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 softare 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 challenge for the community. Some, in particular the loss of scale invariance for accuracy, are extremely dangerous if Posits are to replace floating point. This study helps framing where Posits are better than floating-point, where they are worse, and what tools are missing in the Posit landscape. For general-purpose computing, using Posits as a storage format only could be a way to reap their benefits without loosing those of classical floating-point. The hardware cost of this alternative is studied.

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, 9 pages.

1 INTRODUCTION

Efficent low-level floating-point code, such as high-performance elementary functions [14, 17] 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 [18]. 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 (e.g. Cody and Waite, or Payne and Hanek [17]). 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 [6, 11, 21, 22] and others [4, 5, 12, 14].

Posit developers of low-level code will need to develop a similar bag of tricks. A good starting point is to evaluate the floating-point tricks, and see if they can be transposed to Posits. This is the objective of sections 2 and 3.

However, floating-point computing is not just using tricks: actually, most programmers ignore them. They simply rely on operations and toolboxes that themselves have been designed using a body of numerical analysis techniques, similarly accumulated over the years [8]. 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 discussed in Section 4.

Section7 lists some of the developments that are needed to assist developers of numerical software based on Posits. Section 6 suggests that considering Posits as a storage-format only could be a solution that is both safe and cost-effective. Its cost is evaluated numerically.

Notations and conventions

This article assumes basic knowledge of the Posit format, and refers to the data available on https://posithub.org/ for a detailed description of the Posit format itself.

We use the following notations inspired by the Python `sfpy` package[1] which is a reference implementation of the upcoming Posit standard.

- 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=2.
- PositN represents any of the previous
- PositN(i) with $i \in \mathbb{N}$ denotes the PositN number whose encoding is the integer i.

We note the posit operations $\oplus$, $\ominus$ and $\otimes$.

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

CongA 2019, march 2019, Singapore

Casting posits to floats

The following lemma is widely used in this article:

Lemma 1.1 (Exact cast to binary64). *Any Posit8, Posit16 or Posit32 except NaR is exactly representable as an IEEE754 binary64 (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 correponding ranges of Binary64. Therfore, any posit exponent and any posit mantissa fraction will be converted exactly to a Binary64 exponent and mantissa. □

2 THE GOOD: ROUNDING PROPERTIES AND DOUBLED PRECISION

This section studies the transposition to the Posit number system of doubled-precision techniques that are widely used in floating-point.

2.1 Is the ounding error in the addition of two posits a posit?

The rounding error of the addition of two IEEE 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 other properties of this section, and 2/ it has the following corollary: $a \ominus b == 0$ is equivalent to $a = b$ (except when a or b is NaN), which programmers will assume and compilers can exploit.

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 2.3

Lemma 2.1. *The rounding error in the addition of two Posit8 is always a Posit8.*

Lemma 2.2. *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 2.3. *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 Posit(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.

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

2.2 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 2.4. *For any two positN of the same format a and b different from NaR,*

$$\frac{a}{2} \le b \le 2a \implies a \ominus b = a - b$$

.

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

Lemma 2.5. *For a given PositN format, if the fraction field length of 2^l is $p \ge 1$ bits, then for all d in $[\![0, p]\!]$, the fraction field length of 2^{l-d} is at least $p - d$.*

Proof. By recursion over d:

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

Now, assuming that the property holds for a given $d < p$, we can encode 2^{l-d} with at least $p - d$ mantissa 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 mantissa length $p - d \ge p - d - 1$.

If it is equal to zero, then we need to decrement the range and the exponent field becomes $2^{es} - 1$. As $p - d \ge 1$, 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 mantissa field length is reduced by at most 1.

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

Proof of Lemma 2.4. For the rest of the proof, $\mathbb{U}_{w,es}$ 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

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

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

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

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 mantissa bit.

Noting ulp(·) the function that associates to a posit the weight of the last bit of its mantissa, and ufp(·) the function that associates the weight of the first bit of the mantissa, we have $\text{ulp}(a) \ge \text{ulp}(b)$. The equality $\text{ulp}(a) = \text{ulp}(b)$ occurs either when $\text{ufp}(a) = \text{ufp}(b)$, or when $\text{ufp}(a) = 2\text{ufp}(b)$ and a has one extra bit of precision. We have $a - b = k_a \times \text{ulp}(b) - k_b \times \text{ulp}(b) = (k_a - k_b) \times \text{ulp}(b) = k \times \text{ulp}(b)$, 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 2.5, $a - b$ can be represented as a Posit. □

2.3 TwoSum works in posit arithmetic

In the following, TwoSum denotes the algorithm introduced by Knuth in [10] and Møller in [16]. This algorithm inputs two floating points number a and b. It returns two floating points s and r such

as $s = RN(a + b)$ and $a + b = s + r$ exactly under the condition that $a + b$ does not overflow (underflow to subnormals is not a problem [18]). Here, RN denoting 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 2.6. *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 IEEE754 binary64 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 `fs1+fs2=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.

2.4 FastTwoSum doesn't work with Posits

The FastTwoSum algorithm below [18] compute the same decomposition as TwoSum, but needs fewer operations, at the cost of an additional constraint on the inputs: 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
```

The FastTwoSum does not work for all inputs. One counter-example is the addition of Posit8(0x5F), which represents the binary number 1.11111, with itself. 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 sum and the operand is 10.00001, which would normally be encoded as 0110000001, rounded to 01100000 (10.000). Finally, the final difference gives -0.00111, which is exactly encoded where the exact error should be -0.0001.

Finding faster variants to TwoSum, possibly under different conditions, remains an open problem.

2.5 Doubled precision versus exact accumulation

To sum up this section, doubled precision is possible and practical with Posits.

The reader may feel it is useless if Posits support an exact accumulator (the quire in Posit terms). Here is one justification related to implementations of elementary functions.

First, it is highly probable that exact accumulators eventually find their place in microprocessors. Indeed, several existing machine learning accelerators [2, 9] already use variations of the exact accumulator to compute on IEEE 16-bit floating point. For larger sizes, this could be an useful instance of "dark silicon" [23]. This trend was anticipated in the evolution of the IEEE-754 standard, which included in 2008 several reduction operations: summation, sum of products, sum of squares, sum of absolute differences.

However, the quire is currently practical (in terms of microarchitecture) for Posit8 and Posit16. For larger sizes, supporting the quire in hardware still faces the same issues as the Kulisch accumulator: it is very hardware expensive, conversion back to posit has a long latency, there cannot be as many quires as we have registers, it is a huge number of bits to save and restore in context switches between processes, etc.

This overhead is perfectly acceptable for large linear algebra operations, where its relative cost can be made negligible. When writing accurate elementary functions, however, the need of extended precision is limited to a few extra bits and a few operations [14]. Doubled precision makes much more sense here.

3 THE BAD: EXACT PRODUCTS AND TWOMULT

3.1 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 mantissa 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 will not be exact either.

We sill have in Posits the following lemma:

Lemma 3.1 (Exact multiplications by a power of two). *If $2^k x$ is closer to one than x, then the PositN product $x \otimes 2^k$ is exact.*

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

Unfortunately, in the current implementations of the exponential function in floating-point, 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 an exact multiplication by a power of two. With Posits, this multiplication will introduce an error term, which will make an accurate or correctly rounded implementation more challenging.

3.2 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.

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 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%).

3.3 Multiplicative cancellation

In floating-point, an addition that cancels is for instance $x - 3.1416$ in the case $x = 3.1421$. The computed result will be $4.0000 \cdot 10^{-4}$. 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 mantissa 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 do have the classical addition cancellation issue, but we also have a very similar situation when multiplying a very large number by a very small one. Thanks to tapered arithmetic, it may happen that the operation itself entails no rounding error (as soon as we have twice as many fraction bits in the results as we had in the inputs), while having most of these fraction bits meaningless (even the non-zero ones). Section 5.3 will show that multiplicative cancellations naturally happen in many physics simulations.

Here, the challenge is the same as for additive cancellation: on the one side, this 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.

4 THE UGLY: ACCURACY DEPENDS ON SCALE, OR, NO STANDARD MODEL FOR NUMERICAL ANALYSIS

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 [8]) is based on this very useful property. Just to give a simple and recent example, if s is the exact sum of n floating-point numbers $a_1, a_2, \ldots, a_n$ and $\hat{s}$ is the computed sum (computed using the naive "recursive" summation algorithm), then [20]

$$|s - \hat{s}| \leq (n-1)u \sum_{i=1}^{n} |a_i|. \tag{1}$$

This is no longer true with posits. More precisely, in the Posit$n$ 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

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

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

For illustration, provided that no underflow/overflow occurs, with binary32 numbers, the relative error of all operations will be bounded by 2^{-24}. With posits32, the error due to rounding will be smaller if the absolute value of the result is between 2^{-16} and 2^{16}, of the same order if it is between 2^{-20} and 2^{-16} or between 2^{16} and 2^{20}, and larger elsewhere. It becomes larger and larger as the numbers becomes farther from 1. To illustrate this, Fig. 1 plots the error due to rounding a real number between 2^{15} and 2^{22} to the nearest Posit32 number, and to the nearest binary32 number.

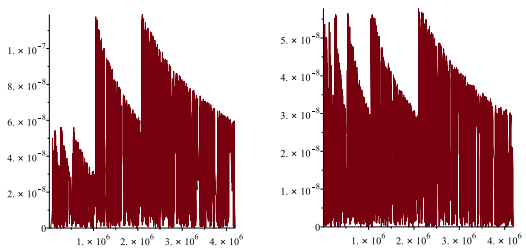

Figure 1: Error due to rounding a real number between 2^{15} and 2^{22} to the nearest Posit32 number (left) and Binary32 (right)

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. And if intermediate values significantly larger than 2^{20} or significantly smaller than 2^{-20} appear in calculations, much accuracy can be lost.

In 1987, Demmel [7] analyzed two number systems, including one that has properties close the the properties of Posits [15].

The issues raised by Demmel still apply with Posits: reliability with respect with over/underflow has been traded with reliability with respect to roundoff.

Of course, as already mentioned by Demmel [7], there are possible work-arounds:

- to be able to use classical error analysis, one could have some global register containing the smallest mantissa 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.
- to use posit arithmetic where it is most accurate, one could 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 lost the habit of worrying about: the days of fixed-point arithmetic are long forgotten and not regretted. Floating-point's success is largely due to its freeing programmers from having to think about the magnitude of their data. A pernicious effect is that programmers will be encouraged to be lazy in this respect by the fact that Posits will be accurate most of the time.

And when large or tiny intermediate values do appear in calculations, the obtained results may be very inaccurate.

5 WHEN POSITS ARE BETTER, AND WHEN THEY ARE WORSE

The Posit literature is full of examples where computing with Posits turns out to be much more accurate than computing with floats of the same size.

We believe it is healthy to first show two very simple examples where Posits behave much worse. These examples will help us illustrate in which situations Posits are better or worse, and more importantly, why.

5.1 Two anti-Posit examples

All the integers between 1 and one million are representable exactly both as Posit32 and as binary32. A simple benchmark is to measure the accuracy of computing the sum of the n first integers. Here are a few easy to reproduce examples.

The sum of the 5000 first integers is computed exactly in binary32. Computed in Posit32, it yields a relative error of $3.6 \cdot 10^{-5}$

The exact sum of the 100000 first integers is 4999950000. The sum computed in Posit32 yields 5001658368, with a relative error of about $3.4 \cdot 10^{-5}$. The sum computed in IEEE float yields 4999890432, with a relative error of $1.2 \cdot 10^{-5}$.

It gets worse for larger n, for instance for $n = 10^6$, the respective relative errors of Posit32 versus binary32 are $2 \cdot 10^{-2}$ and $1.2 \cdot 10^{-4}$.

The standard argument is that Posit computations should be scaled. Can we really expect programmers to scale a sum of integers?

The second example, inspired from [7], 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 binary32 arithmetic, we obtain a relative error $1.723629924 \cdot 10^{-7}$, and with posit32 arithmetic, the relative error is $1.053488201 \cdot 10^{-4}$. For $x = 25$ and $n = 30$, with binary32 arithmetic, we obtain a relative error $1.497794183 \cdot 10^{-7}$, and with posit32 arithmetic, the relative error is $8.023861890 \cdot 10^{-2}$.

5.2 When Posits are better

There has been very serious studies [13] that report a consistent accuracy advantage of Posits over floating-point. It may seem counter-intuitive, as some of the posits operations can be very inaccurate, and these inaccurate operations do happen in the previous studies. Intuition tells us that a computation can not be more accurate than its least accurate step. Why is it not the case in these studies?

A fair rule of thumb is the following: *in a summation yielding a result with an exponent close to zero, Posits are 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. Floating point is computing in vain bits that are going to be shifted out.

Note that the condition that exponent should be close to zero is a relatively soft one: for instance, Posit32 has more fraction bits than IEEE binary32 for numbers whose magnitude ranges from 10^{-6} to

h (Planck)	$6.626070150 \cdot 10^{-34}$
Posit32 value	$\approx 7.70 \cdot 10^{-34}$
binary32 value	$\approx 6.260701786 \cdot 10^{-34}$
N_A (Avogadro)	$6.02214076 \cdot 10^{23}$
Posit32 value	$\approx 6.021 \cdot 10^{23}$
binary32 value	$\approx 6.0221406 \cdot 10^{23}$
c (speed of light)	299792458
Posit32 value	299792448
binary32 value	299792384

Table 1: Posit32 are less friendly to physicists than Binary32

10^6 (this interval is defined more precisely in next Section). The "golden zone" thus defined is fairly comfortable to live with, and may easily hide the fact that out of this zone, Posit32 become less accurate than binary32.

This rule of thumb only applies to summations. As soon as we have, for instance, a sum of products, one single "multiplicative cancellation" may ruin the accuracy of the result.

Still, this rule of thumb confirms that Posits are a very good choice for current machine learning, where convolutions and other neuron summations are scaled at each level to remain in the golden zone.

5.3 When Posits are worse

Table 1 gives the recently standardized[2] values of the Planck constant and the Avogadro number, as well as the nearest Posit32 and Binary32 numbers. We have a clear winner here. 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 multiplicative cancellation! The computed energy e will have no meaninglful digit, even if ν is large enough (there are X rays in the petaherz range) to bring e in a range where it has many mantissa bits.

Another example is the most famous physics equations, $e = mc^2$, where the value of c^2 will be computed in Posit32 with a relative error of $3.29 \cdot 10^{-5}$ while the binary32 error is $6.67 \cdot 10^{-8}$.

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). 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, while unit scaling (mega, micro, etc) are powers of ten, with negative powers of ten not exactly represented in binary. Worse, if code doesn't use the standard units, it then becomes critical to attach the unit to each data exchanged (let us recall that 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.

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

Table 2: Minimum sizes (in bits) required for encoding posits in a floating-point format

	minP	maxP	w_E	w_F
posit8, es=0	2^{-6}	2^{6}	4	5
posit16, es=1	2^{-28}	2^{28}	6	12
posit32, es=2	2^{-120}	2^{120}	8	27
posit64, es=3	2^{-496}	2^{496}	10	58

6 HARDWARE CONSIDERATIONS

To complete the panorama, it is important to have an idea of the cost of hardware Posit support.

6.1 Dissolving Posits in IEEE-like formats

Table 2 gives, for each standard Posit format, the minimum exponent size w_E and minimum mantissa fraction size w_F such that all the posits can be represented exactly in an IEEE-like format with these parameters. w_E is deduced from the smallest (in magnitude) and largest possible Posit values, respectively noted minP and minP.

$$\text{minP} = 2^{-(N-2)\cdot 2^{\text{es}}} \tag{2}$$

$$\text{maxP} = 2^{-(N-2)\cdot 2^{\text{es}}} \tag{3}$$

w_E and w_F are given by the following formulae, where N is the overal size in bits of the Posit$N$ format:

$$w_E = \lceil log2(N-1) \rceil + 1 + es \tag{4}$$

$$w_F = N - 1 - 2 - es \tag{5}$$

There are two ways to exploit Table 2 to integrate a Posit unit in a processor. The first one is to build stand-alone Posit operators, and has already been explored [3, 19]. We briefly review it first.

The second option, reviewed in Section 6.3, consists in using Posits as a storage format only, and computing internally on standard IEEE-754 operators.

6.2 Building standalone Posit operators

To build standalone hardware Posit operators [3, 19], one may combine a posit-to-FP decoder, an FP operator built for non-standard values of w_F from Table 2, 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 the standard 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 usually be compressed in a few bits, e.g. "guard, round and sticky" in addition.
- it may use encodings of the exponent and mantissa that are simpler (e.g. two's complement for the exponent instead of standard biased format) or more efficient (e.g. redundant form for the mantissa);

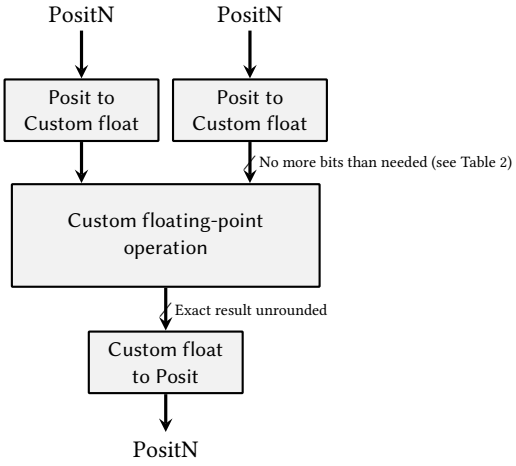

Figure 2: Architecture of a standalone Posit operator

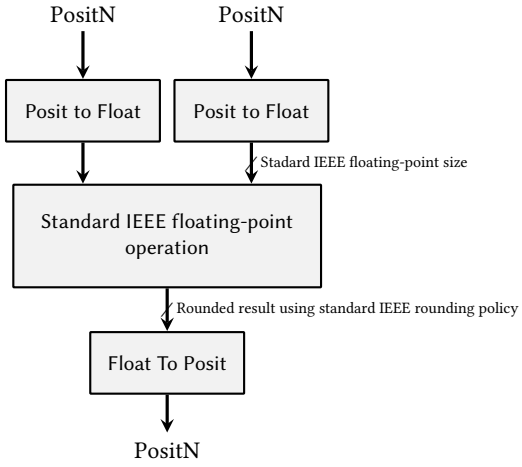

Figure 3: Architecture of Posit operator relying on existing standard floating-point hardware support

- it doesn't need to manage subnormals, as the chosen intermediate formats can encode any posit as normalized mantissa;

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

Beyond these simplifications, this three-block implementation embodies the inavoidable amount of work in a Posit operation, which remain slightly higher (both in area and delay) as floating-point of the same size. The latency can be reduced (but at increased hardware cost) by hardware speculation or the equivalent of classical dual-path FP adder architectures.

6.3 Using Posits as a storage format only

Table 2 also shows that Posit8 and Posit16 can be cast errorlessly in binary32, while Posit32 can be cast errorlessly in Binary64. 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.

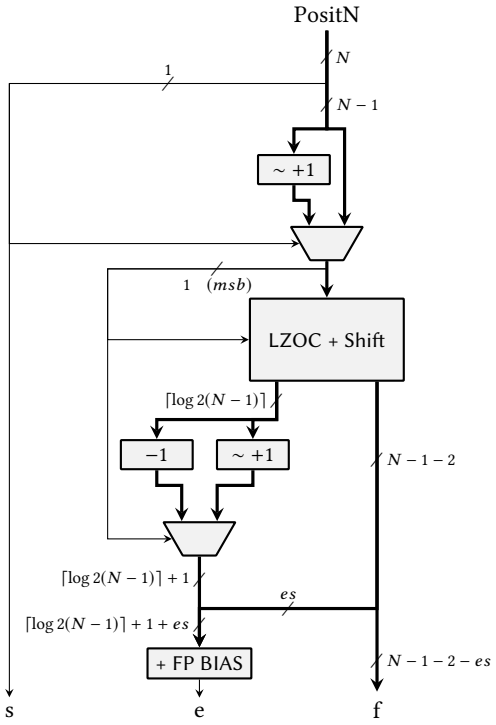

Figure 4: Architecture of a positN to floating-point decoder

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 with all the properties that are lost on Posits;
- Well established FP libraries can be mixed and matched with emerging Posit developments;

It has one drawback that should not be overlooked:

- Using this approach does not allow to implement standard posit operations directly, as double rounding will occur in some situations.

Note that a Posit will never be converted to a subnormal, and that subnormals will all be converted to the zero 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 4 and 5 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 4 and 5, and

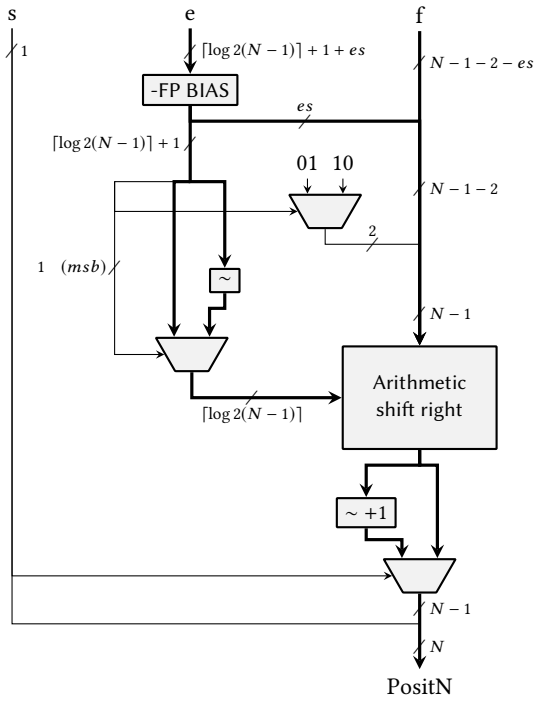

Figure 5: Architecture of a floating-point to positN encoder

	LUTs	Reg.	DSPs	Cycles @ Freq
Posit8 to FP32	26	32	0	5 @ 602MHz
FP32 to Posit8	20	30	0	3 @ 508MHz
Posit16 to FP32	76	78	0	7 @ 542MHz
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	186	192	0	9 @ 508MHz
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.

relied on standard floating point operators integrated in Vivado HLS.

Some synthesis results are given for a Kintex 7 (xc7k160tfbg484-1) FPGA 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.

7 CONCLUSION: TOOLS NEEDED

This article was a survey of the strengths and weaknesses of Posit numbers when compared to floating-point. This included 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.

When Posits are better than floats of the same size, they provide one or two extra digits of accuracy [13]. When they are worse than floats, the degradation of accuracy can very easily be arbitrarily large. This simple observation should prevent us from rushing to replace all the floats with posits.

Another reason not to rush is the fact that forcing programmers to manage the scale of their data is a clear regression.

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.

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

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

[4] 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

[5] 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.

[6] 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

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

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

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

[10] D Knuth. 1998. *The Art of Computer Programming vol. 2 Seminumerical Algorithms*. Reading, Massachusetts: Addison Wesley.

[11] 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)*.

[12] 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.
[13] 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.
[14] Peter Markstein. 2000. *IA-64 and Elementary Functions: Speed and Precision*. Prentice Hall.
[15] 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.
[16] Ole Møller. 1965. Quasi double-precision in floating point addition. *BIT Numerical Mathematics* 5, 1 (1965), 37–50.
[17] Jean-Michel Muller. 2016. *Elementary functions, algorithms and implementation, 3rd Edition*. Birkhaüser Boston. https://doi.org/10.1007/978-1-4899-7983-4
[18] 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.
[19] 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.
[20] Siegfried M. Rump. 2012. Error estimation of floating-point summation and dot product. *BIT Numerical Mathematics* 52, 1 (2012), 201–220. https://doi.org/10.1007/s10543-011-0342-4
[21] 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
[22] 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.
[23] Michael B. Taylor. 2012. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference*. ACM.