The Accuracy and Efficiency of Posit Arithmetic

Stefan Dan Ciocirlan*+, Dumitrel Loghin*, Lavanya Ramapantulu, Nicolae Ţăpuş+, Yong Meng Teo*

*Department of Computer Science, National University of Singapore, Singapore

+Department of Computer Science, University Politehnica of Bucharest, Romania

Email: *{dumitrel, teoym}@comp.nus.edu.sg, lavanya.r@gmail.com, +{stefandan, ntapus}@cs.pub.ro

***Abstract*—Motivated by the increasing interest in the posit numeric format, in this paper we evaluate the accuracy and efficiency of posit arithmetic in contrast to the traditional IEEE 754 32-bit floating-point (FP32) arithmetic. We first design and implement a Posit Arithmetic Unit (PAU), called *POSAR*, with flexible bit-sized arithmetic suitable for applications that can trade accuracy for savings in chip area. Next, we analyze the accuracy and efficiency of *POSAR* with a series of benchmarks including mathematical computations, ML kernels, NAS Parallel Benchmarks (NPB), and Cifar-10 CNN. This analysis is done on our implementation of *POSAR* integrated into a RISC-V Rocket Chip core in comparison with the IEEE 754-based Floting Point Unit (FPU) of Rocket Chip. Our analysis shows that *POSAR* can outperform the FPU, but the results are not spectacular. For NPB, 32-bit posit achieves better accuracy than FP32 and improves the execution by up to 2%. However, *POSAR* with 32-bit posit needs 30% more FPGA resources compared to the FPU. For classic ML algorithms, we find that 8-bit posits are not suitable to replace FP32 because they exhibit low accuracy leading to wrong results. Instead, 16-bit posit offers the best option in terms of accuracy and efficiency. For example, 16-bit posit achieves the same Top-1 accuracy as FP32 on a Cifar-10 CNN with a speedup of 18%.**

***Index Terms*—posit, floating-point, IEEE 754, RISC-V, accuracy, efficiency**

I. Introduction

In the last few years, posit floating-point numeric format [16] has gained traction in the research community [6], [9], [11], [14]. This format is an alternative to the classic IEEE 754 standard [2] implemented by the vast majority of modern hardware. However, IEEE 754 implementations use significant chip area and power because they need to handle many corner cases and exceptions described by the standard. Some studies show that this standard is error-prone and its different implementations may produce different results [15]. These issues become more relevant in modern computing dominated by the explosion of Machine Learning (ML).

Among the alternatives to IEEE 754, *unum* [15] and its third version, named *posit* [16], [23], were introduced by John L. Gustafson to solve some of the issues of IEEE 754 floating-point representation. Compared to IEEE 754, posit has variable length fields to represent the exponent and the fraction of a real number. Hence, posits can represent small numbers more accurately by reserving more bits for the fraction and fewer bits for the exponent. Moreover, posits have only two special representations, namely for 0 and not-a-real (NaR), whereas IEEE 754 reserves many binary representations for not-a-number (NaN). This feature, together with the non-existence of subnormal, make posit implementations simpler. In this paper, we address the following research questions, (i) are posits more accurate and efficient than IEEE 754? and (ii) what is a good trade-off between accuracy and time-energy efficiency when employing lower bit-size posits?

To answer these questions, we propose an alternative hardware-software approach by (i) designing and implementing a Posit Arithmetic Unit named ***POSAR*** in a Rocket Chip-based [3] RISC-V [25] core replacing its Floating Point Unit (FPU), and (ii) modifying existing software to run on this system. RISC-V [25] is an open-source architecture with limited available hardware implementations. However, RISC-V is very promising due to its energy efficiency and modular ISA, thus, it is timely to explore this new CPU architecture. Without modifying the ISA, we use the F extension of the RISC-V specification but change the internal processor representation of floating-point numbers to posit. To address the challenge of executing the same application software on the modified hardware, we make minor high-level code changes and convert IEEE 754 constants to posit.

In summary, we make the following contributions:

- We implement a Posit Arithmetic Unit named ***POSAR***[1] in Chisel to replace the traditional Floating Point Unit (FPU) of a Rocket Chip RISC-V core. In addition, we implement a Scala library for posit arithmetic to test *POSAR*.
- *POSAR* supports any posit size and exponent size, but in this paper we instantiate it for three sizes: 8, 16, and 32 bits, respectively. We test these instantiations in both simulation mode and on an Arty A7-100T FPGA.
- Our evaluation of *POSAR* vs. Rocker Chip's original FPU shows that 32-bit posit achieves at least the same accuracy as 32-bit IEEE 754 floating-point, while logging fewer cycles, but using more FPGA resources. For example, 32-bit posit uses 30% more FPGA resources but it is 30% faster compared to 32-bit IEEE 754 float[2].
- In contrast to other works [6], [20], we show that 8-bit posit does not reach the same accuracy as 32-bit IEEE 754 when running a CNN. On the other hand, 16-bit posit offers the best option in terms of accuracy and efficiency. For example, 16-bit posit achieves the same Top-1 accuracy as 32-bit IEEE 754 on a Cifar-10 CNN with a speedup of 18%.

[1]The source code and an extended paper can be found at https://github.com/dloghin/posar.

[2]These results are implementation-specific. That is, we compare specific implementations of the IEEE 754 floating-point and posit standards.

In the next section, we provide details about the posit format and we summarize related works. In Section III we present the design and implementation of *POSAR*. In Section IV we evaluate our approach before concluding in Section V.

II. Background and Related Work

A. Posit Format

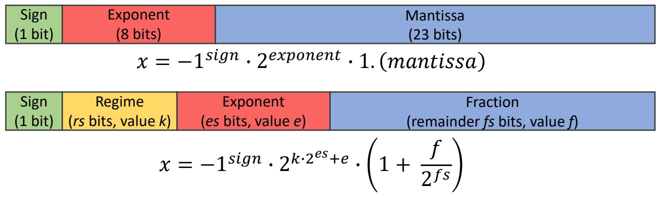
Posit [23] is a real number representation that aims to improve the widely-used IEEE 754 floating-point standard implemented by the majority of modern processors. A 32-bit, single-precision floating-point in IEEE 754 comprises three fields, as shown in Figure 1 (top), namely (i) a sign field of 1 bit, (ii) an exponent field of 8 bits and (iii) a fraction or mantissa field of 23 bits. In contrast to IEEE 754 which reserves many binary representations for the special number NaN (Not-a-Number), posit format has only two special numbers, 0 and NaR (Not-a-Real). If the binary representation of the posit has all the bits equal to zero, except for the first bit from the left which represents the sign, then it is a special number. If the sign bit is 0 then the special posit has the value 0, otherwise it represents the posit NaR. Compared to IEEE 754, posit representation comprises an additional field named *regime* which determines the final exponent value together with the exponent field, as shown in Figure 1 (bottom). In posit, the regime and fraction fields are variable, while the exponent field is customizable. In fact, a posit format can be described by its total size, ps, and its exponent size, es.

In a posit, the regime field follows the 1-bit sign and continues as long as the bits have the same value r_i, followed by a bit of opposed value. The number of regime bits of the same value, rn, is used to determine the value k as $k = -rn$ if $r_i = 0$, or $k = rn-1$ if $r_i = 1$. k is a factor that multiplies the maximum value of the exponent field, 2^{es}, to which the actual value of the exponent field, e, is added to determine the final exponent $(k \cdot 2^{es} + e)$. This feature of elastic exponent field allows a larger fraction field, hence, a higher representation accuracy compared to the fixed 23-bit mantissa of the IEEE 754 format. However, this higher accuracy occurs only in a range called the "golden zone" [9] which can be useful in scientific applications.

To improve the rounding error and to abide by mathematical properties such as associativity and distributivity, the posit standard introduces a *quire* [23] which is a long accumulator [9]. However, the implementation of a quire uses 10 times more area and increases the latency by 8 times compared to posit without quire [9]. For example, an implementation of an unum type co-processor in SMURF [5] for a RISC-V Rocket Chip uses 9 times more area and consumes 12 times more energy than the 64-bit FPU of the Rocket-Chip. Thus, in this paper, we decided not to implement a quire in our *POSAR*.

B. Posit Implementations in Hardware

The closest implementation of a Posit Arithmetic Unit (PAU) to *POSAR* is PERI, a posit-enabled RISC-V core presented in a preprint [24]. Similar to our work, PERI implements a posit unit capable of executing RISC-V F extension



Sign (1 bit)
Exponent (8 bits)
Mantissa (23 bits)
$x = -1^{sign} \cdot 2^{exponent} \cdot 1.(mantissa)$
Sign (1 bit)
Regime (rs bits, value k)
Exponent (es bits, value e)
Fraction (remainder fs bits, value f)
$x = -1^{sign} \cdot 2^{k \cdot 2^{es}+e} \cdot \left(1 + \frac{f}{2^{fs}}\right)$

Fig. 1: IEEE 754 single-precision float (top) vs. posit (bottom)

instructions and evaluates the processor on an Arty A7-100T FPGA. However, we implement our project in the Chisel language to integrate it with the Rocket Chip core, while PERI uses Bluespec System Verilog and integrates the unit into the SHAKTI core [12]. PERI uses two posit formats at the same time. Both are 32-bit in size, with one having $es = 2$ and the other $es = 3$. To switch between these two formats, PERI introduces a new instruction, $FCVT.ES$. In contrast, our *POSAR* supports multiple bit-sized posits but uses only one size at a time to keep full compatibility with existing software. We evaluate 8- and 16-bit posits in addition to 32-bit posits in this paper. In terms of benchmarking, both PERI and we use k-means, along with the computation of $sin(x)$ and e. In addition, we evaluate π computation, matrix multiplication, k nearest neighbors, naive Bayes, classification trees, NPB applications and a CNN, while PERI is evaluated on JPEG image processing and fast Fourier transform.

Other works propose incomplete posit arithmetic units [7], [17], [18], [22]. Specifically, [7] presents a hardware generator that can produce posit adders and multipliers, [17] presents an adder/subtractor, [22] presents an adder, subtractor, and multiplier, while [18] implements conversion of IEEE 754 to/from posit, adder, subtractor, and multiplier. Similar to us, [7] show that posit unit operating on the same size (e.g. 32 bits) as an IEEE 754-compliant unit needs slightly more hardware resources. In contrast, [22] observes that posit takes significantly more FPGA resources than IEEE 754.

C. Posit in Scientific Computing and Machine Learning

In [8], the authors evaluate the impact of posit on NPB benchmarks using software emulation. As expected, the emulation leads to a much higher execution time of the program using posit compared to the IEEE 754 format running natively on the hardware. However, the accuracy of 32-bit posit is higher compared to FP32. In [16], the authors show using high-level emulation that 32-bit posit can achieve better accuracy and (potentially) faster execution compared to the IEEE 754 format on the LINPACK benchmark. In contrast, we use a hardware-based approach to better understand the impact of posit on cycle-efficiency, not only on accuracy.

Some works analyze the suitability of using posits in ML applications [6], [11], [20]. [6] presents Deep Positron, a Deep Neural Network (DNN) accelerator that can run on FPGAs, and claim that 8-bit posits can achieve better inference accuracy than 8-bit floats and integers, while being close to the accuracy of 32-bit IEEE 754 floats. We also find that 8-bit posits achieve good accuracy on CNNs, but they produce wrong results for scientific applications or classic ML algorithms. We acknowledge that we use Cifar-10 dataset for the ML application, whereas Deep Positron uses medical low-dimensionality datasets.

Johnson [20] evaluates multiple numeric formats, which include posit, to replace IEEE 754 floats in DNNs. Among others, the author shows that 8-bit posits with 1-bit exponents can achieve similar Top-1 accuracy on a Resnet50 model compared to classic 32-bit floats. Carmichael et al. [6] analyze fixed-point, floating-point and posit representations on DNN showing that posit offers the best accuracy while exhibiting a smaller latency compared to floats. In [11], posits are used to store ML parameters in memory, being converted to classic IEEE 754 floats when computations are performed. The authors claim that posits of smaller size can represent the parameters compared to bigger sized IEEE 754 floats, and hence, save up to 36% of memory space while less than 1% accuracy loss is exhibited.

III. *POSAR*: Design and Implementation

In this section, we briefly describe the implementation of *POSAR*. We wrote the high-level design code in the Chisel language and integrated it into a Rocket Chip tiny core to replace the original Floating Point Unit (FPU) that implements the IEEE 754 standard. *POSAR* is activated during the execution phase of the pipeline. In addition to supporting all the instructions of the F extension of RISC-V [25], *POSAR* is elastic to cater to parameterized sizes for posit and exponent. Using this elastic feature, we evaluate *POSAR* on 8-, 16-, and 32-bit posits (ps) with 1-, 2-, and 3-bit exponents (es), respectively. To verify our implementation, we wrote a posit arithmetic library in Scala and tested the *POSAR* Chisel code using unit testing. In this section, we discuss our internal posit representation, the instructions implemented, and the challenges of running programs on a posit-enabled processor.

Supported Instructions. *POSAR* supports all the instructions of the F extension of RISC-V [25]. For bitwise addition, subtraction, multiplication, and division we used the Chisel build-in operators. We acknowledge that this choice leaves some room for further optimizations. For testing the hardware implementation, we wrote a small library in Scala representing posit numbers and we used it inside unit tests. We hope this library will help others in trying different hardware implementations of posit operations.

Posit Representation. We use an internal posit representation comprising the sign s, the regime k and its size rs, the exponent e and its actual size in the binary representation, ers, the fraction f and its size fs, and one bit sn for the special numbers 0 and NaR. We decode a binary posit representation before performing an operation in *POSAR*. We encode our internal representation into a binary representation at the end of the operation.

IV. Evaluation

A. Setup and Benchmarks

We compare the original 32-bit FPU of Rocket Chip which claims to implement the IEEE 754 standard (***FP32***) with our *POSAR* operating with posits of three bit widths, namely 8-bit with 1-bit exponent denoted by ***Posit(8,1)*** or ***P8***, 16-bit with 2-bit exponent denoted by ***Posit(16,2)*** or ***P16***, and 32-bit with 3-bit exponent denoted by ***Posit(32,3)*** or ***P32***. We wrote the high-level code for *POSAR* in Chisel, integrated it with Rocket Chip [3], and used SiFive's Freedom E310[3] development platform to implement and synthesize our code to run on an Arty A7-100T FPGA. The original Rocket Chip with FPU and Rocket Chip with POSAR run at the same frequency.

To evaluate our approach, we select benchmarks that use floating-point operations. We organize these benchmarks into three levels as follows. Level one benchmarks are used to evaluate both the accuracy and efficiency, and they represent the computation of well-known mathematical constants using series and sequences. In particular, we compute the constants π and e (Euler's number), using numerical series, as shown in Table I. For π, we use Leibniz and Nilakantha series. Since Leibniz series converges slowly, we run it for two million iterations. In contrast, Nilakantha series converges faster, thus, we run it for 200 iterations. For e, we use Euler's series which is fast-converging, thus, we run it for 20 iterations.

Level two consists of kernels that are typically used in data analytics and ML applications, as summarized in Table II. For these kernels, we evaluate the efficiency of our *POSAR* versus the FPU in terms of cycles. The correctness of the results is checked against reference outputs. Next, we briefly describe each kernel. Matrix Multiplication (**MM**) implements the multiplication of two square matrices which is often used in ML and HPC workloads. In our testbed, we can accommodate matrices of size up to $n = 182$. k-means (**KM**) groups a set of multi-dimensional points into k groups based on their Euclidean distance. KM is often used in ML and data analytics applications. k-nearest neighbors (**KNN**) classifies a multi-dimensional point based on the Euclidean distance to its k nearest neighbors. Linear Regression (**LR**) is a kernel used in ML and data analytics. We implement Multivariate Linear Regression which consists of matrix and vector operations. Naive Bayes (**NB**) implements a simple Bayesian model. The Classification Tree (**CT**) kernel is used in ML and data analytics to represent a target variable based on some input attributes. We implement both the creation (training) and usage (inference) of CT. We use Iris dataset[4] as input for level two benchmarks, except MM. This dataset consists of $n = 150$ data points with $m = 4$ dimensions representing flowers. These points belong to $k = 3$ classes.

Level three of our benchmarking suite represents full-fledged applications. In this paper, level three is represented by one NAS Parallel Benchmark (NPB) [4] scientific application and one Convolutional Neural Network (CNN) ML inference

[3]https://github.com/sifive/freedom

[4]https://archive.ics.uci.edu/ml/datasets/iris

application. Specifically, we selected Block Tri-diagonal (**BT**) solver from NPB and we converted all floating-point variables to 32-bit float. We use the verification threshold error, *epsilon* (ϵ), as a measure of accuracy. That is, a smaller ϵ corresponds to a higher accuracy. The CNN model is implemented in Caffe [19] and trained on Cifar-10 [1] dataset. This CNN has 14 layers and the parameters file has a size of 351 kB. Since our wimpy setup cannot accommodate the entire model, we only run nly the last four layers of this CNN, starting from *relu3*. We perform the validation on all 10,000 images of Cifar-10 test dataset by running the executables on the Arty A7-100T FPGA. The prediction results are compared against the reference execution on an x86/64 host.

B. Accuracy and Efficiency

Level One. We evaluate the accuracy and efficiency of posit in comparison with 32-bit, single-precision IEEE 754 floating-point (*FP32*), using level one benchmarks summarized in Table I. The accuracy is measured in terms of exact fraction digits compared to the reference value of the mathematical constant. The efficiency represents the number of cycles taken by Rocket Chip running on the FPGA to execute the meaningful section of the program. For posits, we compute the speedup with respect to the *FP32* execution.

The results presented in Table I show that *P32* achieves similar or better accuracy compared to *FP32*. Moreover, *P32* achieves a speedup of 1.3 compared to *FP32* FPU, when π with Leibniz series is computed. The accuracy of small posit representations, such as *P8*, is low when estimating numerical series. This is expected if we consider the internals of posit representation. Taking $e = 2.7182\ldots$ as example, we first observe that the closest *P8* numbers are 2.625 (`0x55`) and 2.75 (`0x56`). That is, one cannot get better accuracy for e than these two values. Secondly, Euler series leads to an issue regarding the storage in *P8* of the factorial which grows very fast. The maximum value that *P8* can represent is 192, which is less than 6!. Hence, the accuracy of Euler's series becomes worst when the number of iterations grows. For example, when $N = 4$, we get $e = 2.75$, but when $N = 6$ we get $e = 3$.

Posit operations take fewer cycles to complete, thus, applications with higher numbers of iterations exhibit better efficiency. For example, *P32* is 30%, 9%, and 3% faster than *FP32* for π Leibniz with two million iterations, π Nilakantha with 200 iterations, and e with 20 iterations, respectively. Our analysis revealed that this speedup is the result of faster multiplication and division operations on posits. This, in turn, is the result of simple exception and corner case handling.

Level Two. We observe that *P32* and *P16* lead to the same final results as *FP32* when running level two benchmarks while saving up to 6% of the cycles, as shown in Table II. However, LR with *P8* and *P16* produces wrong results. This is because the final results are affected by the wrong value of one of the determinants computed by the program. In fact, all the programs operating with *P8* produce wrong results, except CT. This shows that small size posits are not suitable for some ML kernels that need high numerical accuracy. We shall see below that *P8* performs better on a CNN. This observation is similar to some of the related works [6], [20]. However, we note that our evaluation is done on different datasets and different CNN. On the other hand, *P16* offers a good alternative to *FP32*, when the dynamic range of the application is within the range of *P16*.

Level Three. For the NPB application, *P32* achieves one level of magnitude higher accuracy than *FP32*. For example, setting $\epsilon = 10^{-4}$ in BT leads to successful validation when *P32* is used. On the other hand, *FP32* needs $\epsilon = 10^{-3}$ in BT to pass the validation. Moreover, *P32* exhibits a marginal speedup compared to *FP32*. These results are in correlation to those of level one benchmarks. Since there are more and diverse floating-point operations in BT compared to level one benchmarks, the accuracy gain of *P32* is more visible. On the other hand, the speedup gain is not spectacular because the fraction of operations where posit is faster than IEEE 754 is smaller. For the same reason of very large number of operations in BT, *P8* and *P16* do not exhibit good accuracy. In fact, *P8* cannot even represent accurately all the validation reference values due to its limitted range. For example, the validation reference value $7.38e-5$ of BT cannot be represented by *P8* because its range stops at $2.44e-4$ (`0x1`).

When compared to the reference execution on an x86/64 host, the Cifar-10 CNN with *FP32*, *P32* and *P16* running on our FPGA with a Rocket Chip core exhibit the same Top-1 accuracy as the reference model, namely 68.15%. Even *P8* achieves a reasonable accuracy of 62.68%. In terms of speed, all three posit representations are around 18% faster compared to the execution with *FP32*. The results with *P16* and *P8* are very promising and open-up a series of future optimizations. For example, these formats save respectively half and three-quarters of the memory for representing inputs and parameters compared to 32-bit *FP32* or *P32*. Next, by packing two *P16* and four *P8* operands per instruction, we can reduce the execution time by two and four times, respectively.

We observe that one reason why *P8* exhibits accuracy loss is due to the out-of-range representation of some parameters or input image pixels. There is at least one out-of-range representation for each of the 10,000 input images of the Cifar-10 test dataset. This is in contrast to *P32* and *P16* which can represent these parameters without loss of accuracy. For example, the minimum positive value of the weights of *ip1* layer is 0.000001119 which cannot be represented by *P8*. The closest posit size that can represent this value relatively accurate is *Posit(15,2)* with 0.0000011176 (`0x10b`). We note that scaling cannot be applied for *P8* because of the wide parameter distribution interval, that is, the minimum positive value is 0.000001119 and the maximum one is 87.84.

C. Resource Utilization

As a proxy to the chip area taken by our implementation, we evaluate the FPGA resource utilization of our *POSAR* compared to the original FPU of Rocket Chip. We evaluate the FPGA resource utilization of the entire system, namely SiFive Freedom E310 with a Rocket Chip core that has an

TABLE I: Accuracy and Efficiency (Level One Benchmarks)

Benchmark	**Iterations**	**Accuracy** [actual value | number of exact fraction digits]								**Efficiency** [cycles | speedup]						
		FP32		***P8***		***P16***		***P32***		***FP32***	***P8***		***P16***		***P32***	
π (Leibniz)	2,000,000	3.14159	5	3.5	0	3.14	2	3.14159	5	216,022,827	166,022,835	1.30	166,022,829	1.30	166,022,830	1.30
π (Nilakantha)	200	3.1415929	6	3.125	1	3.141	3	3.1415922	6	57,940	52,937	1.09	52,952	1.09	52,937	1.09
e (Euler)	20	2.7182819	6	2.625	0	2.718	3	2.7182817	6	15,598	15,177	1.03	15,177	1.03	15,177	1.03

TABLE II: Efficiency (Level Two Benchmarks). Gray background means the result is different from the reference.

Benchmark	**Input**	**Speedup (vs. *FP32*)**		
		P8	***P16***	***P32***
Matrix Multiplication (MM)	n = 182	1.0	1.0	1.0
k-means (KM)	Iris dataset n = 150 m = 4 k = 3	1.01	1.01	1.01
k Nearest Neighbours (KNN)		1.10	1.06	1.05
Linear Regression (LR)		-	1.02	1.02
Naive Bayes (NB)		0.98	1.0	1.0
Classification Tree (CT)		6.2	1.03	1.01

FPU/*POSAR*, running on the Arty A7-100T FPGA. While the results here denote savings in terms of resources from an FPGA perspective, similar or even higher savings in terms of the area will be obtained when the design is implemented on an ASIC [10], [21]. Savings in area directly relate to savings in both static and dynamic power and thus are important for low-power constrained applications.

We observe that all the implementations use the same amount of memory resources (Shift-register Look up table – SRL, LUTRAM, and BRAM) which indicates that the comparison involves only the modified FPU with the rest of the system being the same across all implementations. For significant savings in area and power without much loss in accuracy *P16* seems to be a viable option that saves almost 50% of the DSPs which translate to the multiply-accumulate (MAC) units in an ASIC flow. These savings in area should translate to a 50% drop in dynamic power as the MACs account for a higher power compared to flops or other logic [13]. In contrast, *P32* uses 30% more LUTs and 27% more DSPs compared to *FP32*. These results are worse than those reported in [7], which needs only 4% more LUTs compared to the FPU, but similar to the ones reported in [18]. On the other hand, we note that the original FPU of Rocket Chip is a work-in-progress. That is, it may not implement all the corner cases of IEEE 754 standard. Nonetheless, the higher resource utilization of *P32* may be counterbalanced by its speedup which can lead to higher time and energy efficiency compared to *FP32*.

V. Conclusions

In this paper, we explored the opportunity of replacing a traditional IEEE 754 floating-point unit (FPU) with our proposed posit [16] unit named *POSAR* in the context of scientific computations and machine learning (ML). We present the implementation of *POSAR* which replaces the original FPU in a RISC-V core. We compare the numeric accuracy, efficiency in terms of cycles, FPGA resource utilization of three posit sizes compared to 32-bit IEEE 754 floats.

Acknowledgment

This work was supported in part by Singapore Ministry of Education through Academic Research Fund Tier 1.

References

[1] Cifar-10 Dataset, https://www.cs.toronto.edu/ kriz/cifar.html.
[2] IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.
[3] Asanovic et al., The Rocket Chip Generator, *University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
[4] Bailey, *NAS Parallel Benchmarks*, pages 1254–1259, Springer, 2011.
[5] Bocco et al., SMURF: Scalar Multiple-precision Unum Risc-V Floating-point Accelerator for Scientific Computing, *Proc. of Conference for Next Generation Arithmetic*, pages 1–8, 2019.
[6] Carmichael et al., Performance-Efficiency Trade-off of Low-Precision Numerical Formats in Deep Neural Networks, *Proc. of Conference for Next Generation Arithmetic*, 2019.
[7] Chaurasiya et al., Parameterized Posit Arithmetic Hardware Generator, *Proc. of 36th IEEE International Conference on Computer Design*, pages 334–341, 2018.
[8] Chien et al., Posit NPB: Assessing the Precision Improvement in HPC Scientific Applications, Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics*, pages 301–310. Springer, 2020.
[9] De Dinechin et al., Posits: the Good, the Bad and the Ugly, *Proc. of Conference for Next Generation Arithmetic*, pages 1–10, 2019.
[10] Ehliar and Liu, An ASIC Perspective on FPGA Optimizations, *Proc. of International Conference on Field Programmable Logic and Applications*, pages 218–223, 2009.
[11] Fatemi Langroudi et al., Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit, *Proc. of 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications*, pages 19–23, 2018.
[12] Gala et al., SHAKTI Processors: An Open-Source Hardware Initiative, *Proc. of 29th International Conference on VLSI Design*, pages 7–8, 2016.
[13] Garland and Gregg, Low Complexity Multiply-accumulate Units for Convolutional Neural Networks with Weight-sharing, *ACM Transactions on Architecture and Code Optimization*, 15(3):1–24, 2018.
[14] Guntoro et al., Next Generation Arithmetic for Edge Computing, *Proc. of Design, Automation Test in Europe*, pages 1357–1365, 2020.
[15] Gustafson, *The End of Error: Unum Computing*, Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015.
[16] Gustafson and Yonemoto, Beating Floating Point at its Own Game: Posit Arithmetic, *Supercomputing Frontiers and Innovations*, 4(2):71–86, 2017.
[17] Jaiswal and So, Architecture Generator for Type-3 Unum Posit Adder/-Subtractor, *Proc. of IEEE International Symposium on Circuits and Systems*, pages 1–5, 2018.
[18] Jaiswal and So, Universal Number Posit Arithmetic Generator on FPGA, *Proc. of Design, Automation Test in Europe Conference Exhibition*, pages 1159–1162, 2018.
[19] Jia et al., Caffe: Convolutional Architecture for Fast Feature Embedding, *Proc. of 22nd ACM International Conference on Multimedia*, pages 675–678, 2014.
[20] Johnson, Rethinking Floating Point for Deep Learning, *CoRR*, abs/1811.01721, 2018.
[21] Kuon and Rose, Measuring the Gap Between FPGAs and ASICs, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.
[22] Podobas and Matsuoka, Hardware Implementation of POSITs and Their Application in FPGAs, *Proc. of IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 138–145, 2018.
[23] PositWorkingGroup, Posit Standard Documentation Release 3.2-draft, https://bit.ly/3dGspX1, 2018.
[24] Tiwari et al., PERI: A Posit Enabled RISC-V Core, *CoRR*, abs/1908.01466, 2019.
[25] Waterman, *Design of the RISC-V Instruction Set Architecture*, University of California, Berkeley, 2016.