (about 70%), and this ratio is similar for Posit16 (about 65%).

4.3 Numerical analysis from a blank sheet

A very useful feature of standard floating-point arithmetic is that, barring underflow/overflow, the relative error due to rounding (and therefore the relative error of all correctly-rounded functions, including the arithmetic operations and the square root) is bounded by a small value $u = 2^{-p}$, where p is the precision of the format. Almost all of numerical error analysis (see for instance [13]) is based on this very useful property.

This is no longer true with posits. More precisely, in the PositN format, the relative error due to rounding a positive number $x \in [2^k, 2^{k+1})$ that is not in the twilight zone, is 2^{-p}, where p is no longer a constant, since it satisfies:

$$p = N - \rho - \text{es},$$

with

$$\rho = \begin{cases} \left\lfloor \frac{k}{2^{\text{es}}} \right\rfloor + 1 & \text{if} \quad x < 1, \\ \left\lfloor \frac{k}{2^{\text{es}}} \right\rfloor + 2 & \text{if} \quad x \geq 1. \end{cases}$$

Figure 1 illustrates this.

Numerical analysis has to be rebuilt from scratch out of these formula.

One reviewer argued that the constant error property is not true for floating-point, either, since the relative error degrades in the subnormal range (the left green slopes on Fig. 1). Indeed, numerical analysis theorems are only true provided no intermediate computation overflows or underflows. For instance, with FP32 numbers, the constant relative error bound of 2^{-24} is only valid if the absolute value of the result lies the interval $[2^{-126}, (2 - 2^{-23}) \cdot 2^{127}] \approx [1.17 \cdot 10^{-38}, 3.4 \cdot 10^{38}]$ for the 4 basic operations[2]. Some numerical analysis results therefore only hold if no intermediate value violates these bounds.

As the reviewer argued, these same results also hold with posits, albeit on a smaller interval. For instance, for Posit32, the same bound 2^{-24} is guaranteed for operations whose absolute result lies within $[2^{-20}, (2-2^{-23}) \cdot 2^{19}) \approx [10^{-6}, 10^{6}]$. This is the "golden zone" (in yellow on Fig. 1) where posits are at least as accurate as floats.

The thresholds 2^{-20} and 2^{20} are not out-of-the-ordinary numbers: they can quickly appear in fairly simple calculations involving initial values close to 1. As Fig. 1 shows, the range can be enlarged but then the accuracy bound drops. So this approach of naively applying existing numerical analysis to posits doesn't do them justice. It is very different to have a property that holds everywhere except in the corners, and to have a property that holds only in a small range. Posits have variable accuracy, and many experiments show that it can translate to more accurate computation: this gives some hope that numerical analysis can be rebuilt on the premise of variable accuracy.

A second good reason why numerical analysis has to be rebuilt from scratch is that posits include the quire as standard. Properly exploiting the quire will remove a lot of the error terms that were considered in classical error analysis. However, the accuracy of the quire will be lost as soon as it is converted to a posit out of the golden zone. In any posit-supporting computer, the quire-to-posit conversion instructions will probably include a scaling by a power of two to avoid accuracy loss in this case. This is the third reason why numerical analysis, which mostly ignores scaling, has to be rebuilt from scratch.

Still, whether a relevant numerical analysis can be built for posits remains an open question and a challenge for the posit community.

5 THE UGLY: VERY LARGE, VERY SMALL, AND MULTIPLICATIVE CANCELLATION

The posit literature provides many examples where computing with posits turns out to be more accurate than computing with floats of the same size. Some works compare floating point and posits on complete applications [19]. In [12], the authors use two examples (roots of $ax^2 + bx + c$, and area of a very flat triangle) classically used

[2]For additions and subtractions, the relative error bound 2^{-24} even holds for the full floating-point range, since any addition that returns a zero or a subnormal is exact, or with a relative error of 0.

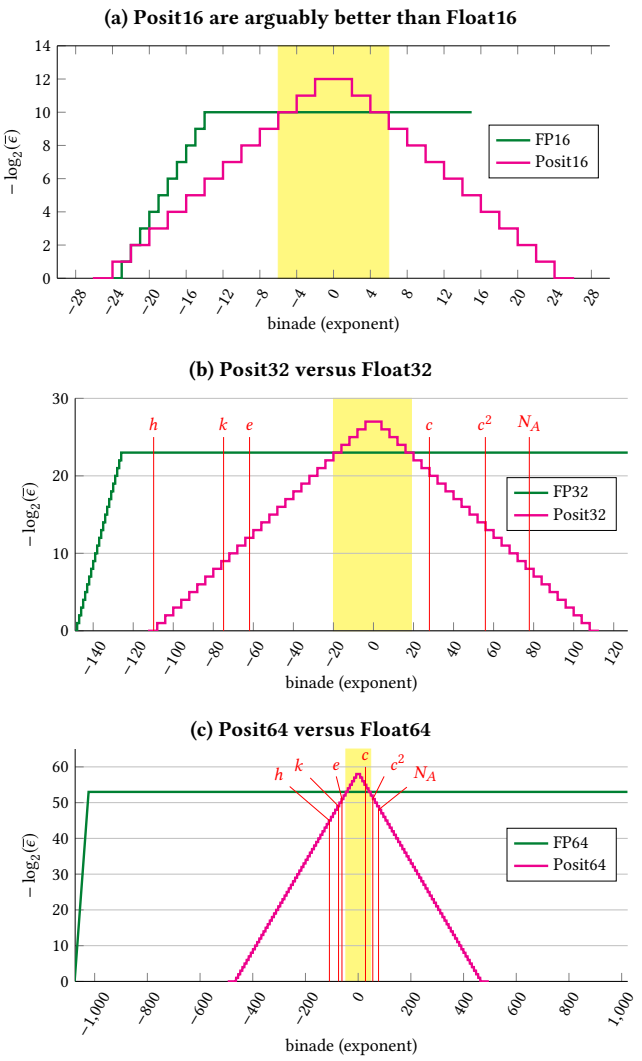

Figure 1: Epsilons to use for the standard model

by floating-point experts to outline the limits of a floating-point system. They show that posits behave better on these examples. Unfortunately they stop there, when a balanced study would require to similarly construct examples that outline the limits of posits. This section first fills this gap, with the purpose to help users understand in which situations posits can be expected to behave much worse than floats of the same size, and more importantly, why.

5.1 Two anti-posit examples

The first example, inspired from [10], is to consider the straightforward calculation of

$$\frac{x^n}{n!},$$

done in the naive way (separate computation of x^n and $n!$ then division). For $x = 7$ and $n = 20$, with FP32 arithmetic, we obtain a relative error $1.73 \cdot 10^{-7}$, and with Posit32 arithmetic, the relative error is $1.06 \cdot 10^{-4}$. For $x = 25$ and $n = 30$, with FP32 arithmetic, we obtain a relative error $1.50 \cdot 10^{-7}$, and with posit32 arithmetic, the relative error is $8.03 \cdot 10^{-2}$.

Of course, there exists a re-parenthesising of $\frac{x^n}{n!}$ that ensures the calculation stays in the golden zone, just like there exists well-known alternative formula that make floating-point accurate for the examples cited in [12]. We will discuss in the conclusion if we can expect compilers to find them automatically.

The second example, less academic, deals with physics. As of May 2019, the International System of Units[3] will be defined purely out of physics laws and a handful of constants, some of which are given Table 1 and plotted on Fig. 1. These constants are all defined with at most 10 decimal[4] digits.

Posit32 is unable to represent most of these constants accurately. Many of them are even out of the golden zone of Posit64 (although to be fair, the Posit64 representation always has more than 10 significant digits and therefore should be good enough).

Can we apply scaling here? This would in principle be as simple as changing the units (we have used in Table 1 the standard units). Physicists themselves sometimes use scaled units, such as electron-volt (eV) instead of Joule for energy. However, it adds to code complexity, and for safety it would need tight language support that is currently not mainstream. To complicate matters, posit-oriented scaling should be by powers of two (with the caveat that such scaling may not be exact, see Section 4.1), while unit scaling (mega, micro, etc) are powers of ten, with negative powers of ten not exactly represented in binary. Worse, if the code doesn't use the standard units, it then becomes critical to attach the unit to each data exchanged (let us recall the Mars Climate Orbiter failure, which was due to a lack of doing so). We can conjecture that attaching units would void the goal of posits to pack more digits into fewer bits.

[3] See https://en.wikipedia.org/wiki/International_System_of_Units and the references therein

[4] It seems the arithmetic community missed an opportunity to push for binary-representable values...

Planck constant h		$6.626070150 \cdot 10^{-34}$
Posit32 value	$\approx$	$7.7 \cdot 10^{-34}$
FP32 value	$\approx$	$6.626070179 \cdot 10^{-34}$
Avogadro number N_A		$6.02214076 \cdot 10^{23}$
Posit32 value	$\approx$	$6.021 \cdot 10^{23}$
FP32 value	$\approx$	$6.0221406 \cdot 10^{23}$
Speed of light c		299792458
Posit32 value		299792384
FP32 value		299792448
charge of e^-		$1.602176634 \cdot 10^{-19}$
Posit32 value	$\approx$	$1.6022 \cdot 10^{-19}$
FP32 value	$\approx$	$1.60217659 \cdot 10^{-19}$
Boltzmann constant k		$1.380649 \cdot 10^{-23}$
Posit32 value	$\approx$	$1.3803 \cdot 10^{-23}$
FP32 value	$\approx$	$1.1.38064905 \cdot 10^{-23}$

Table 1: Constants defining the standard international units

5.2 Multiplicative cancellation

What is more, high-energy physics code routinely manipulates numbers of very large or very small magnitudes, and multiplies them together. For instance, the main use of the Planck constant is the equation $e = h\nu$. Any naive use of this formula will be an instance of what we call multiplicative cancellation. The computed energy e will have no meaningful digit, even if ν is large enough (there are X rays in the petaherz range) to bring e in the golden zone.

Another example is the most famous physics equations: $e = mc^2$. The value of c^2 will be computed in Posit32 with a relative error of $3.29 \cdot 10^{-5}$ while the FP32 error is $6.67 \cdot 10^{-8}$. If we multiply by the (very small) mass of a particle (say, a proton with $m \approx 1.672 \cdot 10^{-24}$ g), the resulting energy could fall in the golden zone, while having no more accuracy than its two factors had.

A reviewer mentioned that c^2 could be exactly represented as a sum of three Posit32 so that mc^2 could be computed accurately as a sum of 3 products in the quire. We welcome this suggestion, a perfect example of the kind of tricks that we are studying. However, it does not make posits more efficient than floats on this example. Besides, to apply this kind of trick, one first must be aware of this situation. This is the main objective of this section.

Of course, posits also inherit the additive cancellation of floating-point. An addition that cancels is for instance $x - 3.1416$ in the case $x = 3.1421$. The computed result will be $5.0000 \cdot 10^{-4}$.

Cancellation is a dual situation: on one side, it is a Good Thing, because the operation itself is exact (it is a case of Sterbenz Lemma). In our example subtraction, there was no rounding error. However, and it is a Very Bad Thing, an arbitrary number digits of the result are meaningless: in our example, the lower four zero digits of the significand correspond to information absent from the original data. If what the programmer intended to compute was $x - \pi$, this result has a relative accuracy of 10^{-1} only.

In posits, we have a very similar situation when multiplying a very large number by a very small one. It may happen that the operation itself entails no rounding error (as soon as we have more significand bits in the result as the sum of the lengths of the input significands), while having most of these bits meaningless (even the non-zero ones).

A multiplicative cancellation will be less obvious to detect than additive one, since the meaningless digits are not necessary zero. Besides, not all multiplicative cancellations are exact.

Here, the challenge is the same as for additive cancellation: on the one side, there is a class of exact computations, and we want to force them in our programs to design more accurate computations, just like we often use Sterbenz lemma on purpose in elementary function code. On the other side, we need to either educate programmers to avoid writing code that depends on meaningless digits, or provide them with tools that protect them.

6 HARDWARE CONSIDERATIONS

This section complements exisiting work comparing hardware implementations of floats and posits [4, 27] with a high-level point of view that focuses on architectural parameters and includes a discussion of the quire.

	range	quire parameters			custom FP
	(min, max)	w_q	w_l	w_r	(w_E, w_F)
Posit8, es=0	$[2^{-6}, 2^{6}]$	32	16	12	(4, 5)
Posit16, es=1	$[2^{-28}, 2^{28}]$	128	64	72	(6, 12)
Posit32, es=2	$[2^{-120}, 2^{120}]$	512	256	332	(8, 27)
Posit64, es=3	$[2^{-496}, 2^{496}]$	2048	1024	1429	(10, 58)

Table 2: Architectural parameters of PositN formats

6.1 Building standalone posit operators

Posits are a form of floating-point numbers whose significand fraction can take a maximum size w_f and whose exponent value can fit on w_e bits (whatever the encoding of these fractions and significand, e.g. sign/magnitude, two's complement or biased). Table 2 reports for the standard PositN formats the values of w_e and w_f, given by the following formulae:

$$w_E = \lceil log2(N-1) \rceil + 1 + es \qquad w_F = N - 1 - 2 - es \quad .$$

One may therefore build a hardware posit operator by combining a Posit-to-FP decoder, an FP operator built for non-standard values of w_F, and an FP-to-Posit converter. The FP operation needs to be non-standard in several ways, each of which actually is a simplification of a standard IEEE 754 FP operation:

- its rounding logic has to be removed (rounding has to be performed only once in the FP-to-Posit converter); The output on Figure 2 labelled "exact result unrounded" should be understood as "quotient and remainder" and "square root and remainder" in division and square root operators. The rounding information can sometimes be compressed in a few bits, e.g. "guard, round and sticky" in addition.
- it may use encodings of the exponent and significand that are simpler (e.g. two's complement for the exponent instead of standard biased format, two's complement for the significand instead of sign/magnitude) or more efficient (e.g. redundant form for the significand);
- it doesn't need to manage subnormals, as the chosen intermediate formats can encode any posit as normalized significand;

Of course, the FP operation itself eventually reduces to fix-point operations.

With these simplifications, this three-block implementation embodies the unavoidable amount of work in a posit operation, and leading hardware posit implementations [4, 27] follow it. Compared to IEEE floats, it trades the overhead of subnormal and other special cases for the overhead of the LDCs (Leading Digit Counters) and shifts in the conversion units. Posits also pay their better best-case accuracy with a slightly larger internal format (see Table 2), which has a measurable impact on the multiplier. Altogether, a quantitative comparison of posits and floats with similar effort [4, Table IV] shows that posit and floats have comparable area and latency.

In both cases the latency can be reduced (but at increased hardware cost) by hardware speculation [2], or the equivalent of classical dual-path FP adder architectures.

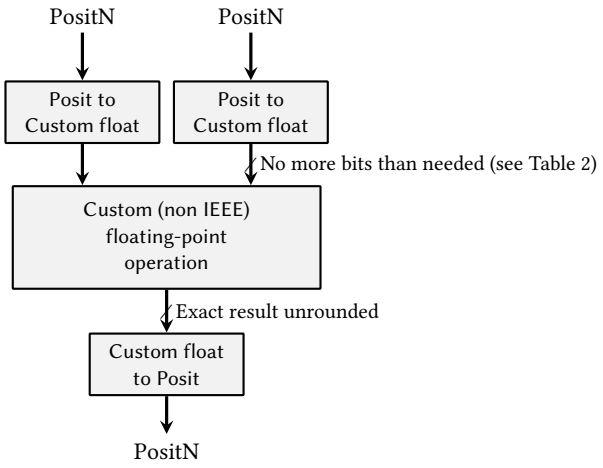

Figure 2: Architecture of a standalone posit operator

6.2 The quire

The idea of an exact accumulator currently has a lot of momentum. Several existing machine learning accelerators [3, 15] already use variations of the exact accumulator to compute on IEEE-754 16-bit floating point. Other application-specific uses have been suggested [8, 33]. For larger sizes, this could be a useful instance of "dark silicon" [32]. This trend was also anticipated with the reduction operators in the IEEE 754-2008 standard, unfortunately without the requirement of exactness.

The quire is not yet completely specified in the posit standard. Currently, the standard specifies a binary interchange format, whose size w_q is given in Table 2. It also defines fused operations as follows: *Fused operations are those expressible as sums and differences of the exact product of two posits; no other fused operations are allowed.* This formulation allows for useful fused operations such as FMA, complex multiplication [30], or even full convolutions for neural networks, to be implemented independently of the quire. In the sequel, however, we discuss the cost of hardware support for a quire with the range of the interchange format.

There are several techniques to implement the quire [17], some of which allow for pipelined accumulations of products with one-cycle latency at arbitrary frequency [8].

The main performance challenge is the latency of converting a quire to a posit. It requires leading digit counting and shifting on a number of bits w_l (also given in Table 2 for standard forrmats) which is half the quire size (256 bits for Posit32, 1024 bits for Posit64). Correct rounding also requires a XOR operation on a number of bits w_r of comparable magnitude, also given in Table 2.

In addition to this, the most cost-effective way of achieving 1-cycle accumulation at high frequency is to keep the quire in a redundant format with delayed carries. If such a format is used, its conversion to a non-redundant format (completing carry propagation) will incur additional overhead.

All this translates to an hardware overhead at best proportional to w_l, and to a large latency. In our current implementation targeting FPGAs, summing two products of Posit32 in the hardware quire has more than 10x the area and 8x the latency of summing them using a posit adder and a posit multiplier. Although we do hope to improve these results, such factors should not come as a surprise: the 512 bits of the Posit32 quire are indeed 18x the 27 bits of the Posit32 significand.

For large dot products, we strongly support that the exactness of the result justifies the hardware overhead, while the latency overhead can be amortized.

However, one should not assume that the availability of a hardware quire magically brings in a FMA or a sum of a few products with latencies comparable to those of non-quire operations. For small computations such as the additive range reduction or the polynomial evaluation of Section 2, we expect other accuracy-enhancing techniques to remain competitive.

6.3 Using posits as a storage format only

Table 2 also shows that Posit8 and Posit16 can be cast errorlessly in FP32, while Posit32 can be cast errorlessly in FP64. One viable alternative to posit implementation is therefore to use them only as a memory storage format, and to compute internally on the extended standard format.

This solution has many advantages:

- It offers some of the benefits of the quire (internal extended precision) with more generality (e.g. you can divide in this extended precision), and at a lower cost;
- the better usage of memory bits offered by the posit format is exploited where relevant (to reduce main memory bandwidth);
- The latency overhead of posit decoding is paid only when transferring from/to memory;
- Where it is needed, we still have FP arithmetic and its constant error;
- Well established FP libraries can be mixed and matched with emerging posit developments;

It has one drawback that should not be overlooked: attempting to use this approach to implement standard posit operations may incur wrong results due to double rounding. This remains to be studied.

Note that a posit will never be converted to a subnormal, and that subnormals will all be converted to the smallest posit. This could be an argument for designing FPUs without hardware subnormal support (i.e. flushing to zero or trapping to software on subnormals).

Figures 3 and 4 describe the needed conversion operators. Note that the Posit-to-FP and FP-to-Posit boxes of Figure 2 are almost similar to these, with the FP bias management removed, and additional rounding information input to the Float-to-Posit converter.

This approach has been tested in an High-Level Synthesis context: we only implemented the converters of Figures 3 and 4, and relied on standard floating point operators integrated in Vivado HLS.

Some synthesis results are given for a Kintex 7 FPGA (xc7k160tfbg484-1) using Vivado HLS and Vivado v2016.3 in Tables 3. The conclusion is that posit decoders and encoders remains small compared to the operators being wrapped, all the more as the goal is to wrap complete FP pipelines, i.e. several operations, between posit decoders and encoders. The latency also remains smaller than that of the operators being wrapped.

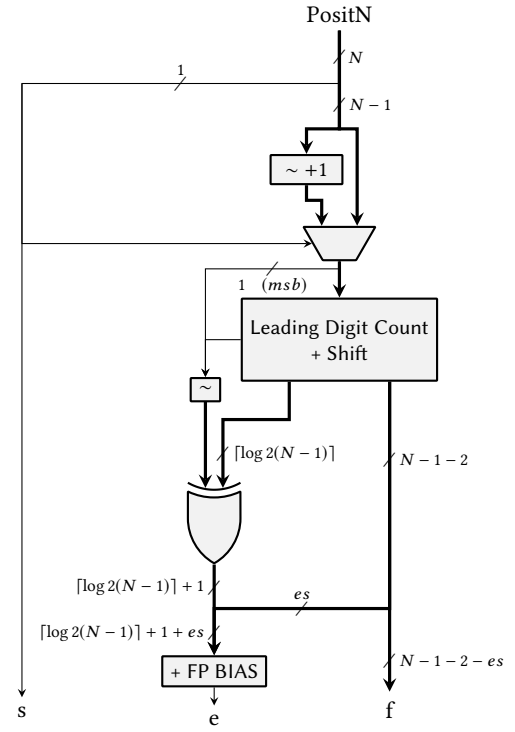

Figure 3: Architecture of a PositN to floating-point decoder

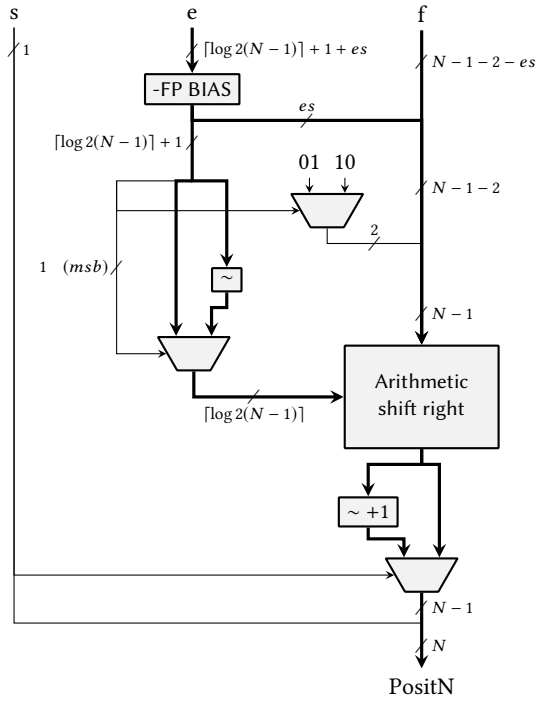

Figure 4: Architecture of a floating-point to PositN encoder

	LUTs	Reg.	DSPs	Cycles @ Freq
Posit8 to FP32	25	28	0	5 @ 546MHz
FP32 to Posit8	20	30	0	3 @ 508MHz
Posit16 to FP32	90	72	0	6 @ 498MHz
FP32 to Posit16	63	56	0	4 @ 742MHz
FP32 ADD	367	618	0	12 @ 511MHz
FP32 MUL	83	193	3	8 @ 504MHz
FP32 DIV	798	1446	0	30 @ 438MHz
FP32 SQRT	458	810	0	28 @ 429MHz
Posit32 to FP64	192	174	0	8 @ 473MHz
FP64 to Posit32	139	106	0	4 @ 498MHz
FP64 ADD	654	1035	3	15 @ 363MHz
FP64 MUL	203	636	11	18 @ 346MHz
FP64 DIV	3283	6167	0	59 @ 367MHz
FP64 SQRT	1771	3353	0	59 @ 388MHz

Table 3: Synthesis results of the posit encoders and decoders, compared with the operators they wrap.

7 CONCLUSION: TOOLS NEEDED

This article was a survey of the strengths and weaknesses of posit numbers, including mathematical properties taken for granted when working with fixed- or floating-point.

There are clear use cases for the posit system. Machine learning, graphics rendering, some Monte Carlo methods, integration-based methods where the magnitude of the result can be framed, and many other applications belong there. There are also clear situations where posits are worse than floating-point. Particle physics simulations are one example, integration methods where the result is unbounded a priori is another one. Between these extremes, writing the equivalent of a BLAS library that is both efficient and accurate whatever the combination of inputs is probably more of a challenge with posits than with floats.

When posits are better than floats of the same size, they provide one or two extra digits of accuracy [19]. When they are worse than floats, the degradation of accuracy can be arbitrarily large. This simple observation should prevent us from rushing to replace all the floats with posits.

Can we expect compilers to protect programmers from situations such as multiplicative cancellation? A comment from a reviewer was: *It may turn out that a naive user can do even more damage with naive use of posits than with naive use of floats, but it's pretty easy for a compiler to recognize such naive uses and automatically correct them.* We agree to some extent, all the more as this argument can be used in a balanced way, also improving floating-point computations [26]. However, this is more a research challenge than the state of the art. Compilers are currently not good enough to be given such freedom, all the more as such improvements will conflict with reproducibility. For instance, the current posit standard wisely prevents the compiler from unsheathing the quire if the programmer didn't instruct it to do so: *Fused operations are distinct from non-fused operations and must be explicitly requested.* Another approach would have been to let the compiler decide which operations will be fused, but it would have sacrificed reproducibility. Code such as elementary function code definitively needs reproducibility. For similar reasons, in floating point, the fusion of $a \times b + c$ into an FMA can lead to all sorts of very subtle bugs, and is disabled by default in mainstream compilers.

In 1987, Demmel [10] analyzed two number systems, including one that has properties close to the properties of posits [21]. The issues raised by Demmel still apply with posits: reliability with respect to over/underflow has been traded with reliability with respect to round-off.

Of course, as already mentioned by Demmel [10], there are possible work-arounds: To be able to use classical error analysis, one could have some global register containing the smallest significand size that appeared in a given calculation. This seems difficult to reconcile with massive parallelism and pipelined calculations; Besides, the programmer would have to manage this register.

One could also be tempted to manually "rescale" the calculations. But this will be quite tedious, and impossible if the orders of magnitude of the input values are not known in advance by the programmer. More importantly, it is something programmers no longer worry about: the days of fixed-point arithmetic are long forgotten, and not regretted. The success of floating point is largely due to its freeing programmers from having to think about the magnitude of their data. With posits, a programmer has to think about the scale of his data again, although the fact that posits will be accurate most of the time may encourage him to be lazy in this respect.

However, this last point can be expressed more positively. In principle, thinking about the scale of data helps writing better, more robust code. Even in floating point, classical buggy behaviours like $x^2/\sqrt{x^3+1}$ (which when x grows, first grows, then is equal to zero, then to NaN) are easily avoided if programmers think of the issue before it hits. Posits, with their large "golden zone", could be viewed as a sweet spot between fixed-point (where the scale of each data has to be managed very rigorously) and floating point (where you can completely ignore the scaling problem, until the day when an overflow-induced bug hits you catastrophically). The golden zone is so large that in many cases, it allows programmers to manage the scaling issue very conservatively. Maybe this will also ease the design of automatic analysis tools that assist programmers, anticipating scaling-related loss of accuracy. For posits to displace floating point, we first need to ensure that such tools exist and are pervasive enough (embedded in compilers and environments like Matlab) that programmers cannot escape them.

Acknowledgements

Many thanks to Nicolas Brunie, Claude-Pierre Jeannerod, Nicolas Louvet, and Gero Müller for interesting comments and discussions about this work. We also thank the anonymous reviewers whose detailed suggestions helped improve this article.

REFERENCES

[1] C. S. Anderson, S. Story, and N. Astafiev. 2006. Accurate Math Functions on the Intel IA-32 Architecture: A Performance-Driven Design. In *7th Conference on Real Numbers and Computers*. 93–105.

[2] Javier D. Bruguera. 2018. Radix-64 Floating-Point Divider. In *25th Symposium on Computer Arithmetic*. IEEE, 87–94.

[3] Nicolas Brunie. 2017. Modified FMA for exact low precision product accumulation. In *24th IEEE Symposium on Computer Arithmetic (ARITH-24)*.

[4] Rohit Chaurasiya, John Gustafson, Rahul Shrestha, Jonathan Neudorfer, Sangeeth Nambiar, Kaustav Niyogi, Farhad Merchant, and Rainer Leupers. 2018. Parameterized Posit Arithmetic Hardware Generator. In *36th International Conference on Computer Design (ICCD)*. IEEE, 334–341.

[5] William James Cody. 1980. *Software Manual for the Elementary Functions*. Prentice-Hall.

[6] Florent de Dinechin, Alexey V. Ershov, and Nicolas Gast. 2005. Towards the Post-Ultimate libm. In *17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. 288–295. https://doi.org/10.1109/ARITH.2005.46

[7] Florent de Dinechin, Christoph Q. Lauter, and J.-M. Muller. 2007. Fast and Correctly Rounded Logarithms in Double-Precision. *Theoretical Informatics and Applications* 41 (2007), 85–102.

[8] Florent de Dinechin, Bogdan Pasca, Octavian Creţ, and Radu Tudoran. 2008. An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products. In *Field-Programmable Technologies*. IEEE, 33–40.

[9] J. Demmel and H. D. Nguyen. 2013. Fast Reproducible Floating-Point Summation. In *21th IEEE Symposium on Computer Arithmetic (ARITH-21)*. 163–172. https://doi.org/10.1109/ARITH.2013.9

[10] James W. Demmel. 1987. On error analysis in arithmetic with varying relative precision. In *8th Symposium on Computer Arithmetic (ARITH)*.

[11] Posit Working Group. 2018. Posit Standard Documentation. Release 3.2-draft.

[12] John L Gustafson and Isaac T Yonemoto. 2017. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (2017), 71–86.

[13] N. J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). SIAM, Philadelphia, PA.

[14] IEEE754-2008 2008. IEEE Standard for Floating-Point Arithmetic. IEEE 754-2008, also ISO/IEC/IEEE 60559:2011. (Aug. 2008).

[15] Jeff Johnson. 2018. *Rethinking floating point for deep learning*. arXiv:1811.01721. Facebook.

[16] Peter Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller. 2009. On the computation of correctly-rounded sums. In *19th IEEE Symposium on Computer Arithmetic (ARITH-19)*.

[17] Ulrich W. Kulisch. 2002. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag.

[18] P. Langlois and N. Louvet. 2007. How to Ensure a Faithful Polynomial Evaluation with the Compensated Horner Algorithm. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18)*. 141–149.

[19] Peter Lindstrom, Scott Lloyd, and Jeffrey Hittinger. 2018. Universal Coding of the Reals: Alternatives to IEEE Floating Point. In *CoNGA, Conference on Next Generation Arithmetic*. ACM.

[20] Peter Markstein. 2000. *IA-64 and Elementary Functions: Speed and Precision*. Prentice Hall.

[21] S. Matsui and M. Iri. 1981. An overflow/underflow free floating-point representation of numbers. *Journal of Information Processing* 4, 3 (1981), 123–133. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[22] Ole Møller. 1965. Quasi double-precision in floating point addition. *BIT Numerical Mathematics* 5, 1 (1965), 37–50.

[23] Jean-Michel Muller. 2016. *Elementary functions, algorithms and implementation, 3rd Edition*. Birkhaüser Boston. https://doi.org/10.1007/978-1-4899-7983-4

[24] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhauser Boston.

[25] Kwok C. Ng. 1992. *Argument reduction for huge arguments: good to the last bit*. Technical Report. SunPro, Mountain View, CA, USA.

[26] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Programming Language Design and Implementation*. ACM, 11. https://doi.org/10.1145/2737924.2737959

[27] Artur Podobas and Satoshi Matsuoka. 2018. Hardware Implementation of POSITs and Their Application in FPGAs. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 138–145.

[28] Siegfried M. Rump, T. Ogita, and S. Oishi. 2008. Accurate Floating-Point Summation Part I: Faithful Rounding. *SIAM Journal on Scientific Computing* 31, 1 (2008), 189–224. https://doi.org/10.1137/050645671

[29] Siegfried M. Rump, T. Ogita, and S. Oishi. 2008. Accurate Floating-point Summation Part II: Sign, K-fold Faithful and Rounding to Nearest. *SIAM Journal on Scientific Computing* 31, 2 (2008), 1269–1302.

[30] Hani H. Saleh and Earl E. Swartzlander. 2008. A Floating-Point Fused Dot-Product Unit. In *International Conference on Computer Design (ICCD)*. 426–431.

[31] Ping Tak Peter Tang. 1989. Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic. *ACM Trans. Math. Software* 15, 2 (1989), 144–157.

[32] Michael B. Taylor. 2012. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference*. ACM.

[33] Yohann Uguen, Florent de Dinechin, and Steven Derrien. 2017. Bridging High-Level Synthesis and Application-Specific Arithmetic: The Case Study of Floating-Point Summations. In *Field-Programmable Logic and Applications*. IEEE.