# Evaluating the hardware cost of the posit number system

Yohann Uguen, Luc Forget, Florent de Dinechin

# Evaluating the hardware cost of the posit number system

Yohann Uguen
*Univ Lyon, INSA Lyon, Inria, CITI*
*F-69621 Villeurbanne, France*
yohann.uguen@insa-lyon.fr

Luc Forget
*Univ Lyon, INSA Lyon, Inria, CITI*
*F-69621 Villeurbanne, France*
luc.forget@insa-lyon.fr

Florent de Dinechin
*Univ Lyon, INSA Lyon, Inria, CITI*
*F-69621 Villeurbanne, France*
florent.de-dinechin@insa-lyon.fr

***Abstract*—The posit number system is proposed as a replacement of IEEE floating-point numbers. It is a floating-point system that trades exponent bits for significand bits, depending on the magnitude of the numbers. Thus, it provides more precision for numbers around 1, at the expense of lower precision for very large or very small numbers. Several works have demonstrated that this trade-off can improve the accuracy of applications. However, the variable-length exponent and significand encoding impacts the hardware cost of posit arithmetic. The objective of the present work is to enable application-level evaluations of the posit system that include performance and resource consumption.**

**To this purpose, this article introduces an open-source hardware implementation of the posit number system, in the form of a C++ templatized library compatible with Vivado HLS. This library currently implements addition, subtraction and multiplication for custom-size posits. In addition, the posit standard also mandates the presence of the "quire", a large accumulator able to perform exact sums of products. The proposed library includes the first open-source parameterized hardware quire.**

**This library is shown to improve the state-of-the-art of posit implementations in terms of latency and resource consumption. Still, standard 32 bits posit adders and multipliers are found to be much larger and slower than the corresponding floating-point operators. The cost of the posit 32 quire is shown to be comparable to that of a Kulisch accumulator for 32 bits floating-point.**

## I. Introduction

Most machine implementations of real numbers rely on floating-point arithmetic. The ease-of-use of floating-point, which explains its popularity, hides complex hardware whose behaviour is specified by the IEEE-754 standard [1].

The posit number system (described in details in [2]) is an emerging machine representation of real numbers that aims at replacing IEEE-754 floating-point. The first posit claim is that floating-point is an inefficient representation. When the exponent can be encoded on only a few bits, the rest of the bits should be used to extend the precision. The second claim, adopted from Kulisch [3], is that the sum of many products is a pervasive operation, justifying specific hardware to compute it exactly. To this purpose, the draft posit standard [4] mandates a *quire*, a variant of the exact Kulisch accumulator [3] for the posit number system.

Most current evaluations of posits in applications are performed through software simulation [5], [6], [7], [8]. The C/C++ SoftPosit library [1] (among others [2]) implements the latest posit standard and allows for direct comparison with floating-point numbers in terms of accuracy.

However, the hardware cost of posits is not yet completely known. Hardware posit adders and multipliers have been written in HDL [9], [10] or using Intel OpenCL SDK compliant templatized C++ operators [11]. Using posits as a storage format by decoding/encoding from/to a large enough IEEE floating-point format as also been studied in [5]. Posits have been evaluated on applications such as machine learning [5], [6] or matrix multiply [7]. Among these works, only [5] is open-source and partially supports the quire, but only for 8-bit posits. [11] and [9] are parametric designs but are not open-source and do not support the quire. The present work, although similar in spirit, refines the architectures in [11], attempting to use the same datapath optimization tricks that are used in the floating-point operators it compares to [12]. Conversely, [9] compares a posit implementation to a floating-point implementation that is 3x larger than the state-of-the-art.

The present work improves the implementation of posit hardware with respect to all the previous works, and enables a comparison with state-of-the-art floating-point. It is parametric, open-source, and it is the first implementation to include a standard-compliant, parametric quire. As the quire is the posit incarnation of the exact Kulisch accumulator for IEEE floating-point, an implementation of the latter is provided for good measure.

The proposed implementation is a templatized C++ library compliant with Vivado HLS. It currently offers standalone posit adders, subtracters and multipliers, with overloading of the C++ operators $+$, $-$ and $\star$ for posit datatypes. Alternatively, the quire can add or subtract posits, or posit products, without rounding error. This open-source library[3] is built on a custom internal representation and extensible to other operators. The longer-term objective is really to make it possible for designers to easily switch an HLS design between floating-point and posit arithmetic, in order to compare their respective accuracy/cost/performance trade-offs.

Section II introduces in more details the posit number

[1] gitlab.com/cerlane/SoftPosit
[2] posithub.org/docs/PDS/PositEffortsSurvey.html as of march 6, 2019
[3] gitlab.inria.fr/lforget/marto

system, the algorithms for decoding and encoding them, and the datapath parameters entailed by these algorithms. Section III provides details on the architectural improvements implemented in the proposed library. Section IV compares the performance and cost of the proposed posit operators, first to the state of the art, then to floating-point operators. It also evaluates the quire in accumulation loops against IEEE floating-point and custom floating-point Kulisch accumulators.

## II. Posits

The posit number system [2] is a floating-point encoding scheme with tapered precision. A posit format is defined by its size in bits ($N$) and its exponent field width ($w_{es}$), which are the two parameters of the proposed templatized implementation.

| $S$ | Regime | | | $es$ | | $F$ | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

Fig. 1: Posit decomposition example ($N = 8$, $w_{es} = 2$).

### A. Decoding the posit

The value of Figure 1 will be used as an illustrative example of how posits work.

The first bit $S$ of the posit encodes its sign. Here the value is positive as $S = 0$. The exponent $E$ of the number is split in two parts. The first part is computed out of the (variable-size) regime field, defined by a sequence of $l$ identical bits ended by the opposite bit. The encoded range $k$ is $-l$ if the bits of this sequence are equal to $S$, otherwise $l-1$. In this example, the sequence consists in two ones: $l = 2$, therefore $k = 1$. The $w_{es}$ following bits are xored with $S$ to obtain the lower exponents bits $es$: the exponent $E$ is the concatenation of $k$ and $es$. In our example, $E = 101$ as $es = 01$.

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 can be defined as:

$$2^E \times (I.F - 2 \times S) \tag{1}$$

The value represented by the example 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 $es$. In this case, the bits shifted out are assumed to be zeros.

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. There is no special encodings for infinity: posit arithmetic saturates instead.

### B. Posit bounds and sizes

Due to the run length encoding of the range, posits with low magnitude exponents have more significand bits. The maximum precision $w_F$ is obtained for the minimum length of the regime (2), therefore

$$w_F = N - (3 + w_{es})$$

On the other hand, the maximal exponent is obtained when the regime running length is $N - 1$. In this case, all the $es$ and $F$ bits are pushed out by the regime. Hence the maximum exponent value is $E_{Max} = (N-2)2^{w_{es}}$. The number of bits needed to store the exponent in two's complement format is therefore

$$w_E = 1 + \lceil \log_2 ((N-2)2^{w_{es}}) \rceil = 1 + w_{es} + \lceil \log_2(N-2) \rceil$$

The $w_{es}$ parameter allows trading between the range of the format and its precision. The posit standard [4] defines four formats with an encoding size $N$ of 8, 16, 32 and 64 respectively. These formats are used for evaluation in this paper, although the library is fully parameterized in $N$ and $w_{es}$. The exponent field size $w_{es}$ of these formats follows the relation $w_{es} = \log_2(N) - 3$.

A posit-compliant environment must also provide a *quire*. This latter allows for the exact accumulation of posit products. It is based on the floating-point Kulisch accumulator. For the standard formats, product magnitudes range from $2^{-\frac{N^2-2N}{4}}$ to $2^{\frac{N^2-2N}{4}}$. Hence, $\frac{N^2}{2} - N + 1$ bits are required to store any such product in fixed-point representation. 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. The addition of $N - 2$ carry bits and one sign bit fulfil that goal, hence the width of standard format quires is

$$w_q = \frac{N^2}{2}$$

The different sizes and bounds for standard posit formats are reported in Table I.

TABLE I: Dimensions and bounds of standard posits ($w_{es} = \log_2(N) - 3$).

| N | $w_{es}$ | $w_E$ | $w_F$ | $E_{Max}$ | $w_q$ | $w_{pif}$ |
|---|---|---|---|---|---|---|
| 8 | 0 | 4 | 5 | 6 | 32 | 14 |
| 16 | 1 | 6 | 12 | 28 | 128 | 23 |
| 32 | 2 | 8 | 27 | 120 | 512 | 40 |
| 64 | 3 | 10 | 58 | 496 | 2048 | 73 |

The next section introduces a custom internal representation for posits based on previously shown sizes. This internal representation is used inside further detailed arithmetic operators.

## III. Architecture

The variable-length fields of the posit formats are not well suited to efficient computation on bit-parallel hardware. As all previous implementations, we first convert posits to a more hardware-friendly representation. A contribution of this work is to formally define this intermediate format.

### A. Posit intermediate format

The *posit intermediate format* (PIF) is a custom floating-point format used to represent with fixed size fields a posit value. Its main difference with standard floating-point is that the significand is stored in two's complement just like the posit significand. This simplifies decoding, but also slightly simplifies the addition of two posits.

The significand is composed of three fields $S$, $I$ and $F$, where $S$ is a sign bit, $I$ is the explicit leading bit of the posit significand, and $F$ is its fraction field, on $w_F$ bits in order to accommodate the most accurate posits of the format (less accurate ones are right-padded with zeroes). For the example of Figure 1, $S = 0$, $I = 1$ and $F = 010$ ($w_F = 3$ so the posit fraction is padded with one zero in this case).

The exponent is stored as the offset from posit minimum exponent, on $w_E$ bits. This is similar to the biased exponents of IEEE floats, and motivated by the same reasons: it simplifies the critical path of the operators, at the cost of small additions in the decoding/encoding of posits, whose latency is hidden by the longer latency of significand processing.

Posit numbers with maximum magnitude exponents have their fraction bits completely pushed out ($F = 0$). For them, Equation 1 becomes

$$\begin{cases} 2^E \times 1, & \text{for positive numbers} \\ -2 \times 2^E = -2^{E+1}, & \text{for negative numbers} \end{cases}$$

Hence, the minimal exponent expressed in *posit intermediate format* is for $-2^{-E_{Max}}$. In this case, in order to verify $E+1 = -E_{Max}$, the exponent value is $E = -E_{Max} - 1$. This leads to a bias value $Bias = (N-2)2^{w_{es}} + 1$.

Finally, three extra bits are added to the format. The *isNaR* bit is used to signal NaR. It avoids the necessity of checking for NaR in arithmetic operators. The Round and Sticky bits capture the necessary and sufficient information that must be kept after an operation on PIF values to correctly round the result back to posit.

To summarize, a *posit intermediate format* contains the following fields:

- A NaR flag $isNaR$ on 1 bit
- A sign $S$ on 1 bit
- An exponent $E$ on $w_E$ bits
- An implicit bit $I$ on 1 bit
- A significand $F$ on $w_F$ bits
- A round bit $round$ on 1 bit
- A sticky bit $sticky$ on 1 bit

The total width of the posit intermediate format is therefore $w_{pif} = w_F + w_E + 5$ bits. Posit intermediate format sizes for standard posit formats are reported in Table I as $w_{pif}$.

### B. Posit to PIF decoder

The proposed posit decoder is described in Figure 2.

The exponent of the posit is the combination of $es$ and $k$, which is computed from the run-length $l$ of the leading bit. Indeed, if the leading bit is 0, then $k = -l$ ($= \bar{l}+1$); if it is 1, then $k = l - 1$. By skipping a bit at the start of the sequence,

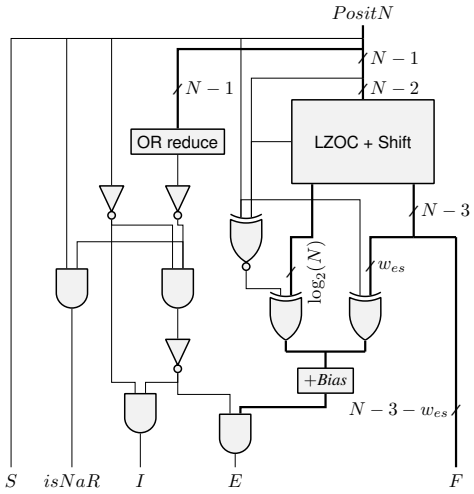


Fig. 2: Architecture of a posit decoder.

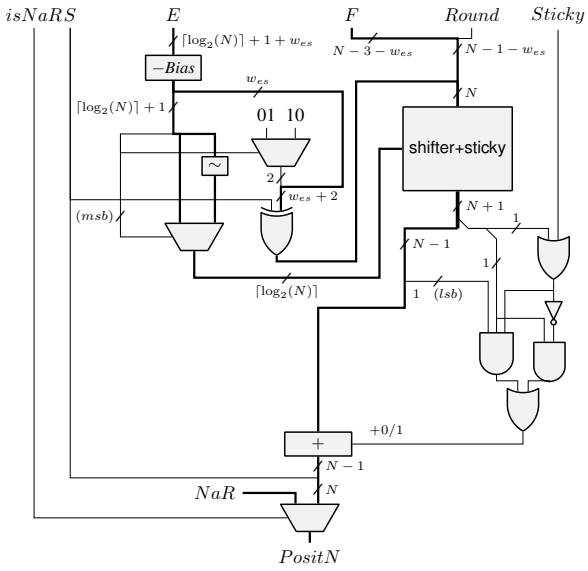


Fig. 3: Architecture of a posit encoder.

the count returns $l' = l - 1$. Therefore $k = \overline{l'+1} + 1$, hence $k = \bar{l'}$ if the leading bit is 0 or $k = l'$ if the leading bit is 1. The same method can be applied for negative numbers by computing $k = l'$ when the leading bit is 1 and $k = \bar{l'}$ when the leading bit is 0. This method is different from the literature and allows for saving an addition when computing $-l$.

The most expensive part of this architecture are (a) the `OR reduce` over $N-1$ bits to detect NaR numbers and (b) the leading zero or one count (`LZOC + Shift`) that consumes the regime bits while aligning the significand. The $+Bias$ aligns the exponents to simplify following operators. This decoding cannot be compared to an IEEE floating-point equivalent as no decoding is needed.

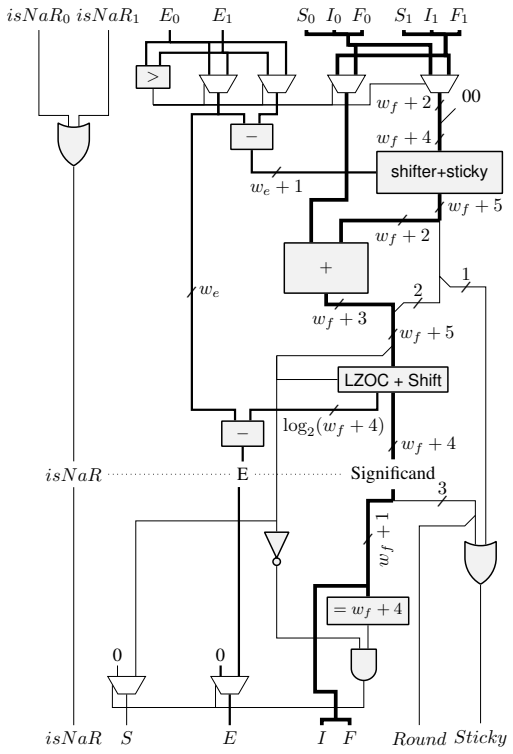


Fig. 4: Architecture of a PIF adder.

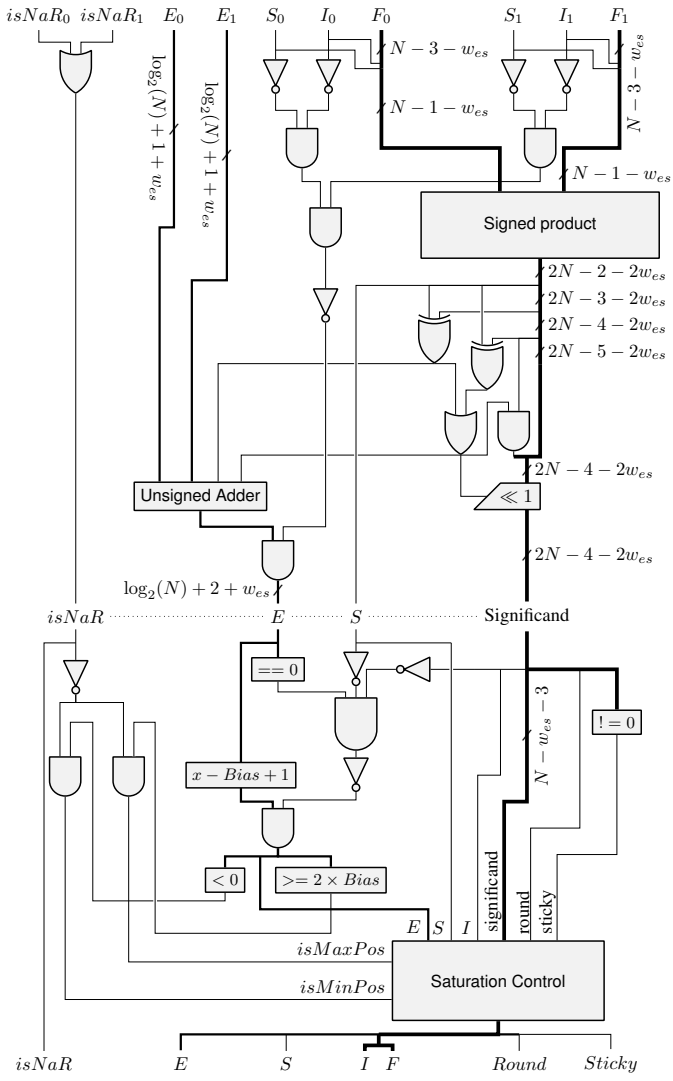


Fig. 5: Architecture of a PIF multiplier.

### C. PIF to posit encoder

Due to the variable-length encoding of posits, the position to which a PIF value must be rounded is known only when performing this conversion. The Round and Sticky bits carry synthetic information about the bits of the infinitely accurate result beyond the $F$ bits, but the encoder (depicted in Figure 3) still embeds quite some logic.

The fraction is first shifted to include the regime bits and $es$. Once shifted, the first $N-1$ bits represent the unrounded posit without sign. The remaining bits of the shifted fraction are used to extract the actual round bit and compute the final sticky bit. This information is used to compute the rounding to the nearest with tie to even.

### D. PIF adder/subtracter and multiplier

The architectures of the PIF adder/subtracter (Figure 4) and multiplier (Figure 5) 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 [12], 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 4 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 [11], [9] seem to use datapaths that are twice as large.

The bottom part of Figures 4 and 5 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.

### E. 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 5 (top).

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

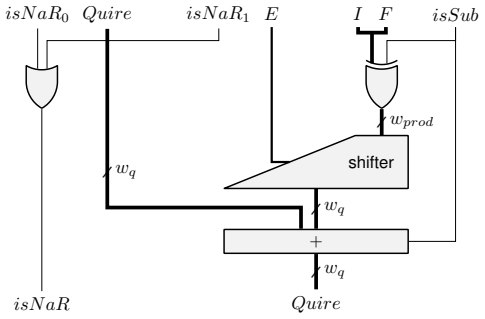


Fig. 6: Architecture of a posit quire addition/subtraction.

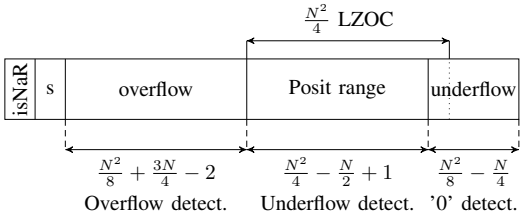


Fig. 7: Quire conversion to *posit intermediate format*.

The posit standard [4] specifies NaR as a special quire value. Testing this special value at each new quire operation is then expensive. Instead, this work proposes to add a flag bit that signals that the value held in the quire is NaR. This bit is set when NaR is added to the quire and stays set until the end of the computation. This extra bit can replace one of the quire carry bits. A slightly more expensive alternative would be to encode and decode NaR value when transferring quire to/from memory.

The proposed quire architecture is depicted in Figure 6.

*1) Addition of products to the quire:* The simplest implementation of the quire addition/subtraction is depicted in Figure 6 where the quire data structure is as depicted in Figure 7. 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 subtraction is performed at very little cost using the same method as in the posit adder/subtracter.

The long carry propagation delay of the addition in this architecture will restrict the maximum frequency achievable. To address this, a solution is to segment the quire [13]. The impact of this choice on cost and performance is evaluated in Section IV.

*2) 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 PIF value (architecture depicted in Figure 8) before the latter is encoded to a posit (Section III-A).

## IV. Evaluation

All the designs presented here have been tested exhaustively for 8-bit and 16-bit standard posits against the reference Soft-

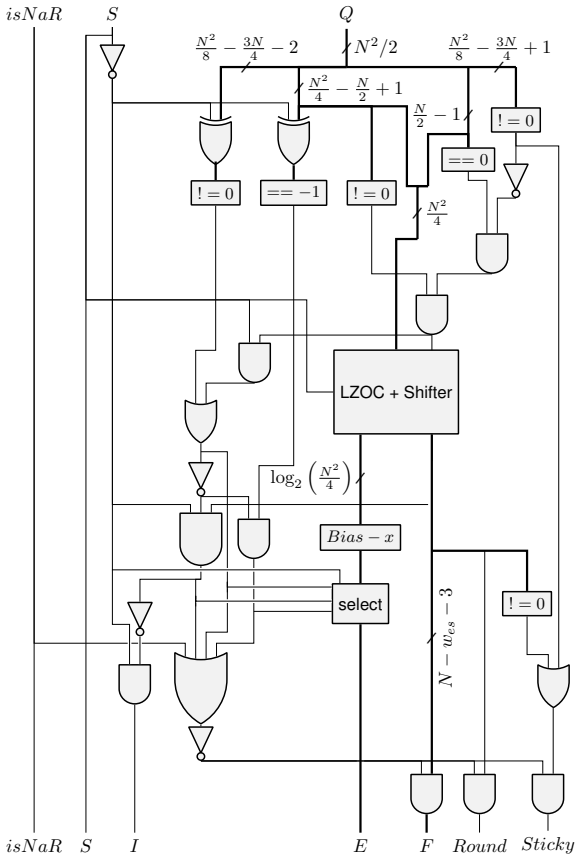


Fig. 8: Architecture of the conversion from the quire to a *posit intermediate format*.

Posit implementation. They have also been tested extensively for other sizes.

The presented posit architectures are first shown to improve the state of the art in IV-A. This ensures fair comparisons, with state-of-the-art floating-point operators in IV-B, and of exact acumulators in IV-C.

### A. Comparison with the state-of-the-art

Results are reported in [9] for a Xilinx Zynq-7000, and in [10] for a Virtex 7. We chose for our comparison the simpler setting of [9] (Zynq-7000, no pipeline) and synthesized both our library and that of [10] [4] for this setting. Results are given in Table II. The present work improves 32 bits operators in all metrics. In the 16 bits cases, the delays are always improved. There is only one case (the 16 bits multiplier) where [9] has better resource consumption. Still, in this case, the area.time ($AT$) and $AT^2$ of the proposed approach are better.

In [11], results are given for a Stratix V FPGA. Their adder operator is actually an adder/subtracter. The corresponding comparison is in Table III. For this table, we used VivadoHLS 2018.3 to generate VHDL files which were then synthesized using Quartus 18.1. This worked without problem for our

[4] source code accessed on June 26th, 2019 from https://github.com/manish-kj/Posit-HDL-Arithmetic

TABLE II: Comparison with [9] and [10] targeting Zynq (combinatorial components)

(a) Posit Adder

| | N | LUT | DSP | Delay (ns) |
|---|---|---|---|---|
| [9] | 16 | 320 | 0 | 23 |
| | 32 | 981 | 0 | 40 |
| [10] | 16 | 460 | 0 | 21 |
| | 32 | 1115 | 0 | 29 |
| This work | 16 | 320 | 0 | 21 |
| | 32 | 745 | 0 | 24 |

(b) Posit Multiplier

| | N | LUTs | DSPs | Delay (ns) |
|---|---|---|---|---|
| [9] | 16 | 218 | 1 | 24 |
| | 32 | 572 | 4 | 33 |
| [10] | 16 | 271 | 1 | 19 |
| | 32 | 648 | 4 | 27 |
| This work | 16 | 253 | 1 | 18 |
| | 32 | 469 | 4 | 27 |

TABLE III: Comparison with [11] targeting Stratix V

(a) Posit Add/Sub

| | N | ALM | DSP | Cycles | FMax (MHz) |
|---|---|---|---|---|---|
| [11] | 16 | ∼500 | 0 | ∼49 | ∼550 |
| | 32 | ∼1000 | 0 | ∼51 | ∼520 |
| This work | 16 | 327 | 0 | 19 | 584 |
| | 32 | 636 | 0 | 24 | 539 |

(b) Posit Multiplier

| | N | ALM | DSP | Cycles | FMax (MHz) |
|---|---|---|---|---|---|
| [11] | 16 | ∼330 | 1 | ∼35 | ∼600 |
| | 32 | ∼600 | 1 | ∼38 | ∼550 |
| This work | 16 | 199 | 1 | 16 | 600 |
| | 32 | 452 | 2 | 21 | 445 |

designs, at the cost of sub-optimal quality of results. We report approximate data for [11] since it is read from graphical plots.

In general, the operators developed in this work require fewer resources and have shorter critical paths. This is mainly due to rigorous implementation of each component (shifters, lzoc, etc.) and improvements over existing architectures (addition saved in the encoder, contraction of the adder addition similar to state-of-the-art floating-point adders, etc.). There is a discrepancy in the 32-bit multiplication in Table III: the 29x29 multiplier is implemented as two DSP blocks in 36x36 mode [14] in our case, while it is implemented in [11] as one DSP block in 27x27 mode, plus some logic. The slower frequency of our library in this case is not surprising, as we synthesize for an Intel FPGA the VHDL generated for a Xilinx FPGA. It will be solved in the near future by a portable HLS library instead of the current Vivado-specific one [15].

### *B. Comparison with floating-point operators*

All the remaining results given in this work are obtained using Vivado HLS and Vivado 2018.3 targeting 3ns delay for a Kintex 7 FPGA (xc7k160tfbg484-1). Table IV compares posits and floats of the same size on addition and multiplication.

On the addition side, we have a perfectly fair comparison between the results labelled "Posit" and the results labelled "IEEE": this latter line describes a fully compliant IEEE adder, with subnormal support, implemented with the same care as the posit operators and using the same parametric subcomponents. We observe that the posit adder is almost twice as large and twice as slow as the IEEE adder. Some of it is due to the variable-length field encoding and decoding (Figures 2 and 3). Some of it is due to the slightly extended internal precision of posits.

We also give results for two other mainstream floating-point implementations. The line labelled Float corresponds to IP used by Vivado HLS when using the `float` and `double` C datatypes (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. However, it is not IEEE-compliant: although the memory format is that of IEEE floats, subnormals are flushed to zero to save resources. The line labelled Soft FP reports a recent HLS-oriented templatized floating-point library [16] which is not IEEE-compliant either.

The comparison on multiplication is less definitive, as it lacks a fully compliant IEEE multiplier implementation with subnormal support. Still, the posit multiplier is much more expensive and slower than the industry standard floating-point. Supporting subnormal adds an overhead roughly corresponding to one posit decoder (one LZOC and one shifter), and is not expected to overturn the game.

In absolute terms, there may be some overhead due to HLS tools, but recent works [17], [16], as well as the the comparison between the "Soft FP" and the "Float" hard IP, suggests that it is becoming negligible.

### *C. Quire evaluation*

The synthesis results for the quire are given in Table V where we perform 1000 sums of product and return the result as a posit. They are compared to a floating-point Kulisch accumulator and to regular floating-point hardware. Kulisch and quire are presented in unsegmented (U) version along with two segmented versions (S32 and S64 for segments of 32 or 64 bits). The unsegmented versions are not able to achieve 3ns due to the long carry propagation. The Kulisch accumulator used in this paper is similar to the 2's complement Kulisch 3 variant architecture from [13], but with a final conversion to float that is IEEE-compliant (round to nearest, ties to even). The implementation has been validated against MFPR [18] simulations. Classically, using an exact accumulator consumes

TABLE IV: Synthesis results of posit and IEEE floating-point adders and multipliers.

(a) Adder

| | N | LUT | Reg. | DSP | Cycles | Delay (ns) |
|---|---|---|---|---|---|---|
| Posit | 16 | 383 | 358 | 0 | 18 | 2.702 |
| | 32 | 738 | 811 | 0 | 22 | 2.659 |
| | 64 | 1660 | 2579 | 0 | 33 | 2.609 |
| IEEE | 16 | 216 | 205 | 0 | 12 | 2.331 |
| | 32 | 425 | 375 | 0 | 14 | 2.690 |
| | 64 | 918 | 792 | 0 | 17 | 2.737 |
| Float | 32 | 341 | 467 | 0 | 9 | 2.529 |
| | 64 | 641 | 1098 | 0 | 11 | 2.562 |
| Soft FP [16] | 16 | 205 | 228 | 0 | 10 | 2.453 |
| | 32 | 416 | 527 | 0 | 13 | 2.239 |
| | 64 | 1237 | 1545 | 0 | 19 | 2.702 |

(b) Multiplier

| | N | LUT | Reg. | DSP | Cycles | Delay (ns) |
|---|---|---|---|---|---|---|
| Posit | 16 | 269 | 292 | 1 | 16 | 2.361 |
| | 32 | 544 | 710 | 4 | 21 | 2.421 |
| | 64 | 1501 | 2410 | 16 | 42 | 2.816 |
| Float | 32 | 80 | 193 | 3 | 7 | 2.201 |
| | 64 | 196 | 636 | 11 | 17 | 2.568 |
| Soft FP [16] | 16 | 38 | 127 | 1 | 8 | 1.825 |
| | 32 | 67 | 228 | 2 | 9 | 2.193 |
| | 64 | 259 | 651 | 9 | 10 | 3.299 |

TABLE V: 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 (ns) |
|---|---|---|---|---|---|---|
| Quire 16 | U | 1409 | 1763 | 1 | 1028 | 3.215 |
| | S32 | 1239 | 1431 | 1 | 1031 | 2.643 |
| | S64 | 1185 | 1555 | 1 | 1030 | 2.756 |
| Quire 32 (512 bits) | U | 5068 | 6256 | 4 | 1040 | 8.850 |
| | S32 | 4394 | 4779 | 4 | 1055 | 2.854 |
| | S64 | 3783 | 4564 | 4 | 1047 | 2.961 |
| Kulisch 32 (559 bits) | S32 | 4446 | 5290 | 2 | 1050 | 2.875 |
| | S64 | 4365 | 5276 | 2 | 1041 | 2.854 |
| Float 32 | | 460 | 806 | 3 | 10011 | 2.676 |
| Float 64 | | 892 | 1999 | 11 | 12021 | 2.737 |

roughly 10x more resources but reduces the latency by 10x, while making the computation exact.

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

Detailed synthesis results of all the subcomponents are given in Table VI. The accumulation loop is the *Quire addition* component. It can be pipelined with an initiation interval of one cycle. During synthesis, the *Carry propagation* component will be merged with the *Quire addition*, reducing its cost. However, there is an irreducible latency for the final carry propagation once the accumulation is over.

TABLE VI: Detailed synthesis results of hardware posit quire (U: Unsegmented, S32 and S64: Segment sizes of 32 and 64 bits)

(a) Posit 16

| | | LUT | Reg. | DSP | Cycles | Delay (ns) |
|---|---|---|---|---|---|---|
| Decoding | | 59 | 64 | 0 | 4 | 1.986 |
| Product | | 50 | 113 | 1 | 7 | 1.832 |
| Quire addition | U | 499 | 1078 | 0 | 5 | 2.681 |
| | S32 | 459 | 357 | 0 | 4 | 2.628 |
| | S64 | 432 | 543 | 0 | 5 | 2.437 |
| Carry prop. | S32 | 108 | 137 | 0 | 5 | 2.548 |
| | S64 | 71 | 134 | 0 | 3 | 2.545 |
| Quire to posit | | 560 | 480 | 0 | 10 | 2.609 |

(b) Posit 32

| | | LUT | Reg. | DSP | Cycles | Delay (ns) |
|---|---|---|---|---|---|---|
| Decoding | | 137 | 142 | 0 | 5 | 2.158 |
| Product | | 93 | 277 | 4 | 10 | 2.143 |
| Quire addition | U | 2384 | 4712 | 0 | 7 | 5.050 |
| | S32 | 1424 | 984 | 0 | 5 | 2.679 |
| | S64 | 1148 | 1066 | 0 | 4 | 2.488 |
| Carry prop. | S32 | 519 | 535 | 0 | 17 | 2.549 |
| | S64 | 480 | 531 | 0 | 9 | 2.945 |
| Quire to posit | | 2534 | 2439 | 0 | 17 | 2.878 |

The *Decoding* and *Product* components can be pushed out of the accumulation loop and pipelined to feed the *Quire addition* component. Conversely, carries must be propagated before the conversion *Quire to posit* can occur. Therefore, the total latency of the design is approximately the sum of the combined *Decoding*, *Product* and *Quire addition* pipeline depths; the *Quire addition* initiation interval, times the number of products to add; the *Carry propagation* pipeline depth; and the *Quire to posit* pipeline depth.

This latency is amortized for large sums. However, it has to be take into account when considering the quire to add a few values, e.g. to emulate an FMA or a fused dot product.

## V. Conclusion

The purpose of this work is to enable evaluating the cost of converting a floating-point application to posits. To that end, a Vivado HLS templatized C++ library implements the posit number system, including the quire. This library has been implemented with the same care as state-of-the-art floating-point, with several improvements in the datapath that translate to greatly improved performance compared to previous posit implementations. Posit hardware is found to be more expensive than float hardware. However, for applications where posits are more accurate than floats of the same size, the real use case should be to vary the parameters, so as to find which

arithmetic provides the required application-level accuracy at the minimal cost. We hope that this work enables such studies.

Future work includes completing the library with missing operations (division, square root), and making it portable to a broader range of HLS tools.

In the context where one can vary the parameters of the posits to evaluate the cost/accuracy/performance ratio, it would be fair to also vary the parameters of the floats. A few extra significand bits to a floating-point format can make up with the golden zone accuracy of the equivalent posit at a lower resource cost and latency. Furthermore, a Kulisch accumulator can also be used to perform exact sum-of-products. In such a context, the accumulator could also be tailored to the application to save latency and resources ([19], [20]).

*Acknowledgements*

This work was partly funded by the Imprenum project of Agence Nationale de la Recherche. Many thanks to Orégane Desrentes for reviewing this article and correcting 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] U. Kulisch, *Computer arithmetic and validity: theory, implementation, and applications*. Walter de Gruyter, 2013.

[4] P. W. Group, "Posit standard documentation," Jun. 2018, release 3.2-draft.

[5] J. Johnson, "Rethinking floating point for deep learning," *arXiv preprint arXiv:1811.01721*, 2018.

[6] 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 *Proceedings of the Conference for Next Generation Arithmetic*. ACM, 2019, pp. 3:1–3:9.

[7] J. Chen, Z. Al-Ars, and H. Hofstee, "A matrix-multiply unit for posits in reconfigurable logic leveraging (open)CAPI (online)," 03 2018, pp. 1–5.

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

[9] 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*. IEEE, 2018, pp. 334–341.

[10] 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.

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

[12] 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.

[13] Y. Uguen and F. De Dinechin, "Design-space exploration for the Kulisch accumulator (Online)," 2017.

[14] *Stratix-V Device Handbook*, Altera Corporation, 2013.

[15] L. Forget, Y. Uguen, F. de Dinechin, and D. Thomas, "A type-safe arbitrary precision arithmetic portability layer for HLS tools," in *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, Nagasaki, Japan, Jun. 2019, pp. 1–6.

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

[17] 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.

[18] 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, p. 13, 2007.

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

[20] Y. Uguen, F. de Dinechin, and S. Derrien, "Bridging high-level synthesis and application-specific arithmetic: The case study of floating-point summations," in *27th International Conference on Field Programmable Logic and Applications*. IEEE, 2017, pp. 1–8.