Comparing Different Decodings for Posit Arithmetic

Raul Murillo(✉), David Mallasén, Alberto A. Del Barrio, and Guillermo Botella

Complutense University of Madrid, 28040 Madrid, Spain
{ramuri01,dmallase,abarriog,gbotella}@ucm.es

Abstract. Posit arithmetic has caught the attention of the research community as one of the most promising alternatives to the IEEE 754 standard for floating-point arithmetic. However, the recentness of the posit format makes its hardware less mature and thus more expensive than the floating-point hardware. Most approaches proposed so far decode posit numbers in a similar manner as classical floats. Recently, a novel decoding approach has been proposed, which in contrast with the previous one, considers negative posits to have a negative fraction. In this paper, we present a generic implementation for the latter and offer comparisons of posit addition and multiplication units based on both schemes. ASIC synthesis reveals that this alternative approach enables a faster way to perform operations while reducing the area, power and energy of the functional units. What is more, the proposed posit operators are shown to improve the state-of-the-art of implementations in terms of area, power and energy consumption.

Keywords: Computer arithmetic · Posit · Decoding · Addition · Multiplication

1 Introduction

Historically, most scientific applications have been built on top of the IEEE 754 standard for floating-point arithmetic [10], which has been for decades the format for representing real numbers in computers. Nevertheless, the IEEE 754 format possesses some problems that are inherent to its construction, such as rounding, reproducibility, the existence of signed zero, the denormalized numbers or the wasted patterns for indicating *Not a Number* (NaN) exceptions [6]. All in all, IEEE 754 is far from being perfect, as different CPUs may produce different results, and all these special cases must be dynamically checked, which increases the hardware cost of IEEE 754 units.

Recently, several computer arithmetic encodings and formats, such as the High-Precision Anchored (HPA) numbers from ARM, the Hybrid 8-bit Floating Point (HFP8) format from IBM, bfloat16, and many more have been considered as an alternative to IEEE 754-2019 compliant arithmetic [7], which has also

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
J. Gustafson and V. Dimitrov (Eds.): CoNGA 2022, LNCS 13253, pp. 84–99, 2022.
https://doi.org/10.1007/978-3-031-09779-9_6

recently included a 16-bit IEEE 754 version. Nonetheless, the appearance of the disruptive posit arithmetic [8] in 2017 has shaken the board. While the aforementioned approaches, except for the half-precision IEEE 754, are vendor-specific, posits aim to be standard. This novel way of representing reals mitigates and even solves the previously mentioned IEEE 754 drawbacks. Posits only possess one rounding mode, and there are just two special cases to check (zero and infinity). Also, posits are ordered in the real projective line, so comparisons are basically as the integer ones, and even conceive the use of fused operations in order to avoid losing precision. This is done by avoiding rounding of individual operations and accumulating the partial results in a large register called *quire*, which can even speed up computations with a large number of operands [15]. Another interesting property of posits is their tapered precision, that is, they are more accurate when their magnitude is in the proximity of zero, that is, their absolute value is near 1. These last properties have attracted a lot of attention from the community because they suit Deep Learning applications [3,9,12,17]. These applications leverage the multiply-accumulate (MAC) operations in order to accelerate the computation of matrix and dot products [2,23]. Furthermore, the numbers employed are typically normalized and thus fall in the proximities of zero. According to some authors, 32-bit posits can provide up to 4 orders of magnitude improvement in terms of accuracy [14,19] when comparing with the equivalent single-precision floating-point format. Nevertheless, this accuracy enhancement comes at a cost. The quire occupies a vast portion of the resulting posit functional unit [14,19,22].

Since 2017, several designs have appeared which implement individual [4,11,16] and fused [3,19,24] posit operators. While their implementations are different, either because of the functionality or due to the design, the unpacking/decoding of posits is common to all of them. This paper presents a study about the different ways of decoding posit numbers in literature, which directly affects how these decoding units unpack posit operands and that could also impact some other portions of the functional unit itself. Results show that decoding posits in a different manner to the classical one inspired by floating-point arithmetic can substantially reduce the hardware resources used by functional units. In addition, this work presents an implementation of posit functional units that follows the alternative decoding scheme aforementioned. The proposed implementation outperforms state-of-the-art designs of posit adders and multipliers in terms of performance and hardware requirements.

The rest of the paper is organized as follows: Section 2 introduces the necessary background about the posit format, and details the two different approaches for decoding posits that have been proposed so far. Section 3 describes the different components of fundamental posit arithmetic units (adders and multipliers), as well as the existing design differences when using each of the decoding approaches. The performance and resource utilization of both approaches are compared in Sect. 4, which shows ASIC synthesis results for different components and arithmetic units from the literature. Finally, Sect. 5 concludes this work.

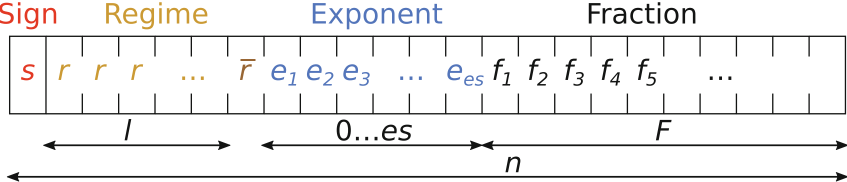

Fig. 1. Posit$\langle n, es\rangle$ binary encoding. The variable-length regime field may cause the exponent to be encoded with less than es bits, even with no bits if the regime is wide enough. The same occurs with the fraction.

2 Posit Arithmetic

A posit format is defined as a tuple $\langle n, es\rangle$, where n is the total bitwidth of the posits and es is the maximum number of bits reserved for the exponent field. As Fig. 1 shows, posit numbers are encoded with four fields: a sign bit (s), several bits that encode the regime value (k), up to es bits for the unsigned exponent (e), and the remaining bits for the unsigned fraction (f). The regime is a sequence of l identical bits r finished with a negated bit $\bar{r}$ that encodes an extra scaling factor. As this field does not have a fixed length, some exponent or fraction bits might not fit in the n-bit string, so 0 would be assigned to them. The variable length of this field allows posit arithmetic to have more fraction bits for values close to ± 1 (which increases the accuracy within that range), or to have less fraction bits for the sake of more exponent bits for values with large or small magnitudes (increasing this way the range of representable values). This is known as *tapered accuracy*, and contrasts with the constant accuracy that IEEE 754 floats present, due to the fixed length of the exponent and fraction fields, as can be seen in Fig. 2 (here, the left part of the IEEE floating-point format corresponds to the gradual underflow that subnormal numbers produce).

Posit arithmetic only considers two special cases: zero, that is represented with all bits equal to 0, and *Not a Real* (NaR) exception, represented by all the bits except the sign bit equal to 0. The rest of the bit patterns are used to represent a different real value. However, at the time of writing this paper, two main different ways of understanding how posit bit strings represent real values have been proposed: using the sign-magnitude format, as floating-point numbers, or considering posits in two's complement notation. While both approaches are equivalent from a mathematical sense (i.e. the same bit patterns represent the same values, regardless of the approach), they present implementation differences that should be considered when implementing such arithmetic format in a physical device. Other alternative interpretations of posits are discussed in [13].

2.1 Sign-Magnitude Posit Decoding

Posit arithmetic is a floating-point format for representing real numbers. Thus, the numerical value X of a normal posit datum was initially defined in [8] by (1)

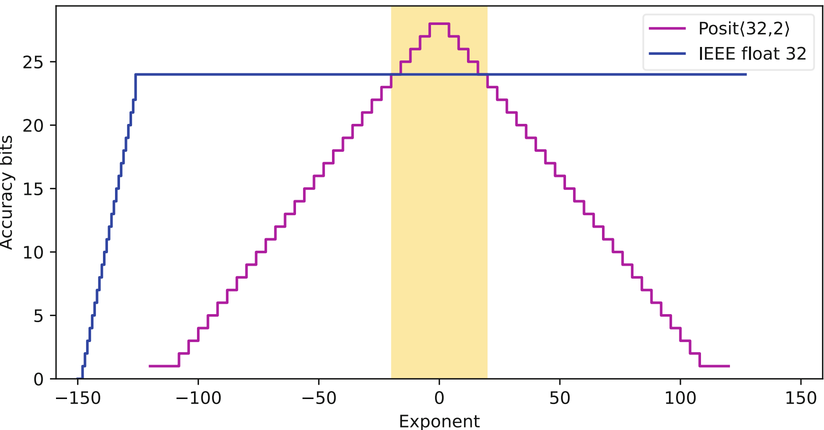

Fig. 2. Accuracy binary digits for 32-bit formats

$$X = (-1)^s \times (useed)^k \times 2^e \times (1 + f), \quad (1)$$

where $useed = 2^{2^{es}}$, e is the integer encoded by the exponent field, k is $l-1$ when $r = 1$, or $-l$ when $r = 0$, and f is the normalized fraction (this is, the value encoded by the fraction bits divided by 2^F, so $0 \leq f < 1$). Under this decoding approach, if a value is negative (when the sign bit is 1), its two's complement is computed before extracting the regime, exponent, and fraction, so values k, e and f in Eq. (1) are always considered from the absolute value of the posit.

The main differences with the standard floating-point format are the utilization of an unsigned and unbiased exponent, the hidden bit of the significand is always "1" (no subnormal numbers are considered), and the existence of the variable-length regime field. However, notice that this decoding is quite similar to the one for classical floating-point numbers: it deals with a sign bit, a signed exponent (regime and exponent can be gathered in a single factor) and a significand with a hidden bit. As a consequence, the circuit design for both arithmetic formats would be similar too. In fact, this float-like decoding scheme is the one used by most of the posit arithmetic units from the literature [4,11,16], as well as by the approximate posit units proposed so far [18,20].

Apparently, trying to implement posits by first forcing them to look more like floats and then converting back does not seem optimal, and the community is still in the early stages of discovering new decodings and circuit shortcuts that leverage this recently proposed format.

2.2 Two's Complement Posit Decoding

The previous decoding scheme of posit numbers deals with negative numbers in a similar manner as signed integers do. From a hardware perspective, converting

posits to their absolute value before decoding them adds extra area and performance overhead, specially when compared with IEEE 754 floats. To address this issue, Isaac Yonemoto, co-author of [8], proposes a different way of decoding posit numbers: for negative values, the most significant digit of the significand is treated as "−2" instead of "1". The rest of the fields remain the same, but under this approach there is no need to compute the two's complement (absolute value) of each negative posit. This is consistent with the way posits were initially intended, as a mapping of the signed (two's complement) integers to the projective reals. The value X of a posit number is now given by (2)

$$X = (useed)^{\tilde{k}} \times 2^{\tilde{e}} \times (1 - 3s + f), \tag{2}$$

where again $useed = 2^{2^{es}}$, but now $\tilde{e}$ is equal to e XOR-ed bitwise with s and $\tilde{k}$ is $-l$ when $r = s$, or $l - 1$ otherwise.

Theorem 1. *For any given posit bit string that encodes a number other than zero or NaR, the expressions* (1) *and* (2) *are equivalent.*

Proof. When the sign bit is 0 (i.e. the bit string encodes a positive number), it is trivial that both expressions evaluate the same.

On the other hand, the case when $s = 1$ requires more attention. First, note that in such a case, two's complement of the bit string must be computed before evaluating expression (1). Hence, all the bits at the left of the rightmost "1" are inverted. Let us consider three cases, depending on which field that bit belongs to.

(i) If the rightmost "1" bit belongs to the fraction field, the fraction $f \neq 0$. Hence, it is evident that $k = \tilde{k}$ and $e = \tilde{e}$, since k and e are obtained from the inverted regime and exponent bits in the original bit string, respectively. It remains to check whether the significands from expressions (1) and (2) have an opposite value. But recall that the fraction from expression (1) is two's complemented, and due to the fact that $0 \leq f < 1$, the two's complement of f is $\tilde{f} = 1 - f$. From this last property it follows that $(1 + \tilde{f}) = -(-2 + f)$.

(ii) If the rightmost "1" bit belongs to the exponent field, then $f = 0$ and $e \neq 0$. For the same reason as in the previous case, $k = \tilde{k}$. But now the exponent field for expression (1) is two's complemented rather than inverted, so we have that $e = \tilde{e} + 1$. However, since $f = 0$, the significand in expression (1) evaluates 1, while expression (2) evaluates $(1 - 3s + f)$ as -2, which compensates the difference in the exponents.

(iii) If the rightmost "1" bit belongs to the regime, then $e = f = 0$. Also, it should be noted that such a bit corresponds to the last (inverted) regime bit (in case the regime is a sequence of 0's) or to the bit immediately preceding the inverted one (when the regime is a sequence of 1's). In both cases, taking two's complement for computing (1) reduces the length of the regime field in 1, so it follows that $k = \tilde{k} + 1$. In addition, note that in this situation $e = 0$, while $\tilde{e} = 2^{es} - 1$. Nevertheless, a similar situation as in the previous case

occurs with the fraction: the significand from expression (2) is evaluated as -2, which compensates the difference of exponents previously mentioned. Note that the multiplicands of both expressions are powers of 2, so it suffices to check that both expressions have the same exponent. Indeed: $(2^{es})^{\tilde{k}} + \tilde{e} + 1 = (2^{es})^{k-1} + (2^{es} - 1) + 1 = (2^{es})^{k}$. □

When dealing with the hardware implementation, the significand of (1) can be represented in fixed-point with a single (hidden) bit that always takes the value "1". On the other hand, when considering expression (2), the significand $(1 - 3s + f)$ belongs to the interval $[-2, -1)$ for negative posits and to $[1, 2)$ for positive ones, so such signed fixed-point representation requires two integer (or hidden) bits that depend on the sign of the posit. More precisely, in this case, negative posits prepend "10" to the fraction bits as the 2's complement hidden bits, and positive posits prepend "01". Note how this contrasts with the unsigned fixed-point representation of the significand in the floating-point and classical sign-magnitude posit decoding formats. Therefore, this approach eliminates complexity in the decoding and encoding stages, but requires redesigning some of the logic when implementing posit operators.

There are not many works that implement this two's complement decoding approach for posit numbers. The first implementation of posit adders and multipliers based on this decoding appeared in [24], and more details about such a scheme were introduced in [7]. Also, [19] presents different energy-efficient fused posit MAC units that follow the same approach as [24].

In this paper we present a generic implementation of posit functional units based on two's complement decoding. Furthermore, we compare different state-of-the-art posit units based on both decoding schemes.

Finally, it is noteworthy that previous works have examined the effect of two's complement notations in floating-point arithmetic [1]. However, in such a case, some features or properties are lost with respect to the IEEE standard for floats. In this work we prove that both sign-magnitude and two's complement coding of posit numbers are equivalent, and therefore all properties are preserved regardless of the used approach. The impact of each decoding scheme is found on the hardware implementation, as will be discussed in Sect. 4.

3 Posit Operators

The advantage of using sign-magnitude decoding for posit numbers is that arithmetic operations can be performed in a similar way to standard floating-point ones (except for the bitwidth of the fields and exception handling). While this can leverage the already designed circuits for floating point, forcing posits to look like floats and then converting back adds some overhead to the operators. However, considering the posit significand as a signed fixed-point value eliminates the need for absolute value conversion, but requires some redesign of the arithmetic cores.

This section describes in detail and compares the design of different arithmetic operations when dealing with each posit decoding scheme.

3.1 Decoding and Encoding Stages

Unlike floating-point hardware that ignores subnormal numbers, the variable-length regime does not allow the parallel decoding of posit numbers, that is, the fraction and exponent cannot be extracted until the length of the regime is known. Thus, when implementing posit operators in hardware, it is usually necessary to extract the four fields presented in Sect. 2 (s, k, e and f, plus a flag for zero/NaR exceptions) from a compact posit number before starting the real computation, as well as packing again the resulting fields after that. The components that perform such processes are usually known as decoders and encoders, respectively, and those are the modules that present more differences in their design according to the decoding mode used.

The classical decoding scheme considers negative posits to be in two's complement. Hence, in such a case, it is necessary to first take a two's complement of the remaining bit string before decoding the regime (which is usually done with a leading ones/zeros detector), exponent and fraction bits, as detailed in Algorithm 1 (zero/NaR exception checking is omitted for the sake of clarity). Then, all the computation is performed with the absolute value of the posits, leaving aside the sign logic until the end, where it requires to take again the two's complement of the bit string according to the sign of the result. The process of encoding a posit from its different fields mainly consists of performing Algorithm 1 backwards, plus handling possible rounding and overflow/underflow situations.

Algorithm 1 Classical posit decoding algorithm

Require: $X \in \text{Posit}\langle n, es\rangle,\quad F = n - es - 3$
Ensure: $(-1)^s \times (useed)^k \times 2^e \times (1 + f) = X$
  $s \leftarrow X[n-1]$
  if $s = 1$ **then**
    $p \leftarrow \sim X[n-2:0] + 1$ ▷ Take two's complement
  else if $s = 0$ **then**
    $p \leftarrow X[n-2:0]$
  end if
  $r \leftarrow p[n-2]$
  $l \leftarrow LZOC(p)$ ▷ Count regime length
  if $r = 1$ **then**
    $k \leftarrow l - 1$
  else if $r = 0$ **then**
    $k \leftarrow -l$
  end if
  $q \leftarrow p[n-l-3:0]$ ▷ Extend with 0's to the right, if necessary
  $e \leftarrow q[F + es - 1 : F]$
  $f \leftarrow q[F-1:0]$

On the other hand, the alternative scheme proposed by Yonemoto handles both positive and negative posit numbers simultaneously, without the need of

computing the absolute value of the posits. Determining the sign of the regime's value requires checking if the posit sign bit is equal to the MSB of the regime, and the exponent value for this case requires XOR-ing es bits with the sign bit. The decoding process for this approach is described in Algorithm 2. Also, as already mentioned, handling the significand in signed fixed-point format for computation requires one extra bit for the sign, since this decoding considers the significand of positive values to be in the interval $[1, 2)$ (as in the previous decoding), or in $[-2, -1)$ when the number is negative. This approach can reduce the latency of the decoding and encoding stages, specially for larger bitwidths, since it requires XOR-ing just es bits (generally es is not greater than 3) instead of computing the two's complement of the n-bit posits as in Algorithm 1. However, the signed fixed-point significand introduces extra complexity in the core of the arithmetic operations, as will be discussed below.

Algorithm 2 Alternative posit decoding algorithm

Require: $X \in \text{Posit}\langle n, es\rangle, \quad F = n - es - 3$
Ensure: $(useed)^k \times 2^e \times (1 - 3s + f) = X$

```
s ← X[n − 1]
p ← X[n − 2 : 0]
r ← p[n − 2]
l ← LZOC(p)                                  ▷ Count regime length
if r ≠ s then
    k ← l − 1
else if r = s then
    k ← −l
end if
q ← p[n − l − 3 : 0]                         ▷ Extend with 0's to the right, if necessary
e ← q[F + es − 1 : F] ⊕ {es{s}}              ▷ Sign bit is replicated to perform XOR
f ← q[F − 1 : 0]
```

3.2 Addition

In posit arithmetic, as well as in the case of floating-point arithmetic, when performing the addition (or subtraction) of two numbers, it is necessary to shift one of the fractions so both exponents are equal. If the first exponent is smaller than the second, the first fraction is shifted to the right by a number of bits given by the absolute difference of the exponents. Otherwise, the same is done to the second fraction. Then, the aligned significands are added, and the result is normalized, so the larger exponent is adjusted if needed.

When using a classical sign-magnitude decoding approach, some extra logic is needed to handle the sign of the result. But such logic is eliminated when dealing with signed significands, since the sign of the result can be inferred from the leftmost bit of the addition of the significands, without initially comparing the magnitude of the inputs. Dealing with signed significands also avoids the need

of performing subtraction or taking two's complement when the sign of both addends differ, which makes up for using one extra bit in the addition of significands. On the other hand, normalization of the significand in two's complement deserves special attention, since it needs to count not only the leading zeroes, but the leading ones when the result is negative. However, as will be shown in Sect. 4, this does not involve hardware overhead for this particular module.

3.3 Multiplication

In a similar manner as in the case of addition, posit multiplication takes inspiration from the floating-point algorithm: both significands are multiplied and normalized, and the exponents are added together. Additionally, the result from significand multiplication must be normalized to fit in the corresponding interval, which involves shifting the fraction plus adding to the exponent the number of shifted bits.

When the significands have a single hidden bit, i.e., using the sign-magnitude posit decoding, the leftmost bit of the multiplication indicates if the resulting fraction must be shifted and 1 must be added to the exponent. Note that in this case, both multiplicands follow the expression $(1 + f) \in [1, 2)$, so the product must be in the interval $[1, 4)$. Thus, normalizing the result might require shifting one bit at most.

On the other hand, when dealing with the two's complement decoding scheme, even though the multiplication can be performed directly (just one more bit for each operand is necessary), the normalization of the result is more complex in this case. As each multiplicand $(1 - 3s + f)$ can be in the range $[-2, -1) \cup [1, 2)$, the result will fall within the range $(-4, -1) \cup [1, 4]$. In terms of fixed-point arithmetic, the operand has two integer bits, so the multiplication has four bits that represent the integer part of the number and that should be examined in the normalization process. Note that the resulting sign is also implicit in the multiplication result. However, this approach introduces one extra case that needs special attention: when the two multiplicands are equal to -2, the result is 4, which requires adding 2, rather than 1, to the exponent when normalizing the result.

Finally, note that similar considerations should be taken into account for the case of the division operation, although it is beyond the scope of this paper.

4 Hardware Evaluation

This section evaluates the hardware impact of each posit decoding scheme. In addition to standard comparison of arithmetic units, in order to provide a more fine-grained evaluation, this section compares the hardware requirements of each individual component when using each of the decodings. To achieve an accurate evaluation, all the results given in this work were generated to be purely combinational and synthesized targeting a 45 nm TSMC standard-cell library with no timing constraint and typical case parameters using Synopsys Design Compiler.

4.1 Components Evaluation

To have a better understanding of how the different ways of decoding posit numbers impact on the hardware resource utilization, we compared ASIC synthesis results for each single component of the posit operators described in Sect. 3 when implemented under each of the decoding schemes presented in this paper. For the sign-magnitude decoding, we extracted the different components from Flo-Posit [16], which includes open-source[1] designs implemented using FloPoCo [5] and requires less hardware resources than other implementations based on the same classical decoding scheme. With respect to the two's complement decoding, there are no available designs other than those proposed in [24], which consist of a C++ header library for HLS that implement a modified posit format. Thus, in order to make as fair a comparison as possible, we implemented the designs described in Sect. 3 using Yonemoto's decoding scheme and using FloPoCo as well, which allows to generate parameterized units for any number of bits and exponent size.

In order to verify the correctness of the proposed architectures, exhaustive tests for units with 16 bits or less, as well as random tests with corner cases for larger bitwidths, were performed using a VHDL simulator. The results were compared against two software libraries: the Universal number library [21], which supports arithmetic operations for any arbitrary posit configuration, and GNU MPFR, which was modified with support for posit binary representation. All these tests were successful. Then, each module (decoder, encoder, core adder and core multiplier) was synthesized separately, so the area, power, datapath delay and energy (power-delay product) could be compared in detail. Results were normalized with respect to the classical decoding scheme.

As can be seen in Fig. 3, which shows the cost of just the decoding stage rather than the cost of the whole arithmetic operation, considerable savings are obtained when decoding the posits by using Yonemoto's two's complement proposal. Under this approach, the decoder module requires about 66% of the area, 45% of power, 59% of delay and 27% of the energy than the same module implemented using the classical sign-magnitude decoding approach. Also, it is noteworthy that for many of the most common operations, like addition or multiplication, two operands need to be decoded, so this module is often duplicated.

Similar figures are obtained for the encoder module. As Fig. 4 shows, using Yonemoto's approach requires about 33% less area than the classical one, but in this case the power and delay savings are not as pronounced as for the decoder module. Nevertheless, using the alternative decoding scheme reduces energy consumption of this process by half.

As already mentioned, dealing with signed significands avoids the need of negating one of the operands when performing addition of different sign values. This is demonstrated in Fig. 5, which compares the hardware requirements of both approaches for just the logic of posit addition (without circuitry for decoding operands nor for rounding/encoding the result). The extra bit for dealing

[1] https://github.com/artecs-group/Flo-Posit/tree/6fd1776.

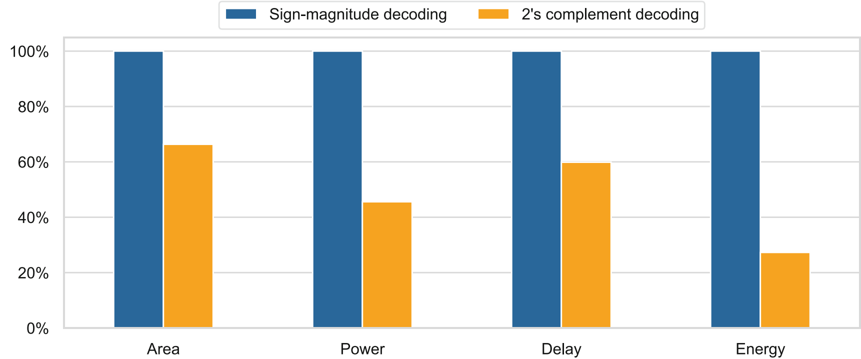

Fig. 3. Relative hardware performance metrics of Posit$\langle 32, 2\rangle$ decoder components.

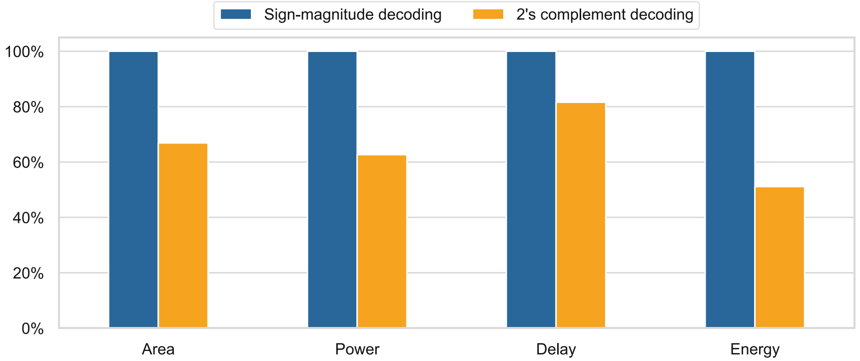

Fig. 4. Relative hardware performance metrics of Posit$\langle 32, 2\rangle$ encoder components.

with the significand in two's complement adds negligible overhead to this component, and together with the reduction of logic to handle addition of different sign operands, makes this scheme to use 91% of the area, 86% of the power and 84% of the datapath delay of the analogous component based on the classical decoding.

The case of the multiplier module is different from the previous ones. Here, handling the significands in two's complement requires one extra bit for each operand, and a total of four more bits for storing the multiplication result, when compared with the sign-magnitude approach. This translates into approximately 7% more area and power, but similar delay, as shown in Fig. 6.

4.2 Comparison with the State-of-the-Art

The components using two's complement decoding scheme seem to provide smaller and faster implementations. However, it is important to verify that whole operators follow the same trend, and that the proposed implementations are not

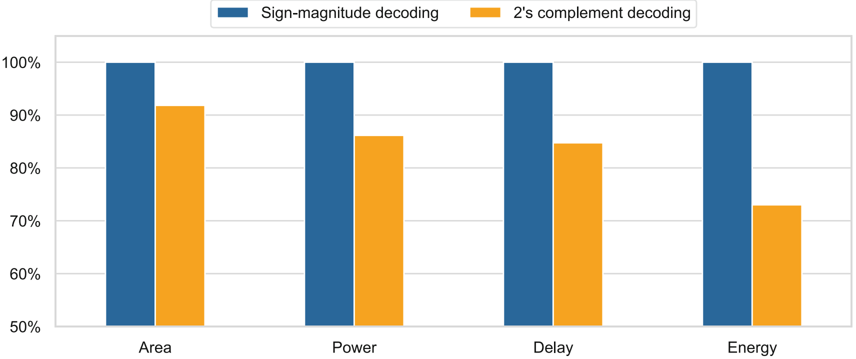

Fig. 5. Relative hardware performance metrics of Posit$\langle 32, 2\rangle$ adder components

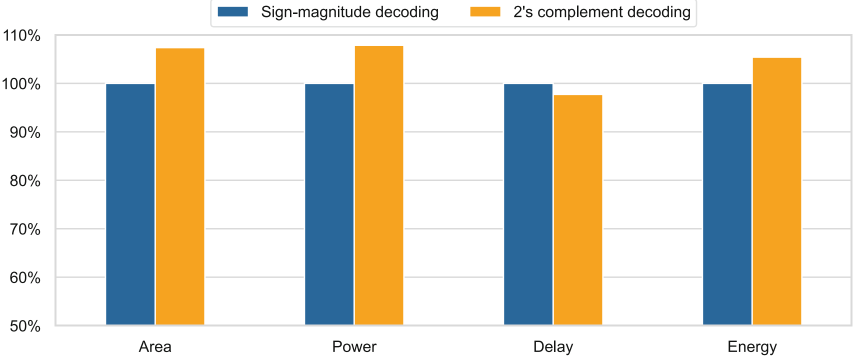

Fig. 6. Relative hardware performance metrics of Posit$\langle 32, 2\rangle$ multiplier components.

sub-optimal. For this purpose, three different implementations of posit operators from the state-of-the-art were compared: PACoGen [11][2] and Flo-Posit [16], which use the sign-magnitude decoding scheme given by (1), and MArTo [24][3], which is based in the decoding scheme proposed by Yonemoto with slight differences. In particular, the designs presented in [24] perform conversion to/from the so-called *posit intermediate format* (PIF), a custom floating-point format that stores the significand in two's complement (just like the approach evaluated in this work) and takes an exponent (including the regime) which is biased with respect to the minimum exponent, as in the IEEE 754 standard. The PIF simplifies the critical path of the operators, at the cost of small additions in the decoding/encoding of posits. Also, the proposed implementation that has been discussed in the preceding section was added to the comparison, so there are two implementations for each posit decoding approach. Unlike MArTo, the proposed operators implement the logic in pure posit format, without conversion of posits

[2] https://github.com/manish-kj/PACoGen/tree/5f6572c.
[3] https://gitlab.inria.fr/lforget/marto/tree/2f053a56.

to a float-like format. For a more detailed analysis of how the different decoding approaches scale according to the number of bits, posit operators for $\langle 8, 1 \rangle$, $\langle 16, 1 \rangle$ and $\langle 32, 2 \rangle$ formats were synthesized. Despite the fact that Posit$\langle 8, 0 \rangle$ is a more common format in the literature, the PACoGen core generator does not allow to generate posit operators with no exponent bits ($es = 0$), so Posit$\langle 8, 1 \rangle$ is selected instead for a fair comparison. Finally, note that MArTo is an HLS-compliant C++ library, rather than a RTL-based implementation as the rest of the libraries used in this work. Thus, the C++ to HDL compilation of MArTo operators is done using Vitis HLS 2021.1 with default options.

Synthesis results for the adder and multiplier units are shown in Fig. 7 and Fig. 8, respectively. Both cases present a clear gap between the designs based on the sign-magnitude posit decoding (PACoGen and Flo-Posit) and the ones using the two's complement scheme (MArTo and the one proposed in this paper), specially for the power-delay product (energy) results. Except for the adder delay, Flo-Posit designs present better figures than the analogous designs from PACoGen, which seems to be far from an optimal implementation. Exactly the same occurs for the proposed designs with respect to those from MArTo library, but in this case the difference between both implementations is much smaller. This might be due to the fact that both MArTo and the proposed units follow quite similar designs but with certain differences in the implementation, since the former designs are generated by a commercial HLS tool, while the latter are directly designed at the RTL level. In accordance with these results, we took

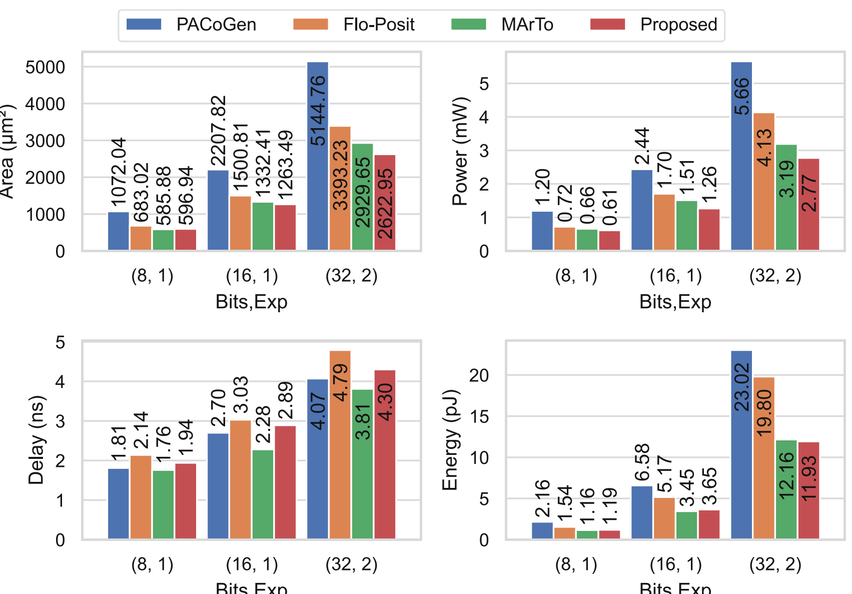

Fig. 7. Synthesis results for different Posit$\langle n, es \rangle$ adder designs.

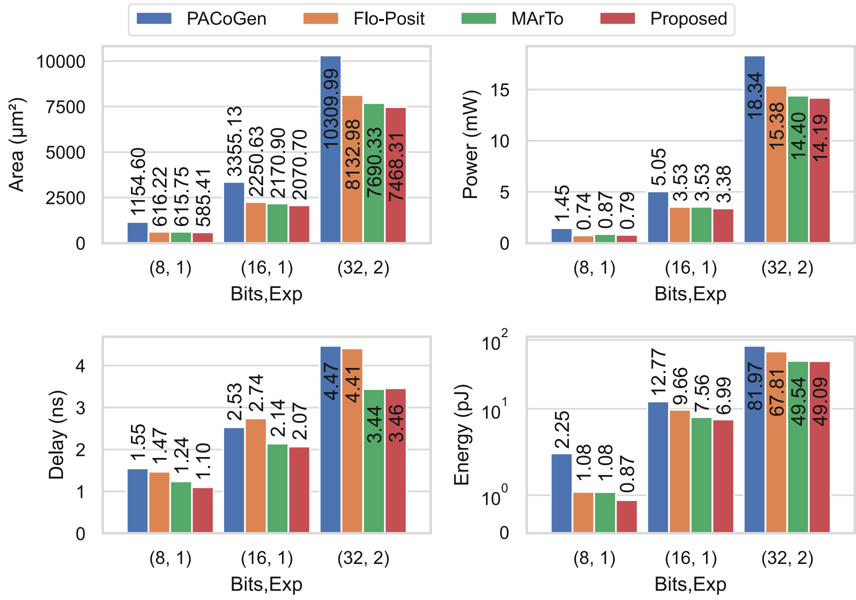

Fig. 8. Synthesis results for different Posit$\langle n, es\rangle$ multiplier designs.

the best designs of each decoding scheme (Flo-Posit and the proposed one) as a baseline for detailed comparison.

In the case of posit adders, the greatest resource savings are obtained for 32-bit operators: when using the alternative decoding, the area is reduced by 22.70%, the power by 32.90%, the delay by 10.22% and the energy by 39.77%.

On the other hand, and in line with the results shown previously, the alternative two's complement decoding scheme for posit multiplier units also presents less resource utilization when compared with the classical float-like scheme, but these savings are not as pronounced as in the case of posit addition. As can be seen in Fig. 8, the 32-bit multipliers based on Yonemoto's decoding approach reduce area, power, datapath delay and energy by 8.17%, 7.72%, 21.54%, and 27.60%, respectively.

5 Conclusions

Multiple designs of posit arithmetic units have been proposed since the appearance of this alternative format. While those units might present several optimizations for area or energy efficiency, one of the main design differences is the way posit strings are decoded. The first works on posit arithmetic presented a sign-magnitude decoding scheme similar to floating-point arithmetic, with a sign bit, a signed exponent and a fraction with a hidden bit equal to 1. But recently, a two's complement decoding for posits proposed by I. Yonemoto, which considers

that the hidden bit means −2 for negative posits, seems to provide more efficient implementations of functional units, at the cost of a more complex circuit design.

This paper aims to shed some light on the different ways of decoding posit numbers presented in literature so far, and how such decodings affect the hardware resources of posit operators. To that purpose, we implemented custom-size posit adder and multiplier units following the alternative decoding scheme proposed by Yonemoto. Synthesis evaluations show that posit units based on classical float-like decoding schemes require generally more hardware resources than analogous units using the two's complement decoding. In addition, the proposed units are shown to improve previous posit implementations in terms of area and energy consumption.

Acknowledgments. The authors wish to acknowledge Isaac Yonemoto for the feedback and explanations on his insight of posit decoding. This work was supported by a 2020 Leonardo Grant for Researchers and Cultural Creators, from BBVA Foundation, whose id is PR2003_20/01, by the EU(FEDER) and the Spanish MINECO under grant RTI2018-093684-B-I00, and by the CM under grant S2018/TCS-4423.

References

1. Boldo, S., Daumas, M.: Properties of two's complement floating point notations. Int. J. Softw. Tools Technol. Transf., 237–246 (2003). https://doi.org/10.1007/s10009-003-0120-y
2. Camus, V., Enz, C., Verhelst, M.: Survey of precision-scalable multiply-accumulate units for neural-network processing. In: 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), pp. 57–61. IEEE, March 2019. https://doi.org/10.1109/AICAS.2019.8771610
3. Carmichael, Z., Langroudi, H.F., Khazanov, C., Lillie, J., Gustafson, J.L., Kudithipudi, D.: Deep positron: a deep neural network using the posit number system. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1421–1426. IEEE, March 2019. https://doi.org/10.23919/DATE.2019.8715262
4. Chaurasiya, R., et al.: Parameterized posit arithmetic hardware generator. In: 2018 IEEE 36th International Conference on Computer Design (ICCD), pp. 334–341. IEEE (2018). https://doi.org/10.1109/ICCD.2018.00057
5. de Dinechin, F., Pasca, B.: Designing custom arithmetic data paths with FloPoCo. IEEE Des. Test Comput. **28**(4), 18–27 (2011). https://doi.org/10.1109/MDT.2011.44
6. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. (CSUR) **23**(1), 5–48 (1991). https://doi.org/10.1145/103162.103163
7. Guntoro, A., et al.: Next generation arithmetic for edge computing. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1357–1365. IEEE (2020). https://doi.org/10.23919/DATE48585.2020.9116196
8. Gustafson, J.L., Yonemoto, I.T.: Beating floating point at its own game: posit arithmetic. Supercomput. Frontiers Innov. **4**(2), 71–86 (2017). https://doi.org/10.14529/jsfi170206

9. Ho, N.m., Nguyen, D.T., Silva, H.D., Gustafson, J.L., Wong, W.F., Chang, I.J.: Posit arithmetic for the training and deployment of generative adversarial networks. In: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1350–1355. IEEE, February 2021. https://doi.org/10.23919/DATE51398.2021.9473933
10. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84 (2019). https://doi.org/10.1109/IEEESTD.2019.8766229
11. Jaiswal, M.K., So, H.K.: PACoGen: a hardware posit arithmetic core generator. IEEE Access **7**, 74586–74601 (2019). https://doi.org/10.1109/ACCESS.2019.2920936
12. Johnson, J.: Rethinking floating point for deep learning. arXiv e-prints, November 2018
13. Lindstrom, P.: Variable-radix coding of the reals. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH), pp. 111–116. IEEE, June 2020. https://doi.org/10.1109/ARITH48897.2020.00024
14. Mallasén, D., Murillo, R., Del Barrio, A.A., Botella, G., Piñuel, L., Prieto, M.: PERCIVAL: open-source posit RISC-V core with quire capability, pp. 1–11, November 2021
15. Muller, J.M., et al.: Handbook of Floating-Point Arithmetic. Birkhäuser Boston, Boston (2010). https://doi.org/10.1007/978-0-8176-4705-6
16. Murillo, R., Del Barrio, A.A., Botella, G.: Customized posit adders and multipliers using the FloPoCo core generator. In: 2020 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5. IEEE, October 2020. https://doi.org/10.1109/iscas45731.2020.9180771
17. Murillo, R., Del Barrio, A.A., Botella, G.: Deep PeNSieve: a deep learning framework based on the posit number system. Digit. Sig. Process. Rev. J. **102**, 102762 (2020). https://doi.org/10.1016/j.dsp.2020.102762
18. Murillo, R., Del Barrio Garcia, A.A., Botella, G., Kim, M.S., Kim, H., Bagherzadeh, N.: PLAM: a posit logarithm-approximate multiplier. IEEE Trans. Emerging Top. Comput., 1–7 (2021). https://doi.org/10.1109/TETC.2021.3109127
19. Murillo, R., Mallasén, D., Del Barrio, A.A., Botella, G.: Energy-efficient MAC units for fused posit arithmetic. In: 2021 IEEE 39th International Conference on Computer Design (ICCD), pp. 138–145. IEEE, October 2021. https://doi.org/10.1109/ICCD53106.2021.00032
20. Norris, C.J., Kim, S.: An approximate and iterative posit multiplier architecture for FPGAs. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5. IEEE, May 2021. https://doi.org/10.1109/ISCAS51556.2021.9401158
21. Omtzigt, T., Gottschling, P., Seligman, M., Zorn, B.: Universal numbers library: design and implementation of a high-performance reproducible number systems library. arXiv e-prints (2020)
22. Sharma, N., et al.: CLARINET: a RISC-V based framework for posit arithmetic empiricism, pp. 1–18, May 2020
23. Sze, V., Chen, Y.H., Yang, T.J., Emer, J.S.: Efficient processing of deep neural networks: a tutorial and survey. Proc. IEEE **105**(12), 2295–2329 (2017). https://doi.org/10.1109/JPROC.2017.2761740
24. Uguen, Y., Forget, L., de Dinechin, F.: Evaluating the hardware cost of the posit number system. In: 2019 29th International Conference on Field Programmable Logic and Applications (FPL), pp. 106–113. IEEE (2019). https://doi.org/10.1109/FPL.2019.00026