ISSN (e): 2250 – 3005 || Volume, 12 || Issue, 3|| May. - June. – 2022 ||
International Journal of Computational Engineering Research (IJCER)

# A Study of POSIT Arithmetic Implementations

Meesala Sowmya, Kala S, Nalesh S
[1]Undergraduate Student, [2]Assistant Professor
Department of Electronics And Communication Engineering,
Indian Institute of Information Technology Kottayam, Kottayam –626635, Kerala, India.
[2] Assistant Professor,
Department of Electronics,
Cochin University of Science And Technology, Kochi-686620, Kerala, India.
Corresponding Author: Kala S

**ABSTRACT**

Since mankind started counting, there have been multiple numbering systems, each with its own advantages and disadvantages. We use Arabic numerals, which are base-10 system, whereas computers use binary numbers, which are base-2. Representing real numbers accurately and efficiently on devices that can handle only discrete and finite information is challenging. The IEEE standard for floating point numbers is the most common implementation that modern computing systems have adopted. But due to the deficiencies in the standard, different implementations of IEEE-754 are not guaranteed to give same answers. IEEE-754 also suffer from limitations like redundant representation for numbers, signed zeros, overflow/underflow issues, etc. Floating point numbers are known to have large energy and area footprints when implemented in hardware. To address the shortcomings, a new data type called posit was proposed in 2017. Posit numbers are the result of decades of work in creating a viable replacement for floating point numbers, and they have better accuracy, speed, and simpler design. Posit arithmetic has been introduced as a replacement to IEEE 754- floating point arithmetic number system. In this paper we discuss various posit arithmetic implementations in the literature.

**KEYWORDS:** Floating point number system, posit arithmetic, hardware implementations, accuracy.

---

Date of Submission: 13-06-2022 Date of acceptance: 27-06-2022

---

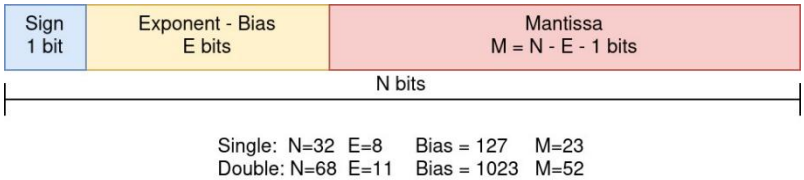
## I. INTRODUCTION

Floating-point numbers, like scientific notations, are represented by an exponent (normally in base two) and a significand, except that the significand must fit on a limited number of bits. Floating point representation is conceptually similar to scientific notation, and it consists of a signed number, also known as the significand, mantissa, coefficient, or ambiguously fraction. This number is encoded as a digit string of a specific length.

It also consist of a signed integer exponent (base two) that changes the magnitude of a number. To find out the value of a floating-point number F, the significand or mantissa M is multiplied by the base β raised to the power of the exponent E, as indicated by,

$$F = M \times \beta^{E}$$

IEEE-754 floating point numbers have the format shown in Fig.1. The number contain following fields: one sign bit S, 2 bit biased exponent, the biased exponent has trailing bits indicating fractions and the leading bit of the fraction is implicitly encoded.




Single: N=32 E=8 Bias = 127 M=23
Double: N=68 E=11 Bias = 1023 M=52

Fig. 1: Floating point Number Format

**Problems with Floating point format:**

1. Wasted Bit Patterns - There are approximately eight million ways to represent 32-bit IEEE floating point. While NaN (Not-A-Number) has two quadrillion, 64-bit floating point has approximately two quadrillion. 2.251 $\times 10^{15}$ to be more precise. A NaN is an exception value used to represent undefined or invalid results, such as the result of a division by zero.
2. Doesn't obey common properties of numbers - The format specifies two zeroes, a negative zero and a positive zero, both of which behave differently. - Associative and distributive law loss as a result of rounding after each operation. Because of the loss of associative and distributive arithmetic behavior, concurrent programmes that use IEEE floating point produce irreproducible results. This is especially troublesome for embedded and control applications.
3. Overflows to ± inf and underflows to 0 - Overflowing to ± inf increases the relative error by an infinite factor, while underflowing to 0 loses sign information.
4. Complex Circuitry - Denormalized floating point numbers contain a hidden bit of 0 rather than 1. This results in a slew of new handling requirements, complicating compliant hardware implementations.

## II. POSIT NUMBERS

Posit numbers were introduced by John L. Gustafson in 2017 "as a direct drop-in replacement for IEEE 754 standard for floating-point numbers" [1]. Like floats, posits round off an answer if it is not exact. Hence, they do not require interval arithmetic or variable size operands. Yet they provide undeniable advantages over floats, including but not limited to: greater dynamic range, higher accuracy, better closure, cross-system identical results for bitwise operations, and easier exception handling. Posits never overflow to infinity or underflow to zero; rather, the greatest and least possible representations are used for the same. "Not-a-Number" (NaN) is used to indicate an action instead of a bit pattern. Simpler hardware design is possible due to lesser instances of special reserved values, compared to floating point representation. Hence, a posit processing unit takes less circuitry than an IEEE float FPU. With lower power use and smaller silicon footprint, the posit operations per second (POPS) supported by a chip can be significantly higher than the FLOPS using similar hardware resources. GPU accelerators and Deep Learning processors can do more per watt and per dollar with posits and still deliver superior answer quality. A posit format is defined as a tuple <n, es>, where n is the total bit-width and es is the maximum number of bits reserved for the exponent field.

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) was introduced in 1985 and since then, it witnessed mass adoption and the vast majority of modern computers have a floating point coprocessor. Even thought floating point numbers have its own flaws, they survived this long because of lack of a viable alternative to them. Cost of hardware implementation and energy consumption issues of floating point numbers makes them unsuitable or inefficient for IoT and machine learning tasks. Posit numbers were introduced to solve many of the problems with floating point numbers. For mainstream adoption of posit numbers, efficient hardware implementations and more studies about viability of using posits as a drop-in replacement for floating point numbers are needed.

## III. RELATED WORKS

The paper [1] introduced the concept of posit numbers in 2017. Posit numbers borrow many concepts from its predecessor, unums or universal numbers introduced in 2015. The paper also discusses about application of posit numbers in deep neural networks and presents studies about training deep neural network on 8 bit posit numbers. The results are comparable to results obtained through 16-bit floating point numbers. This work compares floating point and posit numbers in depth with many useful visualizations that show advantages of posit numbers over floating point. For same bit-width, posit numbers are shown to have larger dynamic range and precision around zero. Posit numbers are also shown to give exact answers for a larger range or numbers for addition.
The work in [2] is a highly influential paper from 1991 that tried to make inner workings of floating-point numbers known to everyone. This paper presents a tutorial on the aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating point standard, and concludes with examples of how computer system builders can better support floating point.

The work in [3] proposes several hardware implementations for posit arithmetic units and contain they include Posit adder/subtractor, multiplier, divider, and square root. Implementation was done on a Xilinx Virtex-7 FPGA VC709 platform. To reduce circuit area, instead of using a Leading Zero Detector (LZD) and a Leading Ones Detector (LOD), the authors choose to implement only a Leading Zeros Detector and argues that implementing both LZD and LOD is suboptimal. The authors also use a modular architecture which includes a

encoder, calculator, and a decoder module. They conclude paper remarking that the LZD design could have been better.

In work [4], the authors present an architecture of a parameterized Posit Arithmetic Unit generator that can generate adders and multipliers of any bit-width pre-synthesis. They synthesize generated arithmetic units using the parameterized generator for 8-bit, 16-bit, and 32-bit adders and multipliers and compare them with IEEE 754-2008 compliant adders and multipliers. In their comparison of m-bit units with n-bit IEEE 754-2008 compliant units, it is observed that the area and energy of an adder and multiplier are comparable to their IEEE 754-2008 compliant counterparts where m = n.

The paper [5] is about how the Single-Precision Floating Point ("F") extension of RISC-V can be leveraged to support posit arithmetic. They also present the implementation details of a parameterized and feature-complete posit Floating Point Unit. The posit FPU has been integrated with the RISC-V compliant SHAKTI C-class core as an execution unit. To enable the compilation and execution of C programs on PERI, they have also made minimal modifications to the GNU C Compiler (GCC), targeting the "F" extension of the RISC-V.

Authors in [6] enable application-level evaluations of the posit system that include performance and resource consumption. To this purpose, this work introduces an open-source hardware implementation of the posit number system, in the form of a C++ templatized library compatible with Vivado HLS. This library currently implements addition, subtraction and multiplication for custom-size posits. In addition, the posit standard also mandates the presence of the "quire", a large accumulator able to perform exact sums of products. The proposed library includes the first open-source parameterized hardware quire. However, this work concluded that the 32 bit posit adders and multipliers are much larger and slower than the corresponding floating point operators.

## IV. IMPLEMENTATION

We have written the RTL of posit adder using Verilog HDL. After creating the hardware description, its correctness is verified and the process is known as behavioral simulation. The design is converted into netlist which lists logic elements needed to implement the design in hardware. FPGA (Field Programmable Gate Array) synthesis is performed by dedicated synthesis tools. Functional simulation is done after the synthesis stage. The netlist or output of the synthesis of your design will be determined and consists of three steps: translate, map, and place and route. The netlist file from synthesis stage does not specify the physical layout of the logic elements in the final design. Goal of the implementation stage is to take the netlist and implement it to the target FPGA device. Timing simulation is done after the implementation stage, and it check for possible timing violations like setup/hold violations. Finally, the design is converted into a file format that is understood by the target FPGA known as the bitstream file. After generation of the bitstream file, it is then transferred into the target FPGA usually using a JTAG connection.

We implemented the posit adder using Xilinx Vivado HLS. The schematic of the design has been shown in Figure 2.

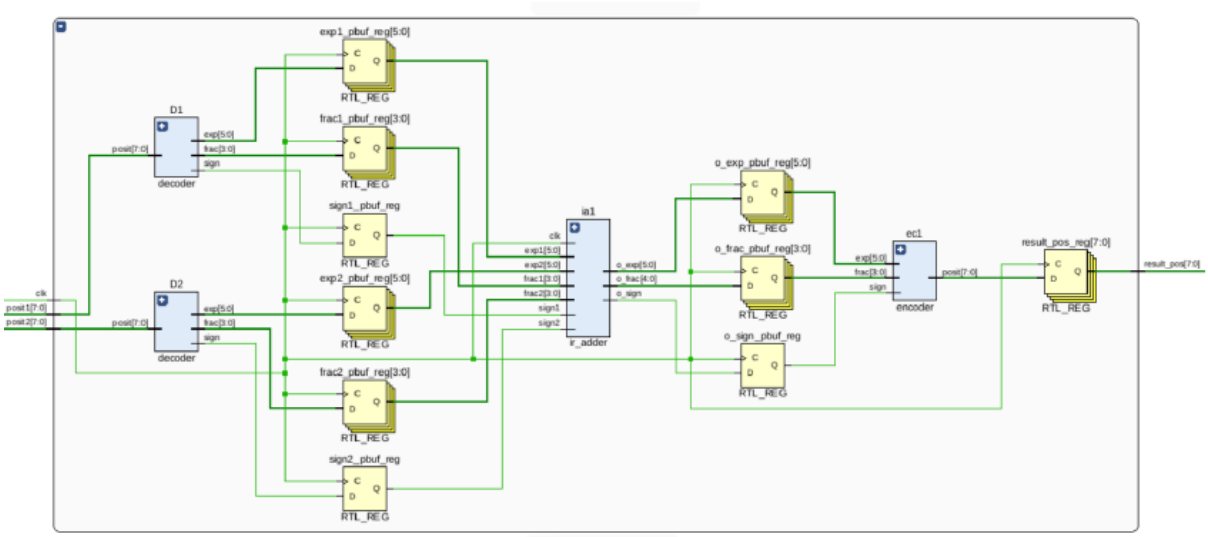



**Fig. 2: Elaborated Design of POSIT Adder Exported from Xilinx Vivado**

## V. CONCLUSION

Through this work, we saw why posit numbers are a promising new replacement for IEEE floating point numbers. We also explored literature on implementing posit arithmetic units. From literature review, We found that only a few works have been carried out in implementing posit arithmetic unit since posits are a relatively new concept. With this work, we will be contributing to fill this gap and guide future researchers looking for a viable replacement for floating point numbers. We have designed and implemented Posit arithmetic unit and a custom verification library in python. We have tested the arithmetic units completely and made sure that the design is correct and conforming to the posit standard. The implementation results and performance show that posit numbers are indeed a viable replacement for the IEEE-754 floating point numbers, and we can expect more works in this area in coming years and more hardware designs using posit numbers instead of traditional floating-point numbers.

## REFERENCES

[1]. John L. Gustafson and Isaac T. Yonemoto. "Beating Floating Point at its Own Game: Posit Arithmetic". In: Supercomputing Frontiers and Innovations 4.2 (Apr. 2017),pp. 71–86. doi: 10 . 14529 / jsfi170206. url: https : / / superfri . org / index . php/superfri/article/view/137.

[2]. David Goldberg. "What Every Computer Scientist Should Know about Floating-Point Arithmetic". In: ACM Comput. Surv. 23.1 (Mar. 1991), pp. 5–48. issn: 0360-0300. doi:10.1145/103162.103163. url: https://doi.org/10.1145/103162.103163.

[3]. Feibao Xiao et al. "Posit Arithmetic Hardware Implementations with The Minimum Cost Divider and SquareRoot". In: Electronics 9.10 (2020). issn: 2079-9292. doi: 10.3390/electronics9101622. url: https://www.mdpi.com/2079-9292/9/10/1622.

[4]. Rohit Chaurasiya et al. "Parameterized Posit Arithmetic Hardware Generator". In: 2018 IEEE 36th International Conference on Computer Design (ICCD). 2018, pp. 334–341. doi: 10.1109/ICCD.2018.00057.

[5]. Sugandha Tiwari et al. "PERI: A Configurable Posit Enabled RISC-V Core". In: ACM Trans. Archit. Code Optim. 18.3 (Apr. 2021). issn: 1544-3566. doi: 10.1145/3446210. url: https://doi.org/10.1145/3446210.

[6]. Yohann Uguen, Luc Forget, and Florent 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. doi: 10.1109/FPL.2019.00026.

Kala S, et. al. "A Study of POSIT Arithmetic Implementations." *International Journal of Computational Engineering Research (IJCER*), vol. 12, no.3, 2022, pp 09-12.