Square Root Unit with Minimum Iterations for Posit Arithmetic

Raul Murillo
Faculty of Physics
Complutense University of Madrid
28040 Madrid, Spain
Email: ramuri01@ucm.es

Alberto A. Del Barrio
Faculty of Computer Science
Complutense University of Madrid
28040 Madrid, Spain
Email: abarriog@ucm.es

Guillermo Botella
Faculty of Computer Science
Complutense University of Madrid
28040 Madrid, Spain
Email: gbotella@ucm.es

***Abstract*—In this paper, we introduce a novel implementation of a square root algorithm specifically tailored for posit arithmetic. Unlike traditional methods, the proposed approach capitalizes on the inherent flexibility of posits, which lack fixed-length fields, to optimize square root computations. By accurately estimating the minimum number of required fraction bits, our algorithm substantially reduces the recurrence iterations without sacrificing accuracy. Implemented across standard 16-bit, 32-bit, and 64-bit posit formats, our units showcase a significant latency reduction in different applications with only a marginal increase in resource utilization. Comparative analysis against previous pipelined designs underscores the area efficiency of our proposed solutions. This research significantly contributes to the advancement of posit-based arithmetic units, presenting promising opportunities for improving computational system efficiency.**

***Index Terms*—Iterative algorithms, Square root, Posit arithmetic**

I. Introduction

Posit arithmetic is a relatively recent alternative to floating-point arithmetic for representing real numbers in computer systems [1]. This format has been shown to provide a trade-off between precision and dynamic range. Posit numbers (posits in short) can represent a wide range of real numbers with varying magnitudes and precision levels, making them well-suited for both large and small values without suffering from the same level of precision loss as floating-point formats. This has been provided to be very interesting for multiple applications, including climate modeling [2], radio astronomy [3], computation of differential equations [4], [5] or deep learning [6], [7]. To offer such a flexible representation of real numbers, in contrast to traditional floating-point formats, posit numbers use a regime-exponent-fraction format where both the regime and the exponent represent the scaling factor, and the fraction contains the significant digits. However, this novel format is not compatible with IEEE 754 formats [8] and derivates [9], [10], and thus dedicated algorithms and hardware units must be redesigned for practical computation in this format.

While multiple works have studied how to implement addition and multiplication in posit format [11], [12], only a few have explored the division and square root operations [13], [14], [15], which are known to be much more complex than the former ones. Efficient hardware support is necessary to leverage properties of posit arithmetic in the aforementioned areas.

Despite the significance of square roots, the efficient computation of such an operation in posit arithmetic remains largely unexplored territory. In this paper, we focus on addressing this gap by proposing a novel computer square root unit tailored specifically for posit arithmetic.

The proposed approach harnesses the inherent properties of posits to provide a high-performance and resource-efficient solution for computing square roots in such a format. In particular, we find the minimum number of iterations that the radix-2 digit-recurrence algorithm needs to provide the exact result, thus reducing the latency associated with square root operations while maintaining accuracy and precision.

Through comprehensive experimentation and evaluation, the proposed minimum-iterations unit consistently exhibits lower latency than existing approaches in posit arithmetic with a negligible area overhead, as well as requiring much fewer resources than previous implementations.

The rest of this paper is organized as follows. Section II reviews the fundamentals of posit arithmetic and previous works related to posit square root units. Section III briefly describes the foundations of digit-recurrence square root and delves into the search of the minimum number of iterations required by the algorithm when applied to posit arithmetic. In Section IV the general architecture is presented and the main features of the proposed units are outlined. An experimental performance analysis and hardware evaluation of the units are discussed in Section V. Finally, Section VI presents the main conclusions of this work.

II. Background

A. Posit arithmetic

Posit arithmetic was introduced in 2017 as an alternative to the ubiquitous IEEE 754 floating-point standard to represent and operate with real numbers [1]. The posit number system is a floating-point encoding scheme with tapered precision, which is achieved thanks to a variable-length encoding of the scaling factor. According to the latest Standard for Posit™ Arithmetic (2022) [16], an n-bit posit number (namely, Posit$n$) is encoded with four fields, as shown in Fig. 1:

Fig. 1. General Posit$n$ binary encoding.

- **Sign.** As for floats or signed integers, the first bit stands for the sign (s): 0 for positive numbers, 1 for negative numbers. In contrast to floats, which use a sign-magnitude representation, posits are encoded in two's complement.
- **Regime.** This field is unique to this number format. The regime consists of a sequence of r_s identical bits R terminated either with the negation of such value ($1 - R = \overline{R}$) or with the least significant bit (LSB) of the posit. This sequence encodes the scaling factor r, given by conditional Eq. (1). For example, when the regime is 4-bits, pattern `1110` encodes $r = 2$, while `0001` stands for $r = -3$.

$$r = \begin{cases} -r_s & \text{if } R = 0 \\ r_s - 1 & \text{if } R = 1 \end{cases}. \tag{1}$$

- **Exponent.** The following two bits preceding the regime encode another scaling factor e. Unlike with floats, the exponent is unbiased. As the length of the regime field is variable, there exists the possibility that some exponent bits (or even all of them) are shifted out of the bit-string. In such cases, missing bits are considered as `0`.
- **Fraction.** The remaining bits after the exponent correspond to the fraction field. Its value $f \in [0, 1)$ is obtained by dividing the unsigned integer encoded in this field by 2^{f_s}. This is similar to the case of floats, with the only difference being that the hidden bit depends on the sign rather than on the exponent, so denormalized posits exist.

Posit numbers only distinguish two special cases: zero and Not a Real (NaR), which are represented as `00...0` and `10...0` respectively. The rest of the representations are composed of the aforementioned four fields, and the real value p encoded is given by Eq. (2),

$$p = (-1)^s \times (1 + f) \times 2^{4r+e}. \tag{2}$$

B. Related work

Since its initial introduction, there has been considerable interest in hardware implementations of posit arithmetic. A plethora of standalone posit arithmetic units with different degrees of capabilities have been described in the literature, mostly focused on addition, multiplication, and fused operations [17], [18], [19], [12].

However, just a few works have addressed posit square root computation [13], [14]. Both works propose functional units for division and square root, since the algorithms for the two operations are pretty similar. Indeed, the unit presented in [13] supports the two operations in the same pipelined hardware design. While the units from [14] are based on the Newton-Raphson algorithm, the fused division and square-root architecture from [13] is iterative, as it implements the non-restoring algorithm.

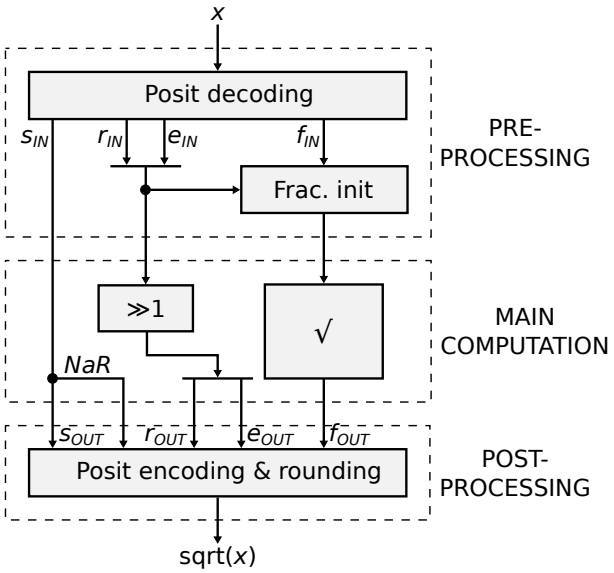

Fig. 2. Basic implementation of posit square root.

Other works related to the implementation of a whole posit processing unit include a square root unit component [20], [21], [22]. While a 32-bit logarithm-approximate unit is employed in [21], the other works implement a pipelined non-restoring square root unit for different posit configurations. Non-standard formats of 8-, 16-, and 32-bit posits with 1-, 2-, and 3-bit exponents, respectively, are considered in [20]. On the other hand, [22] implements arithmetic units for the standard Posit32 and Posit64 formats with 2 exponent bits.

III. Posit square root

The square root is the only one among the basic operations that considers a single input operand. As shown by Eq. (3), the square root of a non-negative posit number p requires obtaining the square root of the significand and producing the scaling factor of the result.

$$\sqrt{p} = \sqrt{1 + f} \times 2^{\frac{(4r+e)}{2}}. \tag{3}$$

When implementing in hardware, if the exponent is odd, it is decremented by 1, and the significand is multiplied by 2.

There exist several classes of algorithms and implementations for the posit square root operation, but all of them follow the same basic implementation depicted in Fig. 2. The main difference resides in the computation of the square root of the significand, for which digit recurrence, multiplicative, approximate, or other special methods such as CORDIC are often used. The methods of performing the square root operation are conceptually very similar to the methods for division [23], [24].

A. Digit-recurrence square root

Digit-recurrence is a class of iterative algorithms that compute one digit of the result (represented in a radix-r form)

Algorithm 1 Non-restoring square root.

procedure NON-RESTORING(X)
  $R_0 \leftarrow X$
  $Q_0 \leftarrow 0$ ▷ $Q = (q_0.q_1q_2 \dots q_m)$
  for $i = 0, \dots, m-1$ **do**
    if $2R_i \geq 0$ have the same sign **then**
      $Q_{i+1}[i+1] \leftarrow 1$
      $R_{i+1} \leftarrow 2R_i - (2Q_i + 2^{-i})$
    else
      $Q_{i+1}[i+1] \leftarrow -1$
      $R_{i+1} \leftarrow 2R_i + (2Q_i - 2^{-i})$
  return Q_m

and the remainder every iteration [23], [24]. The remainder is used to obtain the next least significant digit of the result. Digit-recurrence algorithms are used frequently in modern processors for the calculation of division and square root, as they present a good trade-off in terms of performance, area, and power [25].

In this work, the square root is implemented using the binary (radix-2) non-restoring method. Given a radicand X, the details of such a method are described in Algorithm 1, being R_i and Q_i the remainder and partial result, respectively. At every iteration i, a new digit of the result is obtained from the current remainder, a new remainder is computed for the next iteration, and the partial result is updated.

As using radix-2, the partial result digits can take values $q_i \in \{-1, 0, 1\}$, which must be converted to a binary representation every iteration. The most efficient conversion technique is the well-known *on-the-fly conversion* [23], [24].

To obtain the root, the algorithm requires the operand to be in the range $[0.25, 1)$. Since the significands of posit numbers are within $[1, 2)$, the input to the square root submodule is divided by 2 (of by 4, if the exponent is odd) to guarantee such a condition. This is done by a 1-bit (or 2-bit) operand left-shift.

B. Number of sqrt iterations

The digit-recurrence methods obtain one digit per iteration, from most to least significant, i.e., the result generated by these methods becomes more accurate after each iteration. Therefore, the number of iterations of the recurrence step is dependent on the number of bits of the final result, but also on the scaling introduced by the initialization, the rounding bits, and the radix [23]. In the particular case of radix-2, the number of iterations is $m = k + 2$, where k is the bit-length of the radicand.

In the realm of posit arithmetic, the regime, and thus the fraction field, do not always have the same bit-length. For example, consider the following two Posit8 numbers

$$p_1 := \; 0\,000\,1\,01\,1_2 = 0.000\,732\,421\,875_{10},$$
$$p_2 := \; 0\,11\,0\,10\,11_2 = 112_{10}.$$

By calculating the square root of the above numbers (note that posit arithmetic applies round to nearest even) we get

$$\sqrt{p_1} := \; 0\,00\,1\,10\,11_2 = 0.027\,343\,75_{10},$$
$$\sqrt{p_2} := \; 0\,1\,0\,11\,011_2 = 11_{10},$$

which have more fraction bits than p_1 and p_2, but not necessarily the maximum three fraction bits that a Posit8 number can have. The square root operation always returns a closer value to 1. In posit arithmetic, numbers have more fraction bits as they approach 1. This means that the square root of a posit number will have the same or more fraction bits than the radicand.

Computing more fraction bits than the length of the final fraction is thus unnecessary, as such bits will be discarded (just a pair of extra bits are necessary for proper rounding). On this basis, let us determine, for a given posit number, the exact number of fraction bits that the result of the square root of that number will have. In this way, we will find the minimum number of iterations that the algorithm needs to provide the exact result, thus reducing the latency of the operation.

Let us denote the size (amount of bits) of the regime and fraction fields in a posit number as r_s and f_s, respectively. For a more general proof, let us consider any fixed size for the exponent field to e_s (if the regime is large enough, such exponent bits are not shown in the bitstring, and thus they are considered to have the value 0). Considering that the sign bit is always present, and the regime is delimited by a flipped bit, the length of the fraction length for any Posit$n$ number is given by Eq. (4),

$$f_s = \max\{n - 2 - r_s - e_s, 0\}. \tag{4}$$

Note that, since the minimum regime length is 1, the maximum bit-length of the fraction field is $n - e_s - 3$.

When performing the square root of a posit number, according to Eq. (3), the value comprised by the regime and the exponent is divided by 2. However, the exponent and regime fields of the result must be encoded in the same manner as every posit data, i.e., using 2 bits for the exponent, and considering the regime as a factor shifted according to the exponent size. Therefore, considering $\widehat{p}$ to be the root of a non-negative posit number p, we can re-write Eq. (3) as Eq. (5),

$$\begin{aligned} \widehat{p} = \sqrt{p} &= \sqrt{1+f} \times 2^{\frac{(2^{e_s} r + e)}{2}} \\ &= \sqrt{1+f} \times 2^{2^{e_s} \frac{r}{2} + \frac{e}{2}} \\ &= (1 + \widehat{f}) \times 2^{(2^{e_s} \widehat{r} + \widehat{e})}. \end{aligned} \tag{5}$$

As can be seen, when applying the square root operation to a posit number, the regime value of the result will be half (note that, when implemented in hardware, if the initial regime is odd, the least significant bit will be absorbed by the most significant bit of the exponent, and the case of odd exponent is treated in the algorithm initialization). At this point, it is worth recalling two things: (1) the exponent always encodes a non-negative value, and (2) if the regime value is negative, the regime field contains as bits as such a value, otherwise the length of the regime is one more bit than the value. Under

TABLE I
NUMBER OF RESULT FRACTION BITS, ITERATIONS OF THE ALGORITHM AND LATENCY OF THE SQRT UNIT

	Fraction bits	Radix-2 iterations	Sqrt unit latency
Posit16	4 – 11	6 – 13	8 – 15
Posit32	12 – 27	14 – 29	16 – 31
Posit64	28 – 59	30 – 61	32 – 63
Float16	10	12	14
Float32	23	25	27
Float64	52	54	56

these considerations it follows that the size of the root regime will also be half the initial size, rounding up in the case of odd length. Substituting this in Eq. (4) yields the size of the resulting fraction field, depicted in Eq. (6),

$$\widehat{f_s} = n - 2 - \left\lceil \frac{r_s}{2} \right\rceil - e_s. \tag{6}$$

Note that, since $r_s \in [1, n-1]$, the result given by Eq. (6) is also bounded. In particular, $f_s \in \left[\left\lceil \frac{n}{2} \right\rceil - 2 - e_s, n - 3 - e_s\right]$. Finally, as aforementioned, the algorithms require two more iterations than the fraction size of the result.

The number of resulting fraction bits for common posit configurations, as well as the number of radix-2 iterations of the proposed approach, are summarized in the left and center columns of Table I. Indeed, it shows the minimum number of iterations that must be computed with this algorithm to provide an exact posit answer. Note that, if the size of the final fraction is not considered, the number of iterations of the algorithm should always be fixed according to the largest possible fraction size, i.e. $n - 1 - e_s$. As can be seen, this approach can reduce the number of iterations in this kind of algorithm, especially in the larger bit-length cases. For comparison purposes, fraction size and iterations for IEEE 754 floating-point formats are also displayed in Table I.

IV. UNIT IMPLEMENTATION

The square root algorithm presented in Section III can be implemented in multiple ways. In this paper, a sequential design is presented to reuse the digit iteration logic over several cycles, resulting in a lower area design. This decision has the drawback that can not be pipelined, resulting in a lower throughput. However, pipelining these kinds of iterative architectures substantially increases hardware resources. In modern processors, the throughput is increased by placing several non-pipelined units operating in parallel. Moreover, in this work, we leverage the variable length feature of the posit format to provide higher throughput in the square root operation, computing just the minimum necessary iterations of the digit-recurrence algorithm.

The general organization of the proposed posit sqrt unit is shown in Fig. 2. The three parts of the microarchitecture are clearly differentiated.

The pre-processing includes decoding the posit regime, exponent, and fraction fields, as well as the fraction preparation for digit recurrence square root. The number of digit recurrence steps is also computed in the pre-processing, based on the regime length of the input posit. The main phase of the computation consists of calculating the square root of the fraction. For this purpose, a combinational module that implements a recurrence step has been designed, together with a sequential control unit that manages the data flow. The control unit can handle a variable number of computation iterations so that it signals as soon as the final result is available. The final regime and exponent are also computed in this phase. Post-processing includes the rounding and encoding of the regime to have a posit-compliant result. The three parts of the unit are split into different hardware stages.

As shown in the right-side of Table I, the latency of the units is the sum of the digit iterations cycles, plus the number of pre-processing and post-processing cycles (special cases can be detected in the pre-processing stage, but such latency is omitted for the sake of clarity).

A case that deserves special attention is when a zero remainder is obtained in an intermediate step. If the remainder gets such a value in a certain iteration, it means that the exact root of the input is computed at that time, and no extra iterations are needed. Therefore, detecting the occurrence of a zero intermediate remainder allows for early-termination, reducing the latency in certain situations. This optimization is incorporated into the proposed square root unit.

The aforementioned design was implemented in VHDL for different configurations: Posit16, Posit32, and Posit64. The units were verified with the help of the reference library Universal [26], generating an exhaustive testbench for the 16-bit unit, as well as random tests for the 32-bit and 64-bit units. All these tests were successful. The implementation of these posit units has been made freely available[1].

V. EVALUATION

In this section, the proposed sequential square root unit is evaluated in terms of both performance and hardware resources, and compared with the naive square root approach that performs a fixed number of iterations.

Unfortunately, none of the previous works [13], [14] presenting posit square root units have made the implementations publicly available. Just the works presented in [20], [22] integrate in their designs a posit square root unit available for comparison. Although such units have implementations of the non-restoring algorithm, only the ones from [22] maintain standard posit formats with a 2-bit exponent. Consequently, only the latter designs have been included in this discussion.

A. Latency reduction

The latency of square root calculation with the proposed approach is shown on the right side of Table I. The table shows the minimum and maximum latency for the three posit precisions of interest, Posit16, Posit32, and Posit64, and for the theoretical latency corresponding IEEE 754 floating-point

[1]https://github.com/artecs-group/ARITH24

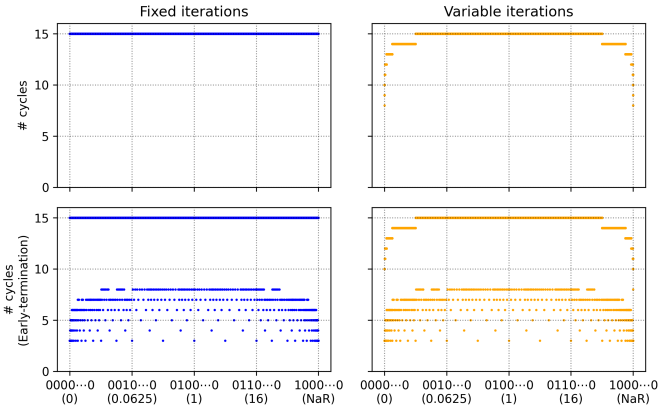

Fig. 3. Latency of the non-restoring Posit16 square root unit per input.

precisions. Note that, in a scenario that does not take into account the actual size of the resulting fraction, the maximum number of iterations should be performed, as in the case of floating point units.

As can be seen in the table, the proposed approach makes it possible to reduce the latency of the square root operation by half, so that it can be even lower than for floating-point.

However, this depends only on the length of the regime and the fraction of the radicand, and therefore on the dynamic range and precision of the particular input. It is noteworthy that such a reduction in the number of iterations does not reduce the precision of the result, but merely produces a result with just the right precision, without any more fraction bits that will be discarded in the rounding and post-processing stage.

To provide a better understanding of the relation between the input radicand and the latency of the unit, we simulated an exhaustive testbench for Posit16 inputs and recorded the number of cycles used by the unit to generate the correct result (with the use of a synchronous counter). Fig. 3 displays, for every non-negative Posit16 number, the latency of the proposed square root unit when considering either a fixed or a variable (minimum) number of iterations.

As can be seen, for the central region (which corresponds to values close to 1), the variable approach can not reduce any iteration. This is because, in such a region, the size of the posit regime is minimal, while more bits are used to represent the fraction, and values within that region have a higher precision. In fact, this is the region where posits provide higher accuracy than floats [27]. However, as we move away from this zone, the number of iterations required by the algorithm becomes smaller, since the regime bits increase (allowing us to represent quantities of much smaller/larger magnitude).

Also, Fig. 3 compares the actual latency of the units with or without using the early-termination for zero intermediate remainder. As can be seen, such an optimization allows to compute the result faster in some cases, even with fewer cycles than those indicated by Table I (3 instead of 8 in the best case, when the input is a power of 4, and the remainder becomes zero after the first iteration of the algorithm). However, the plots regarding early-termination still exhibit the same latency upper bound. Also, about half of the zero intermediate remainders are detected in the region that requires the maximum possible iterations. All this suggests that such an optimization could not be equivalent or alternative, but complementary to the minimum iteration estimation presented in this paper.

On the other hand, it would be worthwhile to evaluate the performance of the proposed units in real applications to compare both approaches. To accomplish this, we considered different benchmarks from the PolyBench/C benchmark suite [28], in particular, those containing square root operations, and simulated Posit32 computation using the software library Universal [26]. The input and output of the square root operations were collected for processing in the implemented hardware units. Table II contrasts the performance of the proposed square root units employing the fixed- and variable-iterations methods.

Utilizing a variable-iterations algorithm effectively reduces latency across all benchmark scenarios, to a greater extent the larger the size of the dataset. While the reduction of cycles is not very pronounced for the *correlation* benchmark (reducing the cycles by 0.02% in the MINI dataset, up to 3.23% in the LARGE dataset), the effect is more notable in the *gramschmidt* benchmark (reducing the number of cycles between 7.58% and 8.92%). The reason behind this lies in the dynamic range of the data. While the posit numbers computed in the *gramschmidt* benchmark lie in an interval of, for example, $\left(3.6 \times 10^{-12}, 1.7 \times 10^{5}\right)$ in the MEDIUM dataset, the correlation benchmark data fall within the interval $\left(16, 2.3 \times 10^{4}\right)$, i.e., much more concentrated in the area of smaller posit regime, the central area shown in Fig. 3.

In summary, it has been shown that the proposed posit square root unit can reduce several iterations, providing better throughput for this operation, but this depends on the dynamic range of the data.

One aspect worth noting is the effect of the early-termination by a zero remainder. To answer this question, these same experiments were repeated without using the detection of the occurrence of a zero intermediate remainder for an early-termination. It turned out that just the LARGE *gramschmidt* benchmark was taking advantage of such optimization. In particular, a total of 17 cycles were saved thanks to this early-termination, in the baseline operator, and 15 cycles in the variable-latency proposed unit. Nevertheless, other applications may stand to benefit from such an optimization, as numerous posits have a perfect square as their significand value, as shown in Fig. 3.

In light of these results, it is substantiated that in real applications, the detection of zero intermediate remainder has almost no overlap with the estimation of the number of iterations, and therefore both optimizations can be leveraged to further increase the performance of the unit. Furthermore, the cost associated with implementing the detection of whether the remainder is zero is minimal.

TABLE II
NUMBER OF CYCLES CORRESPONDING TO SQUARE ROOT OPERATIONS IN POSIT32 FOR DIFFERENT BENCHMARKS

	correlation				gramschmidt			
Design	MINI	SMALL	MEDIUM	LARGE	MINI	SMALL	MEDIUM	LARGE
# operations	924	8080	62640	1681200	30	80	240	1200
Fixed iterations	28644	42480	1941840	52117200	930	2480	7440	37183
Variable iterations	28637	42400	1879200	50434800	847	2292	6803	34291
Reduction	-7	-80	-62640	-1682400	-83	-188	-637	-2892

TABLE III
SYNTHESIS RESULTS

Bits	Design	Area (μm^2)	Power (mW)	Slack (ns)
16	Fixed iterations	625.21	0.0810	0.46
	Variable iterations	638.19	0.0836	0.46
32	Fixed iterations	1296.79	0.1501	0.06
	Variable iterations	1314.43	0.1534	0.06
	PERCIVAL [22]	5937.37	0.5406	0.00
64	Fixed iterations	3256.47	0.3047	0.19
	Variable iterations	3282.80	0.3089	0.19
	PERCIVAL [22]	26665.25	2.0427	0.00

B. ASIC synthesis

The proposed square root unit designs are evaluated in terms of area, power, latency, and energy. Both proposed designs, using a fixed or a variable number of iterations, implement the zero remainder detection for an early-termination. The variable-latency units dynamically compute the minimum required recurrence iterations according to the regime of the input.

The comparison between the variable- and the fixed-iterations units is shown in Table III. Units for 16-bit, 32-bit, and 64-bit posit configurations have been implemented using a TSMC 28 nm technology and targeting the same frequency of 1 GHz.

Starting with the delay, the timing constraint was met by all the units. In fact, the critical path of the units is in the square root module, and not in the pre-processing stage, which contains the logic for performing the estimation of the number of iterations.

As expected, the variable-latency designs present a certain area overhead, as it requires computing the number of digit-recurrence iterations, in contrast to just considering a fixed amount of iterations. However, such units use a very small amount more area, ranging from 0.81% for the Posit64 unit, up to 2.08% in the case of Posit16.

In terms of power, again a small overhead is obtained as in the case of the area, as they are correlated. More precisely, the novel variable-latency 16-bit, 32-bit, and 64-bit posit square root units use 3.21%, 2.20%, and 1.38% more power than the classical fixed-latency, respectively.

Overall, the numbers displayed in Table III show a relatively small overhead for the variable-iterations approach, especially considering the substantial potential for reducing latency in square root operations offered by such units.

Finally, regarding previous works, the posit arithmetic unit presented in [22] contains a 5-stage pipelined square root unit for Posit32, and a 13-stage pipeline unit for Posit64. Although this might provide much better throughput than the proposed units, according to the original paper, the arithmetic units are designed to work at 50 MHz on FPGA, a much lower frequency than the one set as target in this evaluation (1 GHz). To make the comparison as fair as possible, the exact square root units from PERCIVAL [22][2] were implemented targeting the same technology and frequency. The synthesis results in Table III reveal the cost for such a high performance. The area and power of the 32-bit pipelined units are 4.52× and 3.52× more than the proposed variably-latency design, respectively. This overhead is even more pronounced in the 64-bit case, increasing area and power by a factor of 8.12× and 6.61×, respectively.

VI. CONCLUSION

A modified radix-2 non-restoring algorithm is presented as a core unit for posit square root arithmetic circuits. The proposed approach estimates the minimum number of recurrence iterations that provide the exact result without compromising accuracy.

The proposed algorithm has been implemented for standard posit configurations of 16-, 32-, and 64-bits. The units reduce latency by up to half for certain posit values compared to the classical fixed-iterations approach used in other arithmetic formats. Additionally, some other improvements have been made in the square root units to increase throughput.

Evaluation reveals an effective latency reduction of up to 8.92% in real applications with marginal area overhead. Hardware utilization was also compared against previous pipelined designs, illustrating the efficiency of the proposed units.

This work opens avenues for advancing posit-based arithmetic units, enhancing efficiency in computational systems. Future work involves integrating the proposed minimum-iterations approach into posit division units, as the methods of performing such an operation are conceptually similar to those presented in this paper.

[2]Designs obtained from https://github.com/artecs-group/PERCIVAL/tree/fd1d154.

REFERENCES

[1] J. L. Gustafson and I. T. Yonemoto, "Beating Floating Point at Its Own Game: Posit Arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, 2017.

[2] M. Klöwer, P. D. Düben, and T. N. Palmer, "Number Formats, Error Mitigation, and Scope for 16-Bit Arithmetics in Weather and Climate Modeling Analyzed With a Shallow Water Model," *Journal of Advances in Modeling Earth Systems*, vol. 12, no. 10, 2020.

[3] T. K. Gunaratne, "Evaluation of the Use of Low Precision Floating-Point Arithmetic for Applications in Radio Astronomy," in *Conference for Next Generation Arithmetic (CoNGA)*, vol. 13851 LNCS. Springer Nature Switzerland, 2023, pp. 155–170.

[4] R. Murillo, A. A. Del Barrio, and G. Botella, "The Effects of Numerical Precision In Scientific Applications," in *2022 Annual Modeling and Simulation Conference (ANNSIM)*. IEEE, 2022, pp. 152–163.

[5] M. Klöwer, P. V. Coveney, E. A. Paxton, and T. N. Palmer, "Periodic orbits in chaotic systems simulated at low precision," *Scientific Reports*, vol. 13, no. 1, p. 11410, 2023.

[6] R. Murillo, A. A. Del Barrio, and G. Botella, "Deep PeNSieve: A deep learning framework based on the posit number system," *Digital Signal Processing: A Review Journal*, vol. 102, p. 102762, 2020.

[7] J. Lu *et al.*, "Evaluations on Deep Neural Networks Training Using Posit Number System," *IEEE Transactions on Computers*, vol. 70, pp. 174–187, 2021.

[8] IEEE Computer Society, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, vol. 2019, pp. 1–84, 2019.

[9] J. Hormigo and J. Villalba, "New formats for computing with real-numbers under round-to-nearest," *IEEE Transactions on Computers*, vol. 65, no. 7, pp. 2158–2168, 2016.

[10] G. Henry, P. T. P. Tang, and A. Heinecke, "Leveraging the bfloat16 Artificial Intelligence Datatype for Higher-Precision Computations," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 69–76.

[11] R. Murillo, A. A. Del Barrio, and G. Botella, "Customized Posit Adders and Multipliers using the FloPoCo Core Generator," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[12] H. Zhang and S.-B. Ko, "Design of Power Efficient Posit Multiplier," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 861–865, 2020.

[13] A. Raveendran *et al.*, "A Novel Parametrized Fused Division and Square-Root POSIT Arithmetic Architecture," in *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*, 2020, pp. 207–212.

[14] F. Xiao *et al.*, "Posit Arithmetic Hardware Implementations with The Minimum Cost Divider and SquareRoot," *Electronics*, vol. 9, no. 10, 2020.

[15] R. Murillo, A. A. Del Barrio, and G. Botella, "A Suite of Division Algorithms for Posit Arithmetic," in *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2023, pp. 41–44.

[16] J. L. Gustafson *et al.*, "Standard for Posit™ Arithmetic," Posit Working Group, Standard, 2022, online. [Online]. Available: https://github.com/posit-standard/Posit-Standard-Community-Feedback

[17] M. K. Jaiswal and H. K. So, "PACoGen: A Hardware Posit Arithmetic Core Generator," *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019.

[18] R. Murillo, D. Mallasén, A. A. Del Barrio, and G. Botella, "Energy-Efficient MAC Units for Fused Posit Arithmetic," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021, pp. 138–145.

[19] H. Zhang, J. He, and S.-B. Ko, "Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 2019-May. IEEE, 2019.

[20] S. D. Ciocirlan *et al.*, "The Accuracy and Efficiency of Posit Arithmetic," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 2021, pp. 83–87.

[21] D. Mallasén *et al.*, "PERCIVAL: Open-Source Posit RISC-V Core With Quire Capability," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 3, pp. 1241–1252, 2022.

[22] D. Mallasén, A. A. Del Barrio, and M. Prieto-Matias, "Big-PERCIVAL: Exploring the Native Use of 64-Bit Posit Arithmetic in Scientific Computing," *IEEE Transactions on Computers*, 2024.

[23] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Elsevier, 2004.

[24] I. Koren, *Computer Arithmetic Algorithms*. A K Peters/CRC Press, 2018.

[25] J. D. Bruguera, "Low latency floating-point division and square root unit," *IEEE Transactions on Computers*, vol. 69, no. 2, pp. 274–287, 2020.

[26] E. T. L. Omtzigt and J. Quinlan, "Universal: Reliable, Reproducible, and Energy-Efficient Numerics," in *Conference on Next Generation Arithmetic (CoNGA)*, vol. 13253, 2022, pp. 100–116.

[27] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the Hardware Cost of the Posit Number System," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. Barcelona, Spain: IEEE, 2019, pp. 106–113.

[28] L.-N. Pouchet and T. Yuki, "PolyBench/C 4.2," https://sourceforge.net/projects/polybench/, 2016.