Efficient Data Extraction Circuit for Posit Number System: LDD-based Posit Decoder

Jianchi Sun, *Student Member, IEEE* and Yingjie Lao, *Senior Member, IEEE*

***Abstract*—Since being proposed in 2017, the posit number system has attracted much attention due to its advantages over the IEEE 754 standard floating-point format for better dynamic range and higher accuracy, which are crucial to many applications such as neural networks. Those advantages are yielded from a varying-length segment, regime bits, which lead to the size variations for all rest components except the sign bit. Consequently, it requires an extra decoding process to extract the numerical value of a posit number. The state-of-the-art posit decoder is designed based on a leading one/zero detector. However, we find that this conventional method holds implicit redundancy when dealing with binary numbers. In this paper, we design a novel hardware architecture, i.e., the leading difference detector, to optimize the circuit operation by eliminating the redundancy. The experimental results show that the proposed architecture can decrease the delay and power consumption by over 41% compared to the conventional designs for 8-bit, 16-bit, 32-bit, and 64-bit posit decoders.**

***Index Terms*—posit number system, floating-point, decoder, circuit design, machine learning.**

I. Introduction

Since the universal number was proposed, IEEE 754 Standard floating-point format [1] has become one of the most commonly used number formats. In searching for higher accuracy and dynamic range to better serve modern applications, [2] designed *posit* number in 2017, a drop-in replacement for IEEE 754, as claimed by the developers.

With the same bit size as floating-point, posit number offers a more flexible trade-off than floating-point between decimal accuracy and dynamic range. Compared with floating-point, posit shows many advantages such as larger dynamic range, higher accuracy, better closure, and overflow resistance. Besides, [3] found that posit can save the hardware cost such that an n-bit IEEE 754-2008 adder and multiplier can be safely replaced by an m-bit Posit Arithmetic Units adder and multiplier where $m < n$. In addition, posit number achieves superior performance in computing some special functions. For example, it only requires simple bit shifting and flipping to estimate the value of the sigmoid function $(1/(1 + e^{-x}))$ with posit number.

Recent works have been exploring its applications by leveraging the advantages of posit numbers. For instance, [3]–[5] designed ASIC architectures for posit arithmetic core generator, [6]–[8] exploited the implementations of posit system on FPGA, [9] applied approximate computing to the posit system, [10], [11] designed efficient multiplier for posit number, and [12]–[15] adapted posit number system to deep neural networks (DNN). For instance, one of the biggest challenges for DNN is the DRAM capacity and speed limits due to its massive trainable parameters [16], [17]. Alleviating the challenge, techniques like low-precision arithmetic [18], [19] are studied to lessen the data size. Enlightened by this approach, researchers found posit number a great fit to neural network applications due to its high dynamic range [20], which means the users can either have higher dynamic range with the same number size, or similar dynamic range with smaller number size, compared to the floating-point.

A posit number is composed by four parts: sign bit (s), regime bits (r), exponent bits (e), and fraction bits (f), as shown in Fig. 1. The length of the regime bits can vary, which may even take over the space of fraction bits and exponent bits for different number values. This key property yields the trade-off between decimal accuracy and dynamic range. However, it requires an extra decoding/data extraction process to obtain the sizes and values for each component before arithmetic calculation.

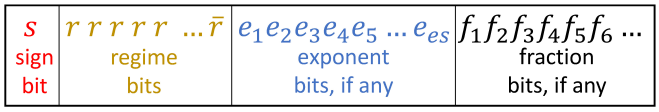


Fig. 1: Generic posit format for finite, nonzero values.

To perform the decoding process for posit, the state-of-the-art posit decoder [3]–[5], [21], [22] are based on hardware structures named leading one detector (LOD) or/and leading zero detector (LZD) [23] (some papers call them leading one/zero counter), whose function is to detect the size of the regime bits. After regime size is obtained, the decoder then 'flush out' the specific values for all parts and get them ready for the subsequent arithmetic calculations. However, we find that this design does not fully utilize the hardware when encoding the regime's size into a binary number and decoding it for bit shifting, and the implied redundancy introduces extra delay and power consumption. In this paper, to address this weakness, we design a novel circuit structure, leading difference detector (LDD). Then we implement a posit number decoder based on the LDD. Our experimental results show that the proposed LDD-based posit decoder can reduce the delay and energy consumption by about 60% and 50%, respectively, compared to the conventional LOD decoder for 8-bit, 16-bit, 32-bit, and 64-bit posit numbers. We believe this is a firm improvement as LOD/LZD is a 'must use' structure for the posit decoder.

Jianchi Sun and Yingjie Lao are with the Holcombe Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634, USA (e-mail: {jianchs, ylao}@clemson.edu).

The rest of the paper is organized as follows: Section II reviews the basic principle of the posit number system, the current decoding methodology, and the corresponding circuit design. Then, our proposed efficient LDD-based posit number decoder is presented in Section III. In Section IV, we present the experimental results to verify the advantages of our design. Finally, Section V concludes this paper.

II. Background

A. Posit Number System

The universal number (*unum*) has several types. The "type I" unum is a superset of IEEE 754 Standard floating-point format, which is widely used today, but it requires extra management to activate variable length. Unlike the "type I" unum that is used for expressing interval arithmetic, the "type II" unum is designed based on the projective reals, which means it becomes a pointer to the values instead of the value itself. Although having many ideal mathematical properties, the "type II" unum has exaggerated hardware cost since it requires a bigger lookup table for most operations [24]. As a representative of the "type III" unum, posit number system is designed to create a hardware-friendly version of the "type II" unum.

As shown in Fig. 1, a posit number is composed by: sign bit (s), regime bits (r), exponent bits(e), and fraction bits (f), together with two pre-known parameters: number size (N) and exponent size (es).

The highest bit will always be the sign bit, where '0' stands positive and '1' stands negative. When negative, we need to take the 2's complement before decoding the rest parts. The very next part is the regime bits. To decode it, we need to count the number of consecutive 0s or 1s after the sign bit, and the last bit of regime bits will be the first different bit. For m consecutive 0s, regime $r = -m$, while for m consecutive 1s, regime $r = m - 1$. If all the bits except the sign bit are the same, they will all be counted as m. One 4-bit decoding example is shown in Table I.

TABLE I: Regime Bits Decoding Example

Regime Bits	000	$001X$	$01XX$	$10XX$	$110X$	111
r	-3	-2	-1	0	1	2

After the regime bits, the very next es bits will be e. If there are not enough bits left, e equals the remaining bits or just 0 if no bit is left. After decoding all the parts mentioned above, the rest of the bits are all f, and $f = 0$ when there is no bit left. With all the extracted data, the value of a posit number can be expressed as:

$$(2^{2^{es}})^r \times 2^e \times 1.f \tag{1}$$

Due to the variable bit sizes for each component of a posit number, an extra data extraction/decoding process is necessary to perform arithmetic operations.

B. Leading One/Zero Detector

To decode a posit number and perform data extraction, LOD and LZD are employed by the state-of-the-art studies [3]–[5], [21], [22]. Those hardware structures detect and output the location for the first $0/1$ for a binary number. The circuit design for fast LOD used by, to the best of our knowledge, all the recent studies, is shown in Fig. 2. A LOD/LZD has two outputs, $K = (i - 1)$ indicates the first $0/1$ occurs at the $i - th$ bit (counting from left), and $Vld = 0$ when no $0/1$ is detected. For example, an LOD will output $K = 101$ to indicate that the first 1 occurs at $6th$ bit when the input number starts with '000001'. Then, based on the outputs of LOD, the posit decoder can obtain the value for regime bits and flush out the rest parts of the posit number with a shifter. Since the posit's decoding process typically finishes within one clock cycle, the 'shifter' here is actually a selector, which selects the correct output from all possible shifting results that are pre-defined. As LOD and LZD have a similar circuit design and current posit decoders only use one of those combined with inverters to handle all the input patterns (as shown in lines 7 and 8 of Algorithm 1), we implement an LOD-based decoder as the baseline comparison in this paper. The detailed decoding process with LOD can be found in Algorithm 1.

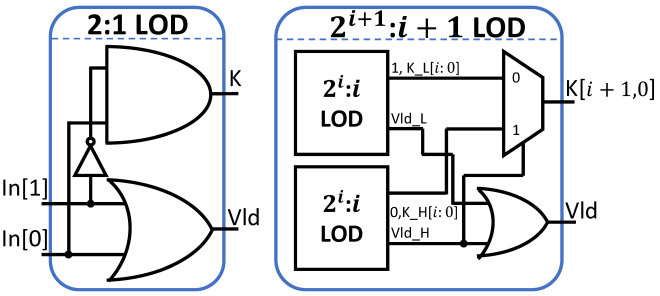


Fig. 2: Circuit design for LOD.

Algorithm 1 Posit Data Extraction with LOD

1: **Input:** $IN[N-1:0]$
2: **Outputs:**
$Sign(s), Regime(r), Exponent(e), Fraction(f),$
$Zero(z), Infinity(inf)$
3: **Pre-defined:** $Input\_Size(N), Exponent\_Size(ES)$
4: $z \leftarrow$ **NOR**$(IN[N-1:0])$
5: $inf \leftarrow IN[N-1]$**&**(**NOR**$(IN[N-2:0]))$
6: $XIN \leftarrow IN[N-1]?(\sim IN[N-2:0]+1) : IN[N-2:0]$
(Take 2's complement if $IN[N-1] = 1$)
7: $LIN \leftarrow XIN[N-2]?(\sim XIN[N-2:0]) : XIN[N-2:0]$
8: $K \leftarrow$ **Leading One Detector**(LIN)
9: $r \leftarrow XIN[N-2]?(K-1) :\sim (K-1)$
10: $temp \leftarrow XIN << (K+1)$
11: **if** $N - K - 2 > ES$ **then**
12: $e \leftarrow$ Highest ES bits of $temp$
13: **else**
14: $e \leftarrow$ Highest $(N - K - 2)$ bits of $temp$
15: **end if**
16: $f \leftarrow temp << ES$

Although this design is intuitive, there are several places that we can further optimize in the hardware implementation. During the decoding process of the example we mentioned above,

the LOD encodes the first 1's location into a binary number '101', then the 'shifter' decodes this number and makes the selection. Such redundancy introduced by the encoding and decoding of binary numbers will consume extra power and circuit area.

III. LDD-based Posit Decoder

A. Leading Difference Detector

In this section, we present the design of a novel posit decoding circuit based on a leading difference detector (LDD), which eliminates the redundant binary decoding process of the conventional decoder.

The decoding process with LDD is shown in Algorithm 2, where N is the posit number bit width, z and o mean all the bits after taking the 2's complement are 0s or 1s. For a better illustration, an example is provided later in this section to explain this algorithm. In essence, the LDD generates a binary indicator '*LDD*' instead of a binary number based on the location of the first different bit. This indicator has the property that its $(i-1)-th$ bit will be '1' and the rest will be '0' if the input's first difference occurs at the $i-th$ bit. An example is shown as Fig. 3, where the output of LDD '00010000000' indicates that the first difference occurs at the $5th$ bit. Please note that the output size of LDD will be 1 bit smaller than its input size, as the difference will never occur at the very first bit. Then, based on the obtained value of *LDD*, the corresponding output for each component will be generated by a customized selection circuit.

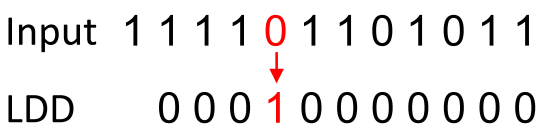


Fig. 3: LDD output format example.

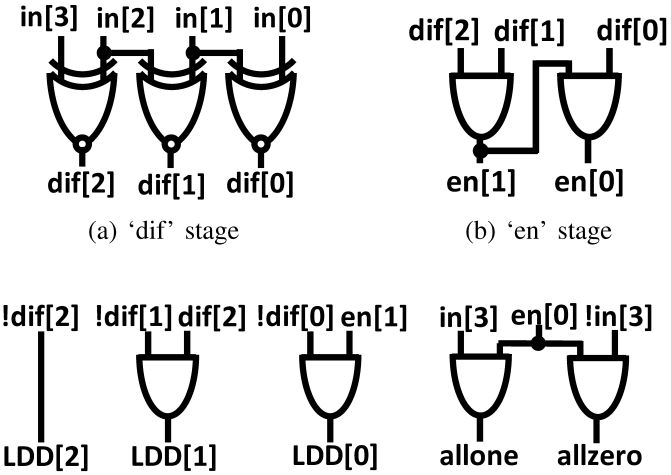
For a better illustration, we provide an example of the circuit design with a 4-bit input XIN (here N = 5 since the sign bit is removed after taking 2's complement at Algorithm 2, line 4) in Fig. 4. There are 3 stages in the LDD circuit:

- The '*dif*' stage (Fig. 4(a)) checks the differences for all adjacent bits in the way that $dif[i]$ = '0' when $in[i+1] \neq in[i]$ (Algorithm 2, line 5-7).
- The '*en*' stage (Fig. 4(b)) implements a priority arbiter [25] to examine the existence of the differences among the higher bits with AND logic (Algorithm 2, line 9). When a difference is detected among the higher bits, the current $en[i]$ will be locked at '0' ignoring the value of $dif[i]$. A straightforward implementation is shown as Fig. 5(a), which uses fewest logic gate but have largest delay. To balance the cell number and circuit delay, we start from implementing a large tree-structured AND gate for $en[0] = AND(dif[(N-2):0])$, and then add 2-to-1 AND gates onto that large AND gate to obtain the rest en. Fig. 5(b) illustrates the design for a 3-bit output 'en' stage, where the red AND gate is added to generate $en[1]$ (Algorithm 2, line 8-10).

Algorithm 2 Posit Data Extraction with LDD

1: **Input:** $IN[N-1:0]$
2: **Outputs:**
$Sign(s), Regime(r), Exponent(e), Fraction(f),$
$AllZero(z), AllOne(o)$
3: **Pre-defined:** $Input\_Size(N), Exponent\_Size(ES)$
4: $XIN \leftarrow IN[N-1]?(\sim IN[N-2:0]+1) : IN[N-2:0]$
(Take 2's complement if $IN[N-1]=1$)
5: **for** $i = 0 : (N-3)$ **do**
6: $dif[i] \leftarrow XIN[i] \odot XIN[i+1]$
7: **end for**
8: **for** $i = 0 : (N-4)$ **do**
9: $en[i] \leftarrow$ **AND**$(dif[(N-3):i])$
10: **end for**
11: $LDD[N-3] \leftarrow\sim dif[N-3]$
12: $LDD[N-4] \leftarrow dif[N-3]$ **&** $\sim dif[N-4]$
13: **for** $i = 0 : (N-5)$ **do**
14: $LDD[i] \leftarrow\sim dif[i]$ **&** $en[i+1]$
15: **end for**
16: $z \leftarrow XIN[N-2]$ **&** $en[0]$
17: $o \leftarrow\sim XIN[N-2]$ **&** $en[0]$
18: $s \leftarrow IN[N-1]$
19: $r, e, f \leftarrow$ Corresponding values from NAND selection arrays based on current LDD. Follow the principle that introduced in Section II-A. Circuit design is introduced in Section III-B.

- The output stage computes the final decision of LDD based on the en (Algorithm 2, line 14). Besides, it uses $en[0]$ to check if all the bits are 0s or 1s, as $en[0] = 0$ only when no difference is detected (Algorithm 2, line 11-17).

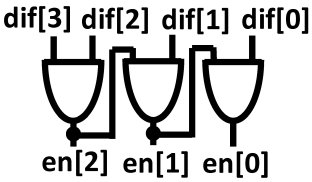


(a) 'dif' stage (b) 'en' stage

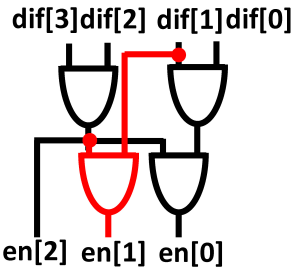
(c) Output stage

Fig. 4: Example circuit for 4-bit LDD.

By removing the process of 'encoding the first 1's location into a binary number' introduced by LOD, the LDD circuits utilize the AND-gate tree instead of the multiplexer (MUX) tree for LOD (Fig. 2) to identify the first difference's location. Since the tree sizes for LDD and LOD are similar, better performance on LDD with simplified logic gates can be expected. Specific comparisons are shown in Section IV.



(a) Fewest logic gates.



(b) Balanced design. The red AND gate is added to generate $en[1]$.

Fig. 5: 3-bit output 'en' stage.

B. Bit Shifter

As we mentioned above, a 'bit shifter' is typically implemented as a selection circuit, which selects the corresponding output from all possibilities based on the input of '# of bits to be shifted'. The conventional posit decoders with LZD/LOD utilize MUX for the shifter [26] as illustrated in Fig. 6(a), where '*o_i[j]*' indicates the corresponding output value for *out[i]* when left shifting j bits. As the first difference will never occur at the first bit, the $o\_i[1]$ is always unused (marked as red) for all cases.

In contrast, with LDD, suppose the input size for LDD is N bits, we can simply express the *out[i]* as:

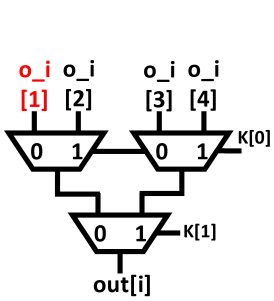
$$out[i] = \overline{(o\_i[N]LDD[0])}...\overline{(o\_i[j]LDD[N-j])}..., \quad (2)$$

which can be implemented as a tree-structured NAND selection array. A 4-bit example is shown in Fig. 6(b), where the LDD's input size $N = 5$. With the 'en' stage from LDD module, only $LDD[N-j]$ will be '1' and the rest bits of LDD will be '0' when the first difference appears at $in[N-j]$. With this characteristic, the NAND gate that takes $LDD[N-j]$ as input will output $\sim o_i[j]$, while all the rest of NAND gates in the input stage will output 1 since the rest bits of LDD are '0'. Then with another stage of NAND operation, we will have $out[i] = o_i[j]$, according to Eq. 2, which achieves the same selection function with a conventional 'shifter'. In addition, since LDD has the bit-to-bit flexibility, no input bit will be unused here.

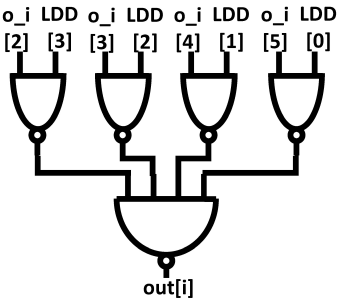
Similar with LDD, the 'shifter' for LDD replaces the MUX tree of conventional shifter with NAND tree by removing the redundancy introduced by the binary numbers, which further optimizes the hardware cost of posit decoder.

IV. Experimental Results

Our experimental results are presented in this section. We implement the LDD-based posit decoder and the LOD-based posit decoder proposed by recent studies [3]–[5], [21], [22] using Verilog HDL and then map them into a 32nm technology node using Synopsys Design Compiler, all circuits are optimized with the same effort level and are driven by identical inverters. Each decoder has three specific circuit designs that are compatible with the 16-bit number system with $ES = 1$, the 32-bit number system with $ES = 3$, and the 64-bit number system with $ES = 4$, according to the posit inventor's recommendation [2]. All the designs are then mapped into a 32nm technology node using Synopsys Design Compiler.



(a) Conventional 4-bit shifter.



(b) 4-bit shifter for LDD.

Fig. 6: Example circuit for 4-bit shifter.

All of our Verilog codes can be found in the GitHub link: https://github.com/JSCooode/LDD_Posit_Decoder. The modules for 16-bit and 64-bit LDD-based decoders are parameterized so that they can be easily configured for any posit system with different number sizes.

TABLE II: Comparison: LDD vs. LOD [3]–[5], [21], [22]

	16-Bit		32-Bit		64-Bit	
	LDD	LOD	LDD	LOD	LDD	LOD
Delay (ns)	1.44	2.75	1.8	4.61	1.88	4.85
Power (μW)	22.2	19.6	66.8	56.1	231.7	167.8
Area (μm^2)	369	489	1201	1461	4474	4733
P-D Prod.	1×	1.7×	3.8×	8.1×	13.6×	25.5×

Our experimental results are summarized in Table II, where 'P-D Product' stands for 'power–delay product', which represents the average energy consumption under the same throughput. The result shows that the delay of the LDD-based decoder is decreased by 47.6%, 60.1%, and 61.2% for 16-bit, 32-bit, and 64-bit designs, respectively, compared to the LOD-based decoder. Meanwhile, the average energy consumption of the LDD-based decoder is also about 50% smaller than the LOD-based decoder for 16-bit, 32-bit, and 64-bit posit numbers. In addition, the LDD-based decoders also have smaller area consumption. For a better illustration, we plot the changes of P-D product for both LDD and LOD under different bit sizes in Fig. 7, from which we can see that LDD-based decoder outperforms LOD-based decoder for all input sizes, and an increasing advantage of LDD-based decoder when expanding the input size can be observed. This indicates that the LDD-based decoder will be even more applicable for modern computer systems that work with bigger data size (like the upgrade from 32-bit systems to 64-bit systems). In

addition, we also implement the 32-bit LOD introduced in [11] with simulation results: (Delay: $4.37ns$, Power: $51.2\mu W$, Area: $1948\mu m^2$), which shows that our design outperforms the recent study.

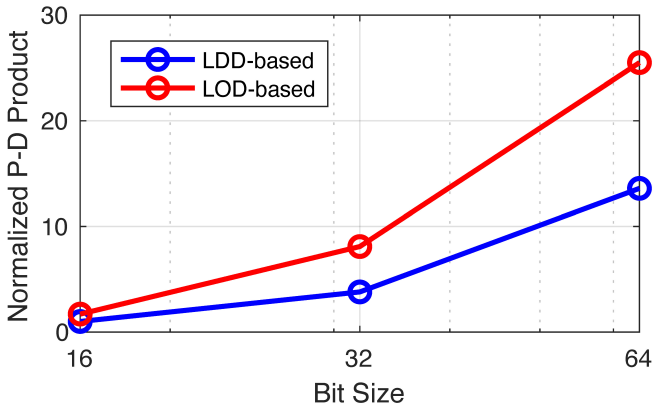


Fig. 7: LDD-based decoder vs. LOD-based decode P-D product comparison.

To decrease the memory cost for neural networks, recent research studied the feasibility of decreasing the data size to 8 bits [27]. As posit shows great potential with its high dynamic range, we also implement the LDD-based decoder for extremely small-sized systems (8-bit, $ES = 1$) to evaluate the novelty of our design in small-data-size use cases and compare it with state-of-the-art decoder as shown in Table III, where LDD still outperforms in all the aspects.

TABLE III: Comparison for Extremely Small Data Size

	Delay (ns)	Power (μW)	Area (μm^2)	P-D Prod.
LDD-based	1.16	9.11	118	1×
LOD-based	1.28	10.21	136	1.24×

V. Conclusion

In this paper, we presented an efficient circuit structure named LDD and designed a novel decoder based on that to perform data extraction for posit numbers. By eliminating the redundant binary number encoding and decoding processes, our proposed LDD-based posit decoder approximately halves the delay and energy consumption with a smaller hardware cost for 16-bit, 32-bit, and 64-bit decoders compared to the conventional design. Future work will be directed towards the design of an efficient posit arithmetic core based on the proposed LDD and the corresponding evaluation of the overall performance on a wide range of applications.

References

[1] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

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

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

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

[5] 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)*, 2019, pp. 106–113.

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

[7] M. K. Jaiswal and H. K.-H. So, "Universal number posit arithmetic generator on fpga," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1159–1162.

[8] J. Hou, Y. Zhu, S. Du, and S. Song, "Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on fpga," *Journal of Signal Processing Systems*, vol. 91, no. 10, pp. 1137–1148, 2019.

[9] R. Murillo, A. A. D. B. Garcia, G. Botella, M. S. Kim, H. Kim, and N. Bagherzadeh, "Plam: a posit logarithm-approximate multiplier," *IEEE Transactions on Emerging Topics in Computing*, 2021.

[10] 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.

[11] L. B. R. K, H. R. S, K. Puli, S. R. R. Annapalli, and V. Pudi, "Design of energy efficient and low delay posit multiplier," in *2023 36th International Conference on VLSI Design and 2023 22nd International Conference on Embedded Systems (VLSID)*, 2023, pp. 1–6.

[12] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "A fast approximation of the hyperbolic tangent when using posit numbers and its application to deep neural networks," in *International Conference on Applications in Electronics Pervading Industry, Environment and Society*. Springer, Cham, 2019, pp. 213–221.

[13] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep positron: A deep neural network using the posit number system," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1421–1426.

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

[15] S. H. F. Langroudi, T. Pandit, and D. Kudithipudi, "Deep learning inference on embedded devices: Fixed-point vs posit," in *1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE, 2018, pp. 19–23.

[16] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *249th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[17] Y. Long, D. Kim, E. Lee, P. Saha, B. A. Mudassar, X. She, A. I. Khan, and S. Mukhopadhyay, "A ferroelectric fet-based processing-in-memory architecture for dnn acceleration," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 5, pp. 113–122, 2019.

[18] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," *arXiv preprint arXiv:1802.04680*, 2018.

[19] S. Hashemi, N. Anthony, H. Tann, R. I. Bahar, and S. Reda, "Understanding the impact of precision quantization on the accuracy and energy of neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1474–1479.

[20] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 174–187, 2020.

[21] Z. Kleijweg, "Hybrid posit and fixed point hardware for quantized dnn inference," *Circuits and Systems*, 2021.

[22] L. Forget, Y. Uguen, and F. de Dinechin, "Comparing posit and ieee-754 hardware cost," *HAL Open Science, hal-03195756v3*, 2021.

[23] K. H. Abed and R. E. Siferd, "Vlsi implementations of low-power leading-one detector circuits," in *Proceedings of the IEEE SoutheastCon 2006*. IEEE, 2006, pp. 279–284.

[24] J. L. Gustafson, "A radical approach to computation with real numbers," in *Supercomputing Frontiers and Innovations*, 2016, p. 3(2):38–53.

[25] A. Bystrov, D. J. Kinniment, and A. Yakovlev, "Priority arbiters," in *Proceedings of Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, 2000, pp. 128–137.

[26] Y. O. M. Moctar, N. George, H. Parandeh-Afshar, P. Ienne, G. G. Lemieux, and P. Brisk, "Reducing the cost of floating-point mantissa alignment and normalization in FPGAs," in *Proceedings of international symposium on Field Programmable Gate Arrays*, 2012, pp. 255–264.

[27] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," *Advances in neural information processing systems*, vol. 31, 2018.