Comparing posit and IEEE-754 hardware cost

Luc Forget, Yohann Uguen, Florent de Dinechin

Comparing posit and IEEE-754 hardware costs

Luc Forget, Yohann Uguen, Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI, France
first-name.last-name@insa-lyon.fr

***Abstract*—The posit number system is an elegant encoding of floating-point values proposed as a drop-in replacement for the IEEE-754 standard. On the one side, posits sacrifice some of IEEE-754 complexity (directed rounding modes, infinities, NaNs). On the other side, their variable-size exponent and significand fields require extra encoding and decoding steps, and their higher best-case accuracy requires wider data-paths. The posit encoding/decoding overhead can be reduced by keeping posits decoded in processor registers, with the operators suitably modified to avoid double-rounding issues.**

An unbiased quantitative comparison of the hardware costs of these two encodings is based on an analytical study and an open-source C++ library suitable for High-Level Synthesis. This library offers posit and IEEE-754 parametrized operators for addition/subtraction, multiplication, and exact accumulation, all developed with the same high design effort and fully compliant to their respective standards.

This library improves the state of the art of posit hardware arithmetic, and still, IEEE-754 operators remain between 30% and 60% faster and smaller than their posit counterparts.

I. Introduction

The set of real numbers is infinite and uncountable. A convenient way to manipulate real numbers in a computer is to define a code size N (a number of bits), then select a finite subset of the real numbers of at most 2^N elements to be encoded on the 2^N available binary codes. Both the subset and its encoding must be be carefully selected so that the computations can be efficiently implemented out of the machine codes. When the exact result of a computation does not itself belong to the representable subset, a convention (rounding and overflow management) specifies what to do.

The mainstream example of such a computer-oriented encoding of real numbers is the IEEE-754 floating-point standard [1]. It defines number sets and encoding schemes for $N \in \{16, 32, 64, 128\}$, and a versatile set of rounding and overflow conventions. A recently introduced alternative, for comparable applications, is the posit encoding scheme [2], [3]. It is presented as a drop-in replacement of the IEEE-754 formats providing better performance and accuracy for the same N thanks to a more efficient encoding [2].

Many works have compared the accuracies of posits and IEEE floats [4], [5], [6], [7]. The focus of the present article is the comparison of their hardware implementation cost.

The method for this comparison has been to develop an open-source[1], fully parametric library of hardware operators called MArTo that covers both IEEE-754 operators and posits of arbitrary sizes, such that posits can effectively be used as a drop-in replacement for IEEE floats. This tool will allow the community to assess their relative speed and cost, just like software libraries such as SoftFloat and SoftPosit have enabled like-for-like comparisons in accuracy.

To make this comparison as unbiased as possible, all the operators are developed with similar design effort using the same core library of fixed-point components. The common design goals are 1/ full standard compliance, 2/ combinatorial designs that can be pipelined for higher frequency, and 3/ area-oriented designs. Classical techniques that improve latency at the expense of area, such as dual-path floating-point addition [8] or hardware speculation [9] are not considered here – as the reader will see, they can benefit posits as well as IEEE. However, the datapath sizes are carefully minimized.

The IEEE-754 side of MArTo is a competent reimplementation of the state of the art and will be presented very briefly. Conversely, posit literature is younger, and this article discusses in detail all the choices made for posit operators.

The posit system is a clever encoding of a selected subset from a larger set of floating-point numbers. To compute on posits, one must first decode them into this larger set [10], [11], [12]. This set is defined formally in Section II, along with an hardware-efficient encoding called the Posit Intermediate Format (PIF).

Section III defines two alternative approaches that can be used in a Posit Arithmetic Unit (PAU): one that performs posit encoding and decoding for each operation, and one where the posit registers hold already decoded PIF data. This second approach reduces the latency of the operations, but requires larger registers, and more complex rounding hardware to be bit-for-bit compatible with the first.

Section IV describes in detail all the hardware blocks that constitute a PAU in each of these alternatives. Section V discusses the implementation of exact accumulators [13] for posits and floating-point.

Finally, Section VI evaluates and compares the costs and delays of posit and IEEE-754 operators on FPGA hardware. To ensure an unbiased comparison, we first show that the proposed posit implementation improves the state of the art, while the IEEE-754 operators compare favorably to it.

II. Encodings of binary floating-point numbers

A. IEEE-754 binary floating-point numbers

An IEEE-754 binary encoding scheme [1] is defined by two positive integers:

- W_e, the exponent field width and
- W_f, the fraction field width.

A value is represented by a bit vector of size $1 + W_e + W_f$, composed of a sign bit s, an exponent field e of size W_e, and a fraction field f of size W_f.

[1] https://gitlab.inria.fr/lforget/marto

When the bits of e are not all ones or all zeros, the encoding represents a normal number with the value

$$x = (-1)^s \times 1.f \times 2^{e-E_{\max}} \tag{1}$$

where e is interpreted as a positive binary integer and $E_{\max} = 2^{W_e-1}-1$. The constant '1' before the fractional point ensures the unicity of the representation.

When all the bits of e are zeroes, this implicit leading 1 is replaced by a zero (subnormal numbers), and the value represented is interpreted as

$$x = (-1)^s \times 0.f \times 2^{E_{\min}} \tag{2}$$

with $E_{\min} = 1 - E_{\max}$. Zero is encoded as a subnormal number with all fraction bits set to zero, entailing the controversial issue that there are two encodings for zero, differing on the sign bit.

The encodings when all the bits of e are ones are reserved for special non-numerical values: infinity if all fraction bits are zeros, or Not a Number (NaN) if at least one fraction bit is one. The infinity values are overflow markers used when the result of an operation is greater than the greatest normal number. NaNs represent the output of an illegal arithmetic operation.

B. The posit encoding

The posit number system [2] also encodes floating-point values. A posit format is defined by two positive integers:

- the word size N,
- the exponent scale size, W_{es}

A value x is represented by a bit vector of N bits (see Figure 1). It starts with the sign bit s. Next comes the *regime*, a variable-length field whose length encodes a coarse grain exponent as detailed below. Then follows the exponent scale field, e_s, of at most W_{es} bits. The remaining bits (if any) constitute the fraction part.

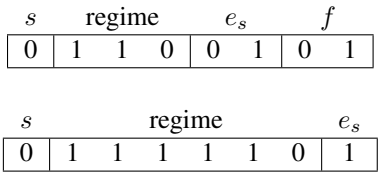


Fig. 1: Two posit decomposition examples ($N = 8$, $W_{\text{es}} = 2$)

The regime field consists of a sequence of l identical bits b, and stops at the first bit different from b (Figure 1) or the end of the word. The value e_h encoded by the regime is $-l$ if the bits of this sequence are equal to s, otherwise $l-1$. In the encoding of Figure 1, top, the sequence consists in two 1s: $l = 2$, and the sign is 0, therefore $e_h = 1$. The bits of the e_s field are xored with s to obtain the lower exponents bits e_l. Finally, the exponent E is the concatenation of e_h and e_l:

$$e = 2^{W_{\text{es}}} e_h + e_l \tag{3}$$

In our example, $e = 101$ as $e_l = 01$.

The remaining bits encode the fractional part f of the significand. An implicit leading bit i is obtained by negating s, here $i = 1$. Finally, the value of the posit is:

$$x = 2^e \times (i.f - 2s)\,. \tag{4}$$

The value represented by the top example of Figure 1 is

$$2^{101_2} \times (1.01_2 - 2 \times 0) = 2^5 \times 1.25 = 40.$$

Note that the regime can extend to the point where there is no room for f or e_s. In this case, the shifted out bits are assumed to be zeros. For instance, in Figure 1, bottom, the length of the regime is $l = 5$, hence $e_h = 4$. The exponent shift value is $e_l = 10_2$ and there is no fraction bit. The represented value is then

$$2^{10010_2} \times (1.0 - 2 \times 0) = 1 \times 2^{10} = 1024$$

Posit formats admit two special values, 0 and Not a Real (NaR). For encoding 0, all the posit fields are null, including the implicit bit. NaR is the equivalent of IEEE-754 NaN (Not a Number). Its encoding only has the sign bit set. Overflows are not encoded: posit arithmetic silently saturates instead.

C. Posit smallest floating-point superset

The sizes of exponent and fraction in a posit value depend on the regime size. However, the hardware bit-widths cannot change dynamically, they have to be designed for the worst case. As (4) shows, posits are an encoding of floating-point numbers. This worst case therefore corresponds to the smallest fixed-precision floating-point format that is a superset of posit numbers. To match (4), this floating-point format only includes normal numbers with fractions expressed in two's complement.

The parameters for the smallest fixed precision floating-point superset are derived as follows. The the size W_f of a two's complement fraction is the maximum precision of a posit value, obtained for the minimum length $l = 2$ of the regime, therefore

$$W_f = N - (3 + W_{\text{es}}) \tag{5}$$

The maximal exponent is obtained when the regime length is $N-1$. In this case, all the e_s and f bits are pushed out by the regime. Hence, the maximum exponent value is

$$E_{\max} = (N-2)2^{W_{\text{es}}} \tag{6}$$

As the opposite exponent can also be reached, the number of bits needed to store the exponent is

$$\begin{aligned} W_e &= 1 + \lceil \log_2 \left((N-2)2^{W_{\text{es}}} \right) \rceil \\ &= 1 + W_{\text{es}} + \lceil \log_2 (N-2) \rceil \end{aligned} \tag{7}$$

The W_{es} parameter allows trading between the range of the format and its maximal precision. The posit draft standard [3] defines four formats with an encoding size N of 8, 16, 32 and 64 respectively, such that

$$W_{\text{es}} = \log_2(N) - 3 \text{ for standard posits.} \tag{8}$$

These formats are used for evaluation in this paper, although the MArTo library is fully parameterized in N and W_{es}.

TABLE I: Parameters of standard posit formats.

Standard posit			smallest FP superset		quire	internal formats	
N	W_{es}	$E_{\max}$	W_e	W_f	W_q	W_{pif}	W_{upif}
8	0	6	4	5	32	12	14
16	1	28	6	12	128	21	23
32	2	120	8	27	512	38	40
64	3	496	10	58	2048	71	73

Table I gives, for each of the standard posit formats, the exponent and fraction sizes of the smallest floating-point superset.

D. Quire

A posit-compliant environment must also provide a *quire*, a fixed point accumulator large enough to allow for the exact accumulation of posit products. It is based on the floating-point Kulisch accumulator [13].

The quire bits must cover all the possible products, with exponents from $P_{\min} = -2 \times E_{\max}$ to $P_{\max} = 2 \times E_{\max}$. A positive number of carry guard bits C can be added to allow the sum of up to 2^C maximal magnitude products before an overflow occurs. The width of such a quire is then

$$W_q = P_{\max} + C - P_{\min} + 1 \tag{9}$$

From (6) and (8), one can see that for standard posit formats, product exponents range from $-\frac{N^2-2N}{4}$ to $\frac{N^2-2N}{4}$. Hence, without carry guard bits the quire width would be $W_q = \frac{N^2}{2} - N + 1$. The standard motivates that the quire should easily be transferred to and from memory. To do so, it should have a size which is a multiple of 8. With the sign bit and the addition of $C = N - 2$ carry guard bits, this goal is attained. Hence the width of the quire is

$$W_q = \frac{N^2}{2} \text{ for standard posits.} \tag{10}$$

E. The hardware-friendly Posit Intermediate Format recoding

With this smallest posit FP superset, it becomes possible to define a new intermediate encoding for numbers in this set that is more adapted to hardware arithmetic operations, in the sense that all its fields have a fixed width. In this work, this encoding scheme is denoted Posit Intermediate Format or PIF.

Figure 2 highlights the role of this format in an end-to-end posit arithmetic operator. Posits are first decoded to PIF. As PIF can represent all posit values, this operation is exact. Then, the arithmetic operation is performed on PIF data. Finally, the result is encoded back to posit. Since rounding (to an exponent-dependent position) must be performed in this encoding step, the output format of the PIF operation must be an Unrounded PIF, which is a PIF extended with all the additional information needed for standard-compliant rounding, detailed below in Section II-F.

We believe the 3-step approach of Figure 2 is inevitable for stand-alone posit operators (except for very small formats where a simple tabulation may be used). It is followed (more or less explicitly) by leading hardware posit implementations [10], [11], [12].

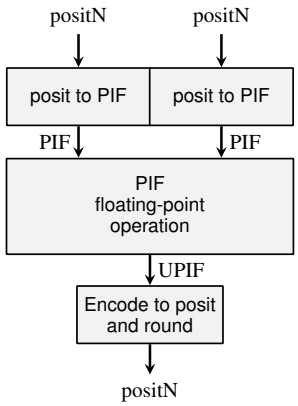


Fig. 2: Architecture of a posit operator in a PAU that uses posit registers and posit-to-posit operators.

The PIF should be designed with two objectives in mind:

- Posit to PIF conversion should be as simple as possible,
- Arithmetic operations should be efficiently computed on this representation.

Because of the second objective, PIF is a simple normalized floating-point representation that uses the parameters of Table I. To address the first objective, the proposed PIF encodes both the exponent and significand in two's complement (where IEEE-754 uses a biased encoding for exponent and a sign-magnitude representation for the significand). This avoids two's complement to sign-magnitude conversions (which may cost a carry propagation). It also has the side effect of slightly simplifying PIF addition of values with opposite signs.

Two's complement encoding for a normalized significand consists of a sign bit s and a fraction f on W_f bits. The two's complement significand value is then

$$v = -2s + \bar{s}.f \quad . \tag{11}$$

However, (11) does not allow the encoding of zero. The proposed PIF encoding scheme introduces an extra bit i which is one if and only if the represented value is strictly positive. The significand value becomes

$$v = -2s + i.f \quad . \tag{12}$$

Zero is the only posit value whose PIF representation has both s and i set to zero. This enables efficient zero detection in arithmetic operators. For non zero values, exactly one of s or i is set.

PIF also has an extra *isNaR* bit, set to one if and only if the represented value is NaR. An alternative option could have been to use a non-posit value, for instance with both s and i set to one. This would trade one bit of representation for a few gates of encoding/decoding logic.

To summarize, the PIF encoding scheme is composed of the following fields:

- a *isNaR* flag,
- the sign bit s,
- the exponent e stored in two's complement on W_e bits,
- the weight one bit i,
- the fraction bits f on W_f bits.

The encoded value is

$$v = \begin{cases} NaR & \text{if } isNaR \text{ is one} \\ (-2 + i + 0.f) \times 2^e & \text{otherwise} \end{cases} \quad (13)$$

F. Unrounded PIF encoding of the result of basic operations

In the general case, the result of an operation on two PIF values is not exactly representable as a PIF, and must be rounded. As PIF is a floating-point format, we may use textbook techniques [14], [8] for this. For the basic operations (addition, multiplication, division and square root) the exact result can always be represented on at most $2W_{\text{pif}}$ bits, then for the purpose of rounding the extra W_{pif} bits can be condensed into only two bits:

- an extra fraction bit at the LSB, called the *round* bit;
- a *sticky* bit, set if and only if the exact value is strictly greater than what is represented by the fraction f extended with the *round* bit (but still smaller than the next representable value). In other words, a *sticky* bit of zero means that the value represented by the extended fraction is exact.

We define the UPIF (Unrounded PIF) format as a PIF with these two extra bits.

The floating-point literature often uses a third additional bit (called the guard bit), useful in the case when a 1-bit normalization of the significand may be needed. In the big picture of Figures 2 and 3 , the PIF operator is in charge of this normalization, so no guard bit is needed in UPIF.

In this paper we only demonstrate the use of the UPIF format on addition/subtraction and multiplication, but it is equally suitable for division and square root. Digit recurrence algorithms [14] compute a remainder along with the quotient or square root, out of which the round and sticky bits can be computed. Multiplication-based algorithms [14], [8] also can output their result in UPIF format – for instance by computing the remainder.

Table I gives the width for PIF and UPIF associated with standard posit formats.

G. Saturation management

Posit arithmetic does not offer an overflow detection mechanism to the user. When the exact result of an operation is bigger than the biggest representable value, this biggest representable value is returned.

This saturation could in principle be handled in a generic way in the "Encode to posit and round" block of Figure 2, or in the "UPIF inplace round" block of Figure 3. However, as each operation leads to different overflow situations, it is more efficient to manage saturation in each PIF operator. Indeed, the UPIF specification exposed previously assumes that saturation has been performed by the operator, otherwise more bits would be needed. Another advantage is that some saturation situations may be detected in parallel with computation, thus reducing latency.

III. Alternatives for a hardware posit unit

This section describes the two options considered in this article for a processor supporting posit arithmetic. The detailed study of the second option is, to the best of our knowledge, novel. The purpose of this section is to define the hardware components that will be described in detail in Section IV.

A. Posit-to-posit operators

Figure 2 shows that building posit-to-posit operators consists mainly in three steps:

1) decoding the posit representation to PIF,
2) performing the computation on PIF, and
3) encoding the result back to the nearest posit value.

Here, the middle step is essentially a floating-point operation on normal numbers, a function that is present in any IEEE-754 operator. It may therefore build upon the rich floating-point literature for many tricks and trade-offs between latency and area, for instance dual-path architectures for addition, or injection rounding for multiplication. This is not the subject of the present article, since posit and IEEE-based operators are equivalent from this point of view.

What is interesting is a comparison of posits and IEEE-754 overhead with respect to this common core processing normal floating-point numbers.

The main overheads of IEEE-754 are support are the encoding, decoding, and management of special values, which is relatively cheap, and subnormal support, which may be quite expensive.

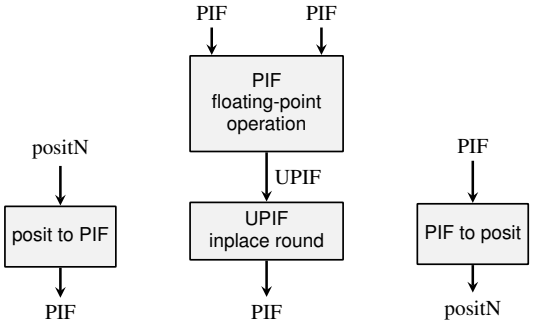
The main overheads of posits are the slightly larger datapath of the core unit, summarized in Table I, and the two conversion steps of Figure 2. As these encoding/decoding blocks involve leading zero counting (LZC) and shifting, they could in principle be compared to hardware subnormal support in an IEEE-754 multiplier (subnormal support in addition is comparatively lighter [8]). However, in an IEEE-754 multiplier, only one subnormal input needs to be considered (the product of two subnormals is tinier than the tiniest representable value), whereas both posit inputs must be converted. Besides, subnormals are part of the floating-point encoding itself, which allows to hide their latency overhead by speculation. To the best of our knowledge this is not possible with posits. Finally, the LZCs and shifters in IEEE-754 only operate on significands, not on full words.

Expressed differently, the overhead of subnormals is due to the variable position of their rounding bit with respect to the leading 1. From this point of view, all posits are as bad as IEEE-754 subnormals.

B. Posit as a memory-only encoding

An alternative architecture to limit the latency impact of the posit encoding and decoding steps consists in keeping posits decoded in the processor: the posit format is used as a storage-only encoding. This architecture is depicted on Figure 3. The decoding and encoding blocks are placed on the memory path, while values in CPU registers are PIF-encoded.

Since multiple operations may occur before the result is written back to memory, rounding and encoding can no longer



(a) Memory read (b) Arithmetic operation (c) Memory write

Fig. 3: Architecture of a PAU using posits as a memory-only encoding, with PIF registers and PIF-to-PIF operators.

be fused. To ensure standard compliance, posit rounding must occur after each computation, in such a way that the PIF result exactly represents result of the operation defined by the posit standard. This is the role of the "UPIF inplace round" box.

In other words, in this architecture, the two conversion boxes, "posit to PIF" and "PIF to posit" must be exact, while the rounding and saturation logic must still be performed after each operation. This will still involve large shifters, however this logic is potentially cheaper than classical rounding: since the PIF is kept normalized, what must be shifted is masks and rounding bits, and these bit vectors that are simpler (more structured) than arbitrary significands. Details will be given in Section IV-C.

IV. Posit hardware detailed architecture

This section details the implementation of posit operators components provided by the MArTo library. The provided schematics aim at representing the usage of high level primitive inside operator function. As such, it should be easy for the interested reader to follow and check the library code. In case of discrepancy, the code is the reference.

A. Posit to PIF decoder

The proposed posit decoder is depicted in Figure 4.

The "LZOC + Shift" block (LZOC stands for "leading zero/one counter") counts the range bits while discarding them, resulting in a normalized fraction.

The most significant exponent bits e_h are computed out of the range count. If the leading bit is equal to s, then $e_h = -l(= \bar{l} + 1)$; else $e_h = l - 1$. An optimization is to skip the first range bit when counting, effectively computing $l' = l - 1$. Indeed, if the first range bit is equal to s, $e_h = \overline{l' + 1} + 1 = \bar{l'}$, or $e_h = l'$ otherwise. This high bit decoding method improves the state of the art by avoiding an adder to compute $-l$. The exponent least significant bits e_l are obtained by xoring with s the W_{es} first bits of the aligned fraction.

The PIF exponent e is the concatenation of e_h and e_l (or is equal to e_h if the corresponding format has $W_{\text{es}} = 0$). The decoder is slightly simplified with $W_{\text{es}} = 0$ posit formats, as it saves the XOR gates labeled $*$ on Figure 4. The PIF fraction f consists of the W_f least significant bits of the aligned fraction.

An OR reduction over the $N - 1$ rightmost bits of the posit input is used to detect both zero and NaR values, in conjunction with s.

The weight 0 significand bit i is computed out of s and the detection of zero value.

The most expensive parts of this architecture are the "OR reduce" over $N - 1$ bits to detect NaR numbers, and the combined leading zero/one counter and shifter.

It is interesting to compare this conversion to the decoding of special cases from IEEE-754 floats (which similarly must be performed on the inputs). There, one OR reduction on the exponent bits is needed to detect subnormals, another one on the significand bits is needed to detect zeros, and two similar AND reductions are needed to detect respectively NaN and infinities. The two OR reductions operate in total on the same width as the posit OR reduction, so the cost is the same. Then, subnormal management requires one LZC and one significand shifter. The latter is smaller than the full-word shifter of a posit decoder, compensating the overhead of the separate LZC.

In our combined LZOC + shifter implementation, the multiplexer at step i is driven by an AND reductions on 2^i bits[2]. The combined sizes of these AND reduction is N, again matching the AND of IEEE-754.

In summary, the decoding of one posit is very comparable to the decoding of an IEEE-754 with subnormal normalization. This will be supported by the experiments in Section VI.

B. UPIF to posit and PIF to posit

The complete UPIF to posit encoder architecture is shown in Figure 5.

First, the rounding bit is appended to the fraction, and the range is computed then prepended. The Shifter+Sticky component then simultaneously right-shifts this word and ORs the shifted-out bits into a unique sticky bit, which is then ORed to the PIF sticky bit to get the final sticky bit.

[2]Asymptotically faster implementations of LZC exist [8], but the one chosen here is better on the FPGAs used for our numerical experiments, thanks to very fast AND/OR reductions through the fast-carry logic.

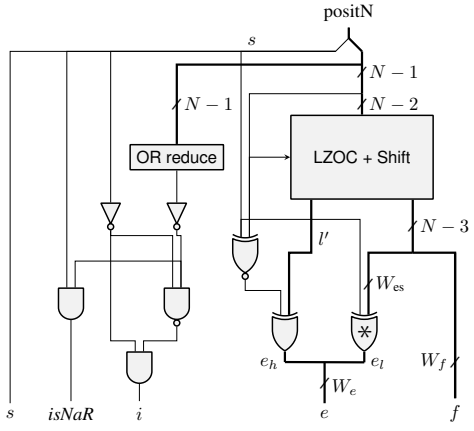


Fig. 4: Architecture of a posit to PIF decoder.

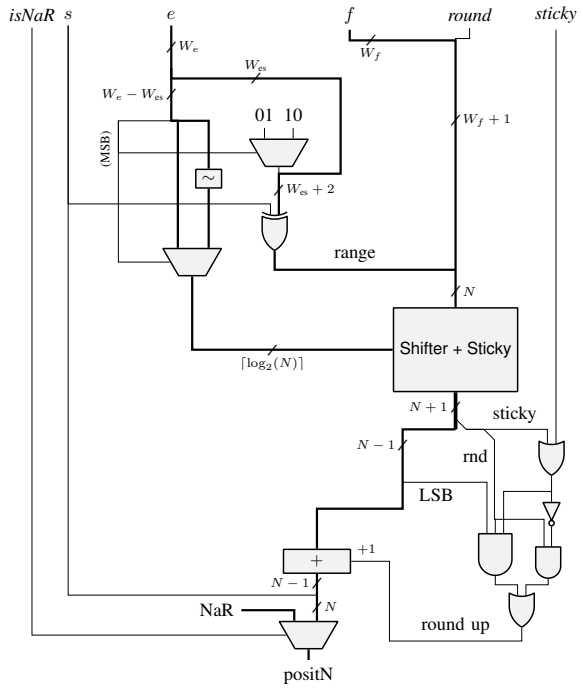


Fig. 5: Architecture of a UPIF to posit encoder.
The PIF to posit encoder is similar, with the round and sticky logic (including the final adder) removed.

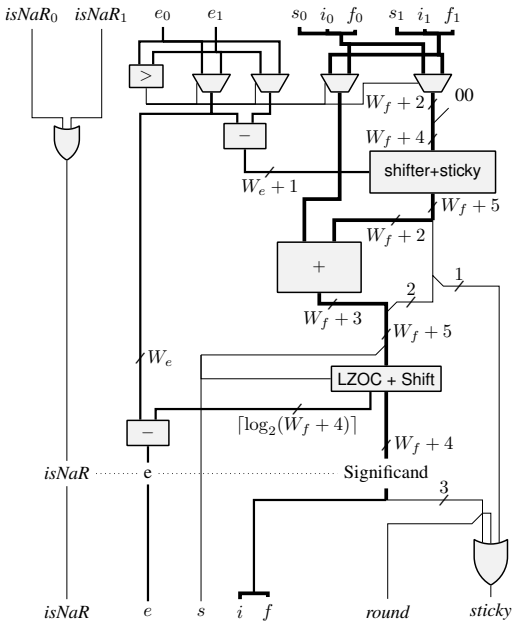


Fig. 6: Architecture of a PIF adder. Exponent comparison block denoted with "$>$" also takes the operand i and s bits to detect zero values, but wires have been omitted here for clarity.

Finally a round-up bit is computed (the right AND of Figure 5 implementing round to nearest, and the left AND the "ties to even" rule), and added to the final encoding.

The case $W_{\text{es}} = 0$ requires additional logic (not shown in Figure 5) to detect and forbid a special case of rounding up that would cause the output to round to NaR or 0.

PIF to posit conversion (as in Figure 3) is simpler: it is exact, thus avoiding the need for the rounding logic. Its complexity is delegated to the UPIF inplace rounding architecture which we detail now.

C. UPIF inplace round

When working with posit as a memory format (Figure 3), the rounding and encoding step are distinct. Indeed, the UPIF result must be rounded as if it had been converted to posit, then converted back to PIF.

One option would indeed be to shift the significand, round it, then shift it back. But then there would be no latency advantage. A cheaper alternative is to shift the rounding bit instead, while masking out significand bits that are lost to rounding.

In details, the last range bit, the e_s bits, and the fraction are concatenated, giving a posit stem of width $W_s = 1+W_{\text{es}}+W_f$. Then, four masks of W_s bits are computed out of e:

- a round mask, with only the rounding bit set,
- a guard mask, the round mask right-shifted one bit,
- a sticky mask, with all bits below the guard bit set,
- a keep mask, with all bits above the round bit (included) set.

These masks could be constructed by dedicated shifters, but our current implementation retrieves them from Look-Up Tables (LUTs) inputting the exponent. This is extremely efficient on the LUT-based FPGAs on which we conduct our quantitative evaluation. When $W_{\text{es}} = 0$, the same precaution as for the fused round/encode operator should be taken.

D. PIF floating-point operations

The architectures of the PIF adder/subtracter (Figure 6) and multiplier (Figure 7) first compute the exact result (top part of the figures) using the transposition to the PIF format of classical floating-point algorithms.

Although the adder is a single-path architecture [8], its datapath can be minimized thanks to the classical observation that large shifts in the two shifters are mutually exclusive. Indeed, the normalizing LZOC+Shift of Figure 6 will only perform a large shift in a cancellation situation, but such a situation may only occur when the absolute exponent difference is smaller than 1, which means that the first shift was a very small one. Conversely, when the first shifter performs a large shift, the rightmost part of the significand can be immediately compressed into a sticky bit, since we know that it will not be shifted back by the second LZOC+Shift. All this allows us to keep most intermediate signals on $W_f + 2$ to $W_f + 6$ bits, where previous work [10], [11] seem to use datapaths that are twice as large.

The posit multiplier shown in Figure 7 is straightforward. It performs the addition of the exponent, the signed product of the significands, then if necessary normalises the result (one-bit fraction shift and exponent update). This architecture aims at minimal area and delay. If energy efficiency is the goal,

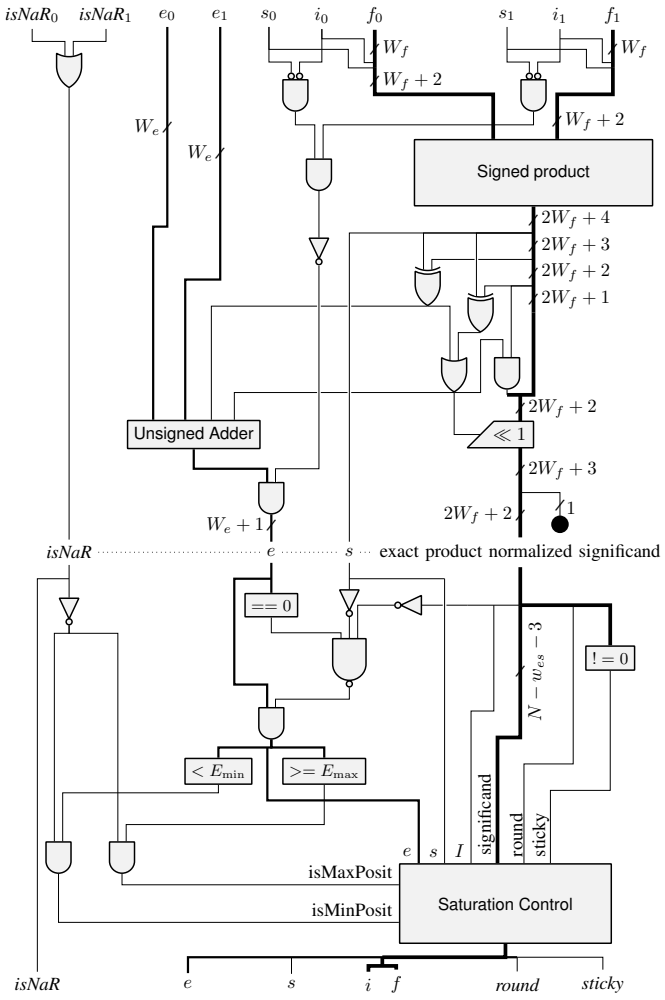


Fig. 7: Architecture of a PIF multiplier.

alternative architecture have been proposed that exploit the relation between exponent magnitude and precision to disable the computation of unneeded product LSBs [15].

The bottom part of Figures 6 and 7 normalize the exact result computed by the top parts to a PIF. For both operators, the exact significand must be realigned, correcting the exponent accordingly.

V. Hardware support for exact accumulation

The idea of an exact accumulator currently has a lot of momentum. Several existing machine learning accelerators [16], [17] already use variations of the exact accumulator to compute on IEEE-754 16-bit floating-point. Other application-specific uses have been suggested [18], [19]. For larger sizes, this could be a useful instance of "dark silicon" [20]. This trend was also anticipated with the reduction operators in the IEEE 754-2008 standard, although without the requirement of exactness.

A. Quire specification and parameters

The posit draft standard [3] currently defines fused operations as *those expressible as sums and differences of the exact product of two posits; no other fused operations are allowed.* It also specifies a binary interchange format, which consists in a fixed point number of size W_q defined by (10). In the sequel, we discuss the cost of hardware support for a quire (an exact accumulator of size W_q), but the draft standard formulation does not prevent cheaper implementations of useful fused operations such as fused multiply-add [8], complex multiplication [21], or even full convolutions for neural networks.

TABLE II: Quire bit-width parameters for standard posit.

Posit		Quire sizes				
N	W_{es}	C	W_q	W_O	W_R	W_Z
8	0	6	32	12	13	6
16	1	14	128	42	57	28
32	2	30	512	150	241	120
64	3	62	2048	558	993	496

As the parameter W_q is a storage requirement, it defines a lower bound of the area cost. Figure 8 shows a high-level functional description of a quire accumulation, and shows that there is a W_q-bit addition on the critical path from the quire to itself, which would also entail a large cycle delay. A technique that relaxes this constraint is reviewed in Section V-B.

The posit draft standard [3] specifies NaR as a special quire value. This means that this special value must be tested at each new quire operation. Instead, we add to the internal quire an isNaR flag bit, set when a NaR is added to the quire, and sticky until the quire is cleared. This isNaR bit can be encoded and decoded only when transferring quire to/from memory, however we suggest that it could even replace one of the quire carry bits in the interchange format.

In the posit context, it is natural to use two's complement for managing signs in the quire. Some implementations of Kulisch's exact accumulator seem to use a sign-magnitude representation for the accumulator [13], matching the sign-magnitude representation of IEEE floating-point. However, a two's complement representation of the accumulator is more efficient even in the IEEE-754 context [16], [22].

B. Addition of products to the quire

The posit quire is able to perform exact sums and sums of products. Therefore, the input format of the quire is defined as the output of the exact multiplier from Figure 7 (top).

To add a simple posit to the quire, it is first converted to PIF, then the PIF value is trivially cast to the same exact multiplier format (the details are skipped for brevity).

The simplest implementation of the quire addition/subtraction is depicted in Figure 8. An exact posit product fraction is shifted to the correct place to the quire format according to its exponent. A large adder then performs the addition with the previous quire value. The two's complement subtraction is performed at the cost of a XOR on the input and a carry-in to the adder, as in the posit adder/subtracter. For this the shifter must be a sign-extending one.

The simple architecture of Figure 8 can be used directly for small sizes (up to posit16). For larger sizes, the long carry propagation delay of the addition in this architecture

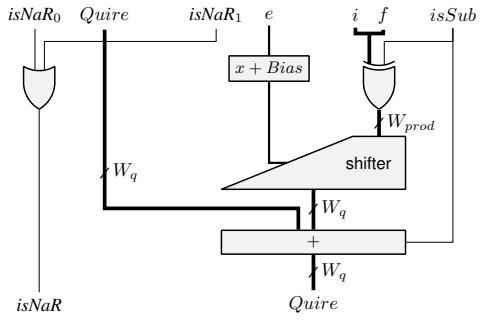


Fig. 8: Architecture of a posit quire addition/subtraction.

will restrict the maximum frequency achievable. To address this, a cost-effective solution [18], [22] is to segment the quire into smaller words (typically standard 32-bit or 64-bit words). Carry propagation is then limited to a segment, and the carries between segments are stored in registers and propagated to the next segment during the next cycle. Another point of view is that the quire is kept in a high-radix carry-save redundant format (radix is 2^{32} or 2^{64}). If such a format is used, its conversion to a non-redundant format will incur additional overhead to complete carry propagation. Some hardware can be dedicated to this, but a cheap alternative is simply to dedicate a few cycles to the completion of the carry propagation, during which the summand input to the quire is kept at zero. The number of carry-propagation cycles is $W_q/32$ for 32-bit segments. These extra cycles are amortized if the quire is used for summing large numbers of values.

Several variants of unsegmented and segmented quires will be evaluated in Section VI.

C. Conversion from quire to posit

The conversion of the quire value to a posit is divided in two steps. The quire is first converted to a UPIF value (architecture depicted in Figure 10) before the latter is encoded to a posit (Section IV-B).

There are four distinct cases to take into account when converting the quire to the UPIF:

- If the quire holds a NaR value, the result is NaR;
- If the quire value is larger in magnitude than the maximum-magnitude posit (overflow), the latter should be returned (saturation);
- If the quire value belongs to the representable posit range, it should be converted;
- If the quire value is smaller in magnitude than the minimum-magnitude non-zero posit (underflow), the latter should be returned (saturation);

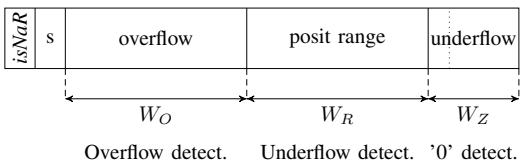


Fig. 9: The bits of a standard quire.

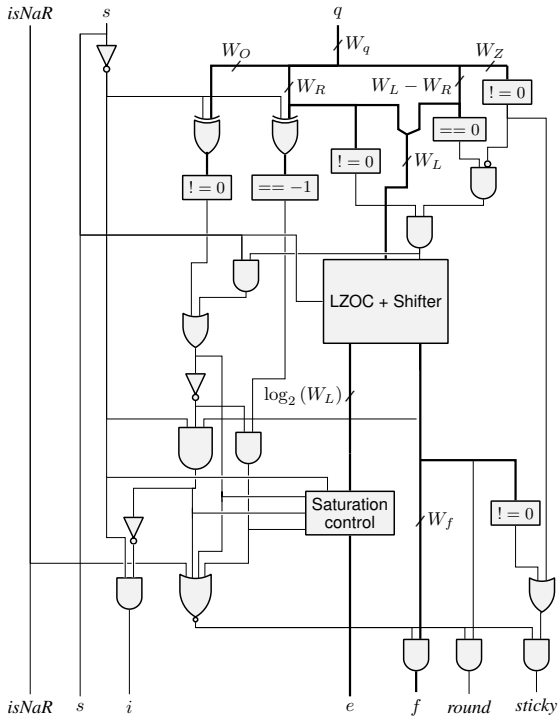


Fig. 10: Architecture of the conversion from quire to UPIF.

Figure 9 illustrates the different interesting zones of a quire. The values of the parameters appearing in this figure are determined as follows out of the parameters defined in Section II-D.

Detection of overflow consists in comparing all the bit in the overflow zone with the sign bit. If at least one differs, the posit overflows. The width of the overflow zone W_O is computed as follow:

$$W_O = Q_{\mathrm{msb}} - E_{\max} = E_{\max} + C \tag{14}$$

For quire values inside the posit range, a normalization should be performed, which uses a LZOC + shifter of input width W_R, with

$$W_R = E_{\max} - E_{\min} + 1 \tag{15}$$

Finally, to detect the difference between an underflow value and a real zero, a wide OR is performed on the underflow zone of width W_Z, with

$$W_Z = E_{\min} - Q_{\mathrm{lsb}} = E_{\max} \tag{16}$$

For standard posit sizes, using (8) yields the following formula, with the corresponding numerical values reported in Table II:

$$W_O = \frac{N^2}{8} + \frac{3N}{4} - 2 \qquad W_Z = \frac{N^2}{8} - \frac{N}{4}$$
$$W_R = \frac{N^2}{4} - \frac{N}{2} + 1$$

VI. Evaluation

This section introduces the MArTo library, compares it with the state of the art, then uses it to compare posit and floating-point operators.

Comparisons of MArTo with other works use the target and toolchain that is closest to those used in the work being compared to. All other results in this section are post place and route, obtained using Vivado HLS and Vivado 2020.1 targeting a Virtex-7 FPGA (xc7vx330tffg1157-1).

A. The MArTo library of posit and floating-point operators

The architectures described in the previous sections have been implemented using HLS-compliant templated C++ code. The library defines parameterized types such as `PositEncoding<N, WES>` and `IEEENumber<WE, WF>`, and functions to perform arithmetic operations on those types. The operators are built on top of the HINT library [23] in order to be compatible with multiple HLS tools, and to benefit from optimised primitives such as the fused LZOC+Shift of Figure 4 or the fused Shifter+sticky of Figures 5 and 6.

B. Correctness of the operators

In order to verify that the proposed architectures are correct, the following functional tests were first run in sofware:

- Exhaustive test against softposit of posit8 and posit16 addition and multiplication, both for Figure 2 and Figure 3;
- Some corner case tests of quire addition/subtraction and conversion back to posit for posit16.
- Exhaustive test for addition/product of IEEE16 against SoftFloat, for the five IEEE-754 rounding modes.

Then the VHDL file produced by the Vivado HLS compiler for a 16-bit posit adder was exhaustively tested using a VHDL simulator. Scripts and sources to reproduce this experiment are accessible from the MArTo repository.

Finally, the standard posit16 multiplier was synthesized, placed and routed for the Zynq FPGA of a Zybo board using the Vivado HLS toolchain, and the resulting FPGA circuit was exhaustively checked against softPosit executed on the ARM core of the Zynq. All these tests were successful.

C. Comparison with the state-of-the-art

As we eventually observe that posits are larger and slower than IEEE floats, it is important to be convincing that we are not using a substandard posit implementation. For this purpose, Table III gathers the results of best-effort comparisons with the current state of the art in posit hardware. It shows that MArTo is a definite improvement of this state of the art.

There is less pressure to show that the MArTo implementation of IEEE floats is efficient. A comparison with Xilinx implementation of IEEE floats is provided in Table V. There, the line labeled Xilinx Float corresponds to IP used by Vivado HLS when using the `float` and `double` data types in the HLS C++ (hence the lack of 16-bit results). This hard IP is the industry standard when using Vivado, and can be considered a state-of-the-art implementation of floating-point for Xilinx FPGAs. It supports some of the IEEE features, such as infinity and NaN encoding. However, it is not IEEE-compliant: although the memory format is that of IEEE floats, subnormals are flushed to zero to save resources.

TABLE III: Comparison with state-of-the-art hardware posit implementations [11], [10], [12], [24]

(a) Comparison with [11] for standard posit addition and product

	Op	Format	LUT	DSP	Delay (ns)
[11]	+	<16, 1>	391	0	32.4
		<32, 2>	981	0	40.0
	×	<16,1>	218	1	24.0
		<32, 2>	572	4	33.0
MArTo	+	<16, 1>	**299**	0	**24.2**
		<32, 2>	**704**	0	**33.9**
	×	<16,1>	**213**	1	**19.4**
		<32, 2>	**483**	4	**28.9**

As no sources is provided, we report as-is the figures from [11], obtained for a Zynq-7000 (xc7z020clg484-1) with Vivado 2017.4. To limit the possible effect of tool improvement on the synthesis, MArTo synthesis results have been obtained for the same part with Vivado HLS/Vivado 2018.3, the oldest version available for download at time of experimentation.

(b) Comparison with [10] on standard posit addition and product

	Op	Format	ALM	DSP	Cycles	FMax (MHz)
[10]	+	<16, 1>	∼500	0	∼49	∼550
		<32, 2>	∼1000	0	∼51	∼520
	×	<16, 1>	∼330	1	∼35	∼**600**
		<32, 2>	∼600	**1**	∼38	∼**550**
MArTo	+	<16, 1>	**274**	0	**11**	**564**
		<32, 2>	**696**	0	**17**	**562**
	×	<16, 1>	**280**	1	**15**	**600**
		<32, 2>	**452**	2	**21**	445

Synthesis reported in [10] trarget Stratix V FPGA. Results are read from a graphic plot, hence the approximate values. As there is no version of the Intel HLS toolchain that supports both Stratix V and the C++ 11 standard used in MArTo, the C++ to HDL compilation is done using Vivado HLS. The obtained HDL is then synthesised and routed for Stratix V using Quartus. Despite being baroque, this toolchain seems to give good results, except for the <32, 2> product where it lacks the knowledge of the target's DSP possible configurations. Indeed, the product is computed using a 36x36 configuration of the DSP block, where a 27x27 configuration would be faster.

(c) Comparison with [12] on posit<32,6> addition and product

	Op	LUTs	DSPs	Cycles	Delay (ns)
[12]	+	946	0	**5**	4.1
	×	854	**1**	**6**	4.4
MArTo	+	**792**	0	**5**	**3.9**
	×	**435**	2	**6**	**4.1**

MArTo synthesis have been performed using Vivado HLS/Vivado 2020.1 using part xc7vx330t-ffg1157-3. Experimental settings of [12] use the same part, but tool version is not reported.

(d) Comparison with [24] for standard posit addition and product

	Op	Format	LUT	DSP	Delay (ns)
[24]	+	<16, 1>	383	0	27.25
		<32, 1>	939	0	35.8
	×	<16,1>	**201**	1	20.9
		<32, 1>	571	4	29.2
MArTo	+	<16, 1>	**300**	0	**25.5**
		<32, 1>	**672**	0	**34.5**
	×	<16,1>	205	1	**19.2**
		<32, 1>	**472**	4	**28.8**

MArTo synthesis have been performed using Vivado HLS/Vivado 2020.1 using part xc7z020clg484-1.

TABLE IV: Synthesis results of combinatorial operators

(a) Combinatorial adder

	N	LUT	(ratio)	delay	(ratio)
posit→posit	16	312	1.33	11.1 ns	1.27
	32	647	1.49	15.8 ns	1.33
	64	1550	1.59	21.6 ns	1.35
PIF→PIF	16	237	1.01	9.7 ns	1.10
	32	562	1.29	12.9 ns	1.08
	64	1244	1.27	14.7 ns	0.92
IEEE→IEEE	16	234	1	8.8 ns	1
	32	434	1	11.9 ns	1
	64	976	1	16.0 ns	1

(b) Combinatorial multiplier

	N	LUT	(ratio)	DSP	delay	(ratio)
posit→posit	16	182	1.03	1	11.3 ns	1.39
	32	466	1.37	4	15.8 ns	1.62
	64	1213	1.58	16	21.1 ns	1.48
PIF→PIF	16	120	0.68	1	7.8 ns	0.96
	32	291	0.86	4	11.5 ns	1.17
	64	695	0.90	16	15.3 ns	1.08
IEEE→IEEE	16	176	1	1	8.1 ns	1
	32	340	1	2	9.8 ns	1
	64	768	1	9	14.3 ns	1

(c) Posit - PIF converting operators

	N	LUT	delay		N	LUT	delay
Posit to PIF	16	61	2.59 ns	PIF to posit	16	41	2.12 ns
	32	106	4.74 ns		32	98	2.50 ns
	64	278	5.52 ns		64	301	2.83

TABLE V: Synthesis results of pipelined operators

(a) Pipelined adder

	N	LUT	Reg.	DSP	cycles	delay
Posit	16	320	128	0	4	2.69 ns
	32	719	460	0	7	2.83 ns
	64	1635	1207	0	10	2.93 ns
IEEE	16	193	137	0	4	2.90 ns
	32	435	337	0	6	2.88 ns
	64	1001	880	0	10	2.99 ns
Xilinx Float	32	167	355	2	10	2.43 ns
	64	628	758	3	10	2.43 ns

(b) Pipelined multiplier

	N	LUT	Reg.	DSP	cycles	delay
Posit	16	213	80	1	4	2.85 ns
	32	443	198	4	6	2.93 ns
	64	1140	811	16	12	*4.10 ns*
IEEE	16	189	122	1	4	2.69 ns
	32	381	246	2	6	2.74 ns
	64	783	801	9	8	2.67 ns
Xilinx Float	32	82	151	3	5	2.72 ns
	64	115	494	11	10	2.75 ns

Considering that the Xilinx Float adders use DSP blocks to implement some of the shifters, the hardware costs of Xilinx adders and IEEE adders (obtained with MArTo) are really comparable. This illustrates that the overhead of subnormal handling in floating-point adders is small. Conversely, there is in Table V a very large difference in the resources used in multipliers. This demonstrates the cost of hardware subnormal handling in floating-point multipliers.

The Xilinx IP also seem to have a fixed pipelined, and do not benefit from a relaxed clock constraint to reduce the latency, hence their large latency.

Since Xilinx floats lack subnormal support, the following bases on MArTo only the posit versus IEEE comparisons.

One may wonder if this comparison doesn't also highlight some inefficiency of hardware generated using HLS tools, but recent works [25], [26] suggest that such overhead is becoming negligible for this class of applications.

D. Comparison between posit and IEEE-754 operators

Table IV compares combinatorial implementations of posits and floats of the same size on addition and multiplication. In this table, the "posit→posit" lines present results for the classical posit operators of Figure 2. The "PIF→PIF" lines presents results for the posit-compatible PIF operators that use the architecture of Figure 3b, including the inplace round component.

A first observation is that posit arithmetic is indisputably both larger and slower than IEEE-754 arithmetic. This contradicts the comparison in [11], which seems to use a very poor IEEE implementation.

As expected, the PIF-to-PIF operators are lighter and faster than the posit-to-posit ones. They still pay the price in area of a wider significand datapath (see Table I) compared to IEEE operators: for the adders, PIF-to-PIF consume more LUTs than IEEE operators; for multipliers, they consume more DSP blocks (there is a step effect due to the discrete nature of DSP blocks). Again we observe in the IEEE multipliers the logic cost of subnormal support, but we also observe a comparable cost in the PIF multiplier, essentially due to the inplace round logic. Still, the PIF to PIF operators achieve delays that are closer to those of IEEE operators than to those of posit operators, which was their main motivation.

Note that the area cost of PIF/posit conversions (altogether about half the size of a complete IEEE adder) must still be paid in a posit arithmetic unit that uses the PIF-to-PIF approach. Only its delay (altogether about half the delay of a complete IEEE adder) is avoided. However, there is another advantage in a PIF-to-PIF PAU: the hardware for these conversions is naturally shared between different operations (such sharing is also possible in principle in a posit-to-posit PAU, but then it will restrict instruction-level parallelism).

Table V compares pipelined versions of the same operators, targeting a frequency of 333 MHz (3ns cycle time), and producing one output per clock cycle. As the latency estimated by the Vivado HLS tool is usually pessimistic, the reported latencies are obtained by an automated exploration that finds the smallest pipeline depth allowing the design to run with the

TABLE VI: Synthesis results for a sum of 1000 products (U: Unsegmented, S32 and S64: Segment sizes of 32 and 64 bits).

		LUT	Reg.	DSP	cycles	delay
quire 16	U	1200	1026	1	1019	2.70 ns
	S32	978	1062	1	1021	2.68 ns
	S64	1004	958	1	1019	2.36 ns
quire 32 (512 bits)	U	5884	6235	4	1031	*3.65* ns
	S32	3641	7237	4	1040	2.89 ns
	S64	3513	5189	4	1033	2.78 ns
Kulisch 32 (559 bits)	S32	3624	7632	2	1034	2.937
	S64	3612	5165	2	1026	2.801
IEEE Float 32		840	711	2	6012	2.92 ns
IEEE Float 64		1798	1723	9	8015	*3.33* ns
Xilinx Float 32		445	544	3	6008	2.72 ns
Xilinx Float 64		809	1386	11	8013	2.70 ns

target clock period. The script performing this exploration is open source, and is also accessible from the MArTo repository for reproducibility.

There is no PIF to PIF line in this table: for this setup, the PIF to PIF approach fails to provide any latency improvement (the arithmetic operators require the same number of cycles, and sometimes require one more cycle). We therefore choose not to report these results, which we consider synthesis artifacts as they are inconsistent with the expectations and with Table IV.

E. The cost of supporting all rounding modes in IEEE

Tables IV and V report result for IEEE operators that only support round to nearest, ties to even. Another example of IEEE complexity that translates to very little hardware cost is the support of the 5 standard rounding modes. For instance, adding this support to the 32-bit adder increases its area from 434 to 458 LUTs and actually decreases the delay from 11.9 to 11.7ns (another synthesis artifact). It remains well below the posit cost.

F. Quire versus standard operations

Synthesis results for the quire are given in Table VI, where we use MArTo to write a C++ loop that performs the sum of 1000 products and return the result as a posit. They are compared to a similar loop using floating-point Kulisch accumulator, and using regular floating-point hardware.

Quire is presented in unsegmented (U) version along with two segmented versions (S32 and S64 for segments of 32 or 64 bits). For 32 bits, the unsegmented version is not able to achieve 3ns cycle time, due to the long carry propagation.

The Kulisch accumulator used here is also based on a 2's complement segmented accumulator [22, variant 3], with an IEEE-compliant final conversion to float (round to nearest, ties to even). The implementation has been validated against MFPR [27] simulations.

Unsurprisingly, the cost and performance of a posit32 quire and a Kulisch accumulator for 32 bits floats are almost identical.

TABLE VII: Detailed synthesis results of hardware posit quire

			LUT	Reg.	Cycles	Delay (ns)
posit16	Quire addition	U	618	885	4	2.576
		S32	403	585	3	1.886
		S64	444	606	3	1.984
	Carry prop.	S32	6	390	3	1.539
		S64	2*	261	2	1.651
	Quire to posit		480	166	3	2.735
posit32	Quire addition	U	3609	4986	7	3.212
		S32	1305	2265	3	2.791
		S64	1389	2276	3	2.791
	Carry prop.	S32	281	2874	8	2.851
		S64	189	2391	7	2.183
	Quire to posit		1845	1457	17	2.878

An exact accumulator consumes vastly more resources than standard operators: a factor 10 for 32-bit floats (a smaller factor for posits, but only due to the higher cost of the standard operators). Such factors should not come as a surprise: the 512 bits of the posit32 quire are indeed 18 times the 27 bits of the posit32 significand. This is the price of the accuracy of an exact accumulator.

Another advantage of exact accumulation is that it offers a latency reduction proportional to the latency of the floating-point or posit adder (here a factor 6-7). This is thanks to the fact that the accumulation loop is 1/ a fixed-point addition and 2/ exact, which offers opportunities to exploit more parallelism in its computations[16], [19].

Detailed synthesis results of the quire sub-components are given in Table VII. The *quire addition* line reports the cost for the architecture of Figure 8, including the large shifter and the fixed-point accumulation loop. This component accepts a new input every cycle. The two other lines describe the conversion of the quire result back to posit. The *carry propagation* is a loop component that adds zeroes, and is mostly merged with the *quire addition* component. However, there is an irreducible latency for the final carry propagation once the accumulation is over.

The latency overhead of the expensive conversion from quire to float or posits is easily amortized for large loops. However, it is also clear that a hardware quire will be very inefficient when used for small sequences of operations (e.g. fused multiply and add, complex arithmetic, small matrices or convolutions, etc).

VII. Conclusion

The purpose of this work is to compare the cost of hardware arithmetic operators for two competing number systems: the established IEEE-754 system and its posit challenger.

This comparison is performed thanks to a library of operators for the two systems, providing hardware description that are state-of-the art for posit, and high quality (if not state of the art) for IEEE-754. This open-source library is provided as header-only templatized C++, designed for modern High-Level Synthesis tools.

Posit-to-posit operators are shown to be significantly more expensive, both in terms of resources and delay, than IEEE operators for the same input width. For instance, addition and multiplication on 32-bit standard posits require about 50% more hardware and about 50% more delay than standard-compliant addition of binary32 floats. This overhead should be put in balance with the increased accuracy sometimes offered by posits. On the example of 32-bit formats, posits offer up to 3 extra bits of accuracy (a 11% improvement) in a limited domain of exponents, while degrading the accuracy outside of this domain due to tapered precision.

An original alternative implementation of posits is proposed: it keeps posits decoded in a wider intermediate format to avoid some of the posit encoding overhead. This alternative leads to operations that are comparable in delay to IEEE floats, but at a higher cost, all the more as it requires wider internal registers which also have a system-wide cost.

This article also provides and compares exact accumulators in both systems, without a clear advantage on a side or the other.

If there is a take-away message in this study, it would be that the indisputable complexity of the IEEE-754 standard, much attacked by posit proponents, does not necessarily translate into expensive hardware. Among the features that the posit system discards as useless, most (in particular overflow management, NaNs, and directed rounding mode) were designed to be implementable at very little cost. The only really expensive feature is subnormal support, due to rounding happening in a variable position of a bit vector. Posit arithmetic, despite the simplicity and elegance of the number system, involves such variable-position rounding, and therefore entail an overhead that is comparable in nature to subnormal support.

This work has framed baseline posit implementations. On this basis, it is possible to consider many optimizations studied for floating-point operators (such as dual-path architectures, leading zero anticipation, or various forms of hardware speculation). These optimizations will improve delay, but at the expense of area.

Before that, future work includes completing the library with missing operations, starting with division and square root.

HLS has the potential of making it very easy to study, at the application level, the impact of number systems on cost, performance, and accuracy. This is the long-term goal of the library presented here.

Acknowledgements

This work was partly funded by the Imprenum project of Agence Nationale de la Recherche. Many thanks to Orégane Desrentes for her corrections to some of the figures.

REFERENCES

[1] "IEEE standard for floating-point arithmetic," IEEE 754-2008, also ISO/IEC/IEEE 60559:2011, Aug. 2008.

[2] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[3] "Posit standard documentation," Jun. 2018, release 3.2-draft.

[4] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Performance-efficiency trade-off of low-precision numerical formats in deep neural networks," in *Next Generation Arithmetic*. ACM, 2019, pp. 3:1–3:9.

[5] P. Lindstrom, S. Lloyd, and J. Hittinger, "Universal coding of the reals: alternatives to IEEE floating point," in *Next Generation Arithmetic*. ACM, 2018.

[6] F. De Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: the good, the bad and the ugly," in *Next Generation Arithmetic*. ACM, 2019.

[7] N. Buoncristiani, S. Shah, D. Donofrio, and J. Shalf, "Evaluating the numerical stability of posit arithmetic," in *International Parallel and Distributed Processing Symposium*. IEEE, 2020, pp. 612–621.

[8] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhauser Boston, 2018.

[9] D. R. Lutz, "ARM floating-point 2019: Latency, area, power," in *26th Symposium on Computer Arithmetic*. IEEE, 2019, pp. 69–76.

[10] A. Podobas and S. Matsuoka, "Hardware implementation of POSITs and their application in FPGAs," in *International Parallel and Distributed Processing Symposium*. IEEE, 2018, pp. 138–145.

[11] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized posit arithmetic hardware generator," in *36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 334–341.

[12] M. K. Jaiswal and H. K.-H. So, "Pacogen: A hardware posit arithmetic core generator," *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019.

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

[14] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2004.

[15] H. Zhang and S. Ko, "Design of power efficient posit multiplier," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 861–865, 2020.

[16] N. Brunie, "Modified Fused Multiply and Add for exact low precision product accumulation," in *24th Symposium on Computer Arithmetic (ARITH-24)*. IEEE, Jul. 2017.

[17] J. Johnson, "Rethinking floating point for deep learning," arXiv, 1811.01721, 2018.

[18] F. de Dinechin, B. Pasca, O. Creţ, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *Field-Programmable Technologies*. IEEE, 2008, pp. 33–40.

[19] Y. Uguen, F. de Dinechin, V. Lezaud, and S. Derrien, "Application-specific arithmetic in high-level synthesis tools," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 1, 2020.

[20] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Design Automation Conference*. ACM, 2012.

[21] H. H. Saleh and E. E. Swartzlander, "A floating-point fused dot-product unit," in *International Conference on Computer Design (ICCD)*, 2008, pp. 426–431.

[22] Y. Uguen and F. de Dinechin, "Design-space exploration for the Kulisch accumulator ," Mar. 2017, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01488916

[23] L. Forget, Y. Uguen, F. de Dinechin, and D. Thomas, "A type-safe arbitrary precision arithmetic portability layer for HLS tools," in *Highly Efficient Accelerators and Reconfigurable Technologies*, Jun. 2019.

[24] F. Xiao, F. Liang, B. Wu, J. Liang, S. Cheng, and G. Zhang, "Posit arithmetic hardware implementations with the minimum cost divider and square root," *Electronics*, vol. 9, no. 10, 2020.

[25] S. Bansal, H. Hsiao, T. Czajkowski, and J. H. Anderson, "High-level synthesis of software-customizable floating-point cores," in *Design, Automation & Test in Europe*. IEEE, 2018, pp. 37–42.

[26] D. Thomas, "Templatised soft floating-point for high-level synthesis," in *27th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2019.

[27] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, 2007.