A Suite of Division Algorithms for Posit Arithmetic

Raul Murillo*, Alberto A. Del Barrio† and Guillermo Botella†
*Dept. of Computer Architecture and Automation, *Faculty of Physics, †Faculty of Computer Science*
Complutense University of Madrid, 28040 Madrid, Spain
{ramuri01, abarriog, gbotella}@ucm.es

***Abstract*—Posit™ arithmetic is a promising alternative to IEEE 754 floating-point arithmetic due to its higher accuracy, larger dynamic range, and bitwise compatibility. While posit arithmetic has been well studied for basic arithmetic operations, division has received little attention. This paper proposes multiple divider designs for posit arithmetic based on digit recurrence and iterative approximation, and evaluates their performance. ASIC synthesis results show that the proposed designs significantly reduce hardware requirements for 32-bit division units compared to previous works by 1.14× in area, 1.11× in power, and 1.04× in datapath delay. Moreover, the paper introduces an approximate logarithmic posit division that achieves an 8.8× reduction in area and 29× reduction in energy consumption with negligible degradation of the final results, making it suitable for error-tolerant applications.**

***Index Terms*—Posit arithmetic, Computer arithmetic, Division, Approximate computing**

I. Introduction

The IEEE 754 standard has been widely used for scientific and real number-based applications, but issues such as rounding and NaN representations have been encountered. Several alternatives to IEEE 754-2008 arithmetic have been proposed, and posit™ arithmetic is one of the most promising [1]. Posit format has been investigated for various areas, including deep learning and simulations [2], [3]. However, few papers have presented posit division algorithms and implementations, despite the need for such operations in machine learning and scientific applications. Efficient hardware support is necessary to leverage properties of posit arithmetic in these areas.

This paper proposes multiple posit divider designs, including digit recurrence and iterative approximation schemes. Classical division algorithms, such as non-restoring, Newton-Raphson, and Goldschmidt, are adapted to posit arithmetic. In addition, an approximate division method based on logarithmic properties is also proposed, which is the first of its kind. Synthesis results show that the proposed 32-bit dividers reduce hardware resources in terms of area, delay, power, and energy by a factor of 1.14×, 1.04×, 1.12×, and 1.16×, respectively, and that approximate posit division can reduce energy consumption by a factor of 29× and area by a factor of 8× without affecting the quality of results in error-tolerant applications.

This work was supported by grant PID2021-123041OB-I00 funded by MCIN/AEI/ 10.13039/501100011033 and by "ERDF A way of making Europe", and by the CM under grant S2018/TCS-4423.

II. Background

A. Posit Arithmetic

In 2017, posit arithmetic [1] emerged as an alternative to IEEE 754 floating-point standard. Posit numbers (posits) offer some advantages over floating-point numbers: they guarantee reproducible results across platforms, provide higher accuracy for values near ± 1 for the same bitwidth, and do not consider overflow or underflow, simplifying exception handling.

Posits comprise four fields, as shown in Fig. 1: a sign bit (s), a regime (r), a two-bit exponent (e), and the remaining bits for the normalized fraction (f). The regime is a sequence of k identical bits ending with a negated bit or the end of the posit; it encodes an extra scaling factor of $-k$ if regime bits are 0, otherwise $k-1$. Only two special cases, zero and Not a Real (NaR), are considered (when all bits apart from the sign are 0). The other bit patterns represent different real values P as shown in (1):

$$P = (1 - 3s + f) \times 2^{(1-2s)\times(4r+e+s)}. \tag{1}$$

Posit arithmetic has demonstrated promising accuracy improvements thanks to its tapered precision and the implementation of fused operations such as fused multiply-add, in which intermediate computations are not rounded, but accumulated in a larger register named *quire* [4].

B. Related Work

Existing research on posit arithmetic focuses mostly on addition, multiplication, and fused operations [5]–[7], with fewer works addressing posit division. Two implementations of posit division based on the Newton-Raphson method have been proposed [8], [9]. They use a sign-magnitude decoding scheme, followed by a custom floating-point format conversion to perform the division algorithm with different exponent and mantissa sizes. However, previous research suggests that using a two's complement decoding scheme is more efficient for posit arithmetic [10]. In this work, we explore a two's complement encoding scheme for the Newton-Raphson algorithm. Another implementation of posit division based on the non-restoring algorithm was proposed in [11], which also used two's complement notation, but with higher latency than the Newton-Raphson method. A fused division and square-root architecture based on the non-restoring algorithm was presented in [12].

DOI 10.1109/ASAP57973.2023.00020

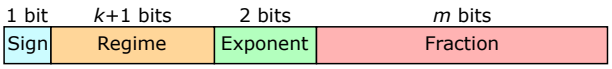

Fig. 1. Generic n-bit posit binary encoding.

III. Posit Division Algorithms

Division is the most complex of the four basic arithmetic operations. Unlike the other basic arithmetic operations, it often yields a quotient and remainder. In floating-point and posit arithmetic, the remainder is often disregarded, and division produces an approximate result due to potential non-multiples of the divisor. Fast division algorithms are commonly used to reduce complexity.

Posit division involves a fixed-point division with additional hardware for exponent handling, leading to more complex divider circuits. The division of A by B in posit arithmetic can be represented by (2):

$$\frac{A}{B} = \frac{(1-3s_A+f_A)2^{M_A}}{(1-3s_B+f_B)2^{M_B}} = \frac{(1-3s_A+f_A)}{(1-3s_B+f_B)} \times 2^{M_A-M_B}, \tag{2}$$

where $M = (1-2s)\times(4r+e+s)$ refers to the scaling factor of the number. As can be seen, in posit division fractions are divided and exponents are subtracted.

The major steps for a posit division are:

- Extract the sign (s) of the result from the two sign bits.
- Find the magnitude of the difference between exponents.
- Divide the significands of A and B, i.e., the fractions f_A and f_B with hidden bits .
- Normalize the result, if necessary, and decremented the exponent accordingly.

The general architecture for posit division is shown in Fig. 2. If the result of the divider is not in the correct range of values, then the normalizing step is executed. Since the significand of any posit number is in the interval $[-2,-1)\cup[1,2)$, the result from the division of two significands will be in $[-2,-0.5]\cup(0.5,2)$. Thus, up to two bits (when the quotient is -0.5) should be shifted from the result, and subtracted to the final exponent.

Binary division requires a significant number of hardware resources and multiple clock cycles. It is often not included in FPGA or ASIC libraries. Different algorithms can be used to implement a binary divider. However, unlike in the case of floating-point division, the numerator, denominator, and quotient are all signed values, which requires modifications to the algorithm. The selected division algorithm will impact the accuracy, area, and delay of the final functional unit.

Various classes of division algorithms exist, including digit recurrence algorithms (non-restoring), iterative methods, and logarithm-based approximations. Here, we present a digit recurrence algorithm and two iterative methods, discussing their necessary modifications for posit arithmetic. We also introduce an approximate division algorithm based on logarithms. Hereafter, the numerator is denoted as $N = f_A$, the denominator as $D = f_B$, the quotient as Q, and the remainder as R, such that $N = D \times Q + R$.

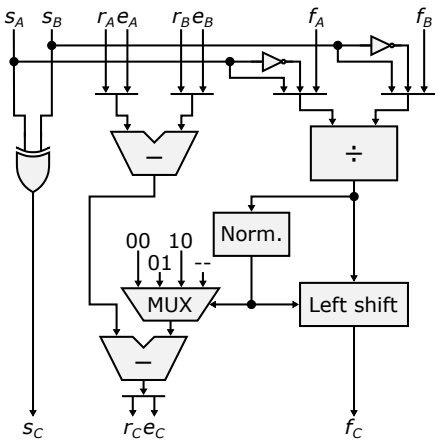

Fig. 2. Basic scheme for posit division.

Algorithm 1 Non-restoring division

```
procedure NonRestoring(N, D)
    R_0 ← N
    for i = 1, ..., k do
        if 2R_{i-1}, D have the same sign then
            q_i ← 1
            R_i ← 2R_{i-1} − D
        else
            q_i ← −1
            R_i ← 2R_{i-1} + D
    Q ← 00.q_1 q_2 ... q_k
    return Q
```

A. Non-restoring Division

This method generates a single quotient bit per iteration using the recurrence of the partial remainder R_i. The number of steps in this method is proportional to the number of bits in the numerator. Although it is slower than other division methods, it produces an exact answer. Algorithm 1 assumes $|N| < |D|$ and uses the digit set $\{-1, 1\}$ for quotient digits instead of $\{0, 1\}$, but it can be easily converted to conventional two's complement representation [13]. To ensure $|N| < |D|$, the numerator N is shifted one position right, and the result Q is shifted one bit left, so one extra iteration is computed. Three more iterations are needed to get the round and guard bits for result normalization. Finally, the quotient Q (to which are added two 0's corresponding to the integer bits of the significand) is corrected to ensure that the final remainder R_k and the numerator N have the same sign [13].

B. Newton–Raphson Division

The Newton-Raphson division method iteratively approximates the reciprocal of the denominator D by (3), until the desired accuracy is achieved.

$$X_i = X_{i-1} \times (2 - D \times X_{i-1}). \tag{3}$$

Algorithm 2 Newton–Raphson division

```
procedure NewtonRaphson(N, D)
    X_0 ← 1/D                          ▷ Initial estimation of 1/D
    for i = 1, ..., k do
        X_i ← X_{i−1} × (2 − D × X_{i−1})
    Q ← N × X_k
    return Q
```

Algorithm 3 Goldschmidt division

```
procedure Goldschmidt(N, D)
    X_0 ← 1/D                          ▷ Initial estimation of 1/D
    for i = 0, ..., k do
        N_i ← N_{i−1} × X_i            ▷ N_{−1} ← N
        D_i ← D_{i−1} × X_i            ▷ D_{−1} ← D
        X_{i+1} ← 2 − D_i
    Q ← N_k
    return Q
```

The method is also known as division by reciprocation, as it calculates the reciprocal of D and multiplies it by the dividend to obtain the quotient. The number of iterations required is proportional to $\log_2(n)$, but the initial estimation directly affects the accuracy of the result [13]. Algorithm 2 illustrates this process for k steps. The initial guess can be obtained from a lookup table (LUT) or an algorithm. More implementation details can be found in Section IV.

C. Goldschmidt Division

The Goldschmidt Division, also known as division by convergence, is an iterative process of repeatedly multiplying the numerator and denominator by a common factor X_i for $i = 0, 1, \ldots, k$, until the denominator converges to 1. The resulting numerator converges towards the desired quotient as shown in (4):

$$Q = \frac{N}{D} = \frac{N}{D} \times \frac{X_0}{X_0} \times \frac{X_1}{X_1} \times \ldots \times \frac{X_k}{X_k} \longrightarrow \frac{Q}{1}. \quad (4)$$

The iterative process is shown in Algorithm 3. This method requires an initial factor X_0 and each iteration involves a single subtraction from 2 and two independent multiplications(which increases the performance of the divider).

D. Approximate Logarithmic Division

On modern processors, the division of real numbers is orders of magnitude slower than multiplication, and it requires multiple clock cycles for iterative convergence. For that reason, it is common to compute a fast approximation of the division in applications with certain error tolerance. A common approach for this is to leverage the properties of logarithms, which allow the division to be performed by a simple subtraction [14].

In this paper, we modify the logarithm-approximate multiplication presented in [15] to perform posit division. To that end, it is necessary to compute the absolute value of negative posits (by taking the two's complement of the posit). This is due to the fact that logarithms of negative values are not well defined.

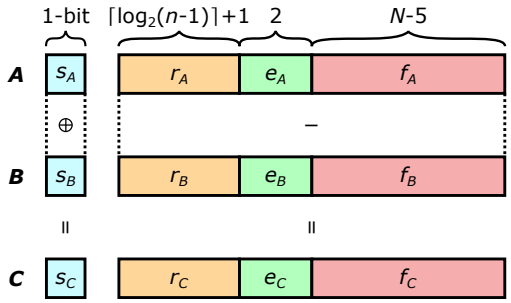

Fig. 3. Scheme for implementing posit logarithmic division in hardware.

However, in this way, it is possible to get an approximate quotient of two posits by concatenating the regime, exponent, and fraction of each operand and subtracting them, as shown in Fig. 3. Finally, as demonstrated in [14], the relative error of such an approximation is no greater than 12.5%.

IV. Hardware Evaluation

A. Implementation of Posit Dividers

The division algorithms discussed in Section III are implemented in VHDL in a combinational manner, with one or two iterations for 16 and 32-bit formats, respectively. The initial reciprocal guess is obtained from a LUT, resulting in a fairer comparison with the PACoGen implementation. To ensure accuracy equivalent to that of PACoGen, a LUT of size $2^9 \times 10$ is used, with rectangular multipliers selected to save resources. The proposed algorithms were verified with exhaustive and random/corner case tests for various posit bitwidths, with non-restoring division providing exact answers and iterative dividers differing from the exact solution by ≤ 5 units in the last place (ULPs).

B. ASIC Synthesis

To evaluate the proposed designs, we measure area, delay, power, and energy. Standard cell synthesis is performed using Synopsys Design Compiler with a 45-nm library by TSMC, under default configuration. Results are presented for 16 and 32-bit units, as dividers with smaller bitwidths can be easily implemented with a LUT. For comparison, we synthesize the PACoGen combinational posit divider [8], which implements the Newton-Raphson algorithm using a LUT of size $2^8 \times 9$ for the initial guess and the same number of iterations as proposed in our work. To the best of our knowledge, this is the only available open-source implementation of posit division. Synthesis results are presented in Fig. 4.

The non-restoring divider requires less area and power than iterative implementations (53% area, 65% power compared to the area-efficient Newton-Raphson implementation), but is 3-4 times slower than multiplicative operators and consume up to $2\times$ more energy.

Multiplicative dividers exhibit similar results in terms of area and power. The Goldschmidt divider is faster than the Newton-Raphson implementation, since it uses non-dependent multiplications that can be performed in parallel.

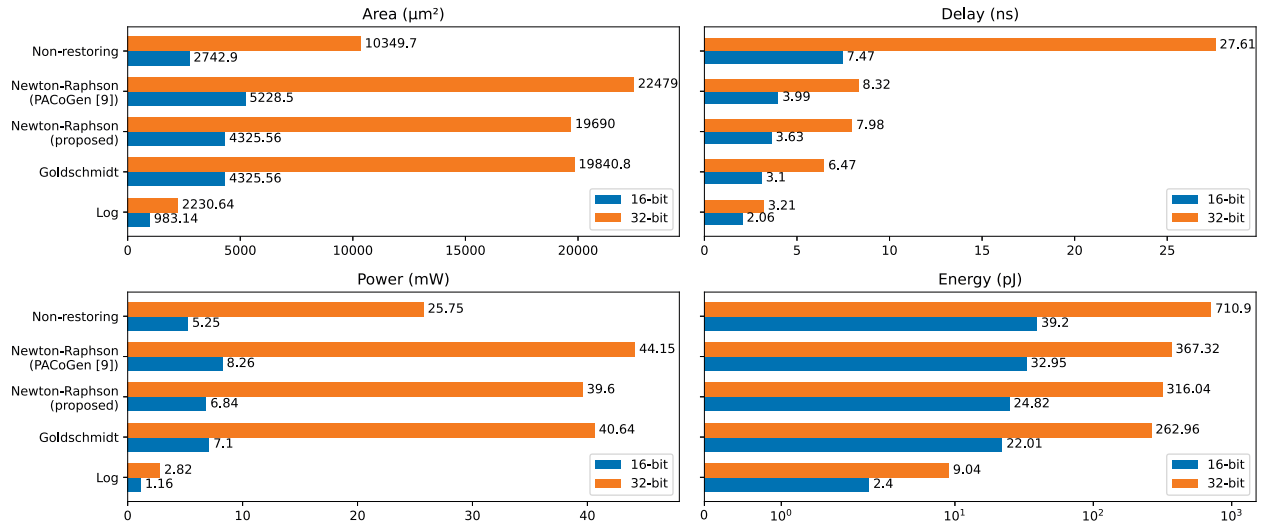

Fig. 4. Synthesis results for different implementations of posit division.

The Newton-Raphson implementation proposed in this work improves upon the PACoGen version, with savings in area, delay, power, and energy consumption by a factor of 1.142×, 1.043×, 1.115×, and 1.163×, respectively, despite requiring a larger LUT and wider multipliers.

Finally, logarithm-approximate division units use significantly less hardware than iterative units, with the 32-bit approximate divider using even less area than the 16-bit exact non-restoring unit. Compared with the 32-bit Newton-Raphson unit, the area is reduced by a factor of 8.8×. Also, the approximate divider reduces energy consumption by more than 29× compared to Goldschmidt's divider of the same bitwidth. However, the result provided by this unit is not exact in many cases.

V. Conclusions

We implement division algorithms for posit arithmetic, including digit recurrence, iterative, and approximate methods. ASIC synthesis results demonstrate varying trade-offs between area, delay, and power consumption, requiring the selection of the appropriate design for system requirements and implementation characteristics. Our designs use two's complement encoding of posit numbers, reducing hardware requirements compared to prior works. This is the first work proposing an approximate logarithmic posit division, which might be suitable for error-tolerant applications. Evaluation shows that the approximate divider reduces area and energy consumption by a factor of at least 8.8× and 29×, respectively.

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] 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.

[3] 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.

[4] D. Mallasén, R. Murillo, A. A. D. Barrio, G. Botella, L. Piñuel, and M. Prieto-Matias, "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.

[5] 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.

[6] 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, pp. 1–5.

[7] 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.

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

[9] D. N. Rao, G. Sai Charan, D. V. Venkata Sairam, and K. S., "Posit Number Division using Newton-Raphson method," in *2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, 2021, pp. 1–6.

[10] R. Murillo, D. Mallasén, A. A. Del Barrio, and G. Botella, "Comparing Different Decodings for Posit Arithmetic," in *Conference on Next Generation Arithmetic (CoNGA)*, vol. 13253, 2022, pp. 84–99.

[11] F. Xiao, F. Liang, B. Wu, J. Liang, S. Cheng, and G. Zhang, "Posit Arithmetic Hardware Implementations with The Minimum Cost Divider and SquareRoot," *Electronics*, vol. 9, p. 1622, 2020.

[12] A. Raveendran, S. Jean, J. Mervin, D. Vivian, and D. Selvakumar, "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.

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

[14] J. N. Mitchell, "Computer Multiplication and Division Using Binary Logarithms," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, 1962.

[15] R. Murillo, A. A. Del Barrio Garcia, G. Botella, M. S. Kim, H. Kim, and N. Bagherzadeh, "PLAM: a Posit Logarithm-Approximate Multiplier," *IEEE Trans. on Emerging Topics in Computing*, vol. 10, no. 4, 2022.