Received: 31 May 2021 | Revised: 4 August 2021 | Accepted: 12 August 2021

DOI: 10.1002/spe.3022

SURVEY PAPER

WILEY

Posits and the state of numerical representations in the age of exascale and edge computing

Alexandra Poulos | **Sally A. McKee** | **Jon C. Calhoun**

Holcombe Department of Electrical and Computer Engineering, Clemson University, Clemson, South Carolina, USA

Correspondence
Alexandra Poulos, 433 Calhoun Dr Riggs Hall Room 221-C, Clemson, SC 29634, USA.
Email: alpoulo@clemson.edu

Funding information
National Science Foundation, Grant/Award Numbers: SHF-1910197, SHF-1943114

Abstract

Growing constraints on memory utilization, power consumption, and I/O throughput have increasingly become limiting factors to the advancement of high performance computing (HPC) and edge computing applications. IEEE-754 floating-point types have been the de facto standard for floating-point number systems for decades, but the drawbacks of this numerical representation leave much to be desired. Alternative representations are gaining traction, both in HPC and machine learning environments. Posits have recently been proposed as a drop-in replacement for the IEEE-754 floating-point representation. We survey the state-of-the-art and state-of-the-practice in the development and use of posits in edge computing and HPC. The current literature supports posits as a promising alternative to traditional floating-point systems, both as a stand-alone replacement and in a mixed-precision environment. Development and standardization of the posit type is ongoing, and much research remains to explore the application of posits in different domains, how to best implement them in hardware, and where they fit with other numerical representations.

KEYWORDS

floating-point representations, heterogeneous computing, mixed-precision computing, posits, reconfigurable computing

1 | INTRODUCTION

The landscape of high performance computing (HPC) and edge computing is changing: as HPC systems grow to exascale, the size and scope of applications designed to run on these systems increase in tandem. I/O demand is increasing, as data transfer rates are unable to match processor speeds.[1] Consequently, the I/O bottleneck is becoming an increasing barrier to performance gains. Similarly, Internet of Things (IoT) devices have become ubiquitous.[2] The increasing complexity of the software deployed on edge devices is pushing the boundaries of power and memory-bandwidth capabilities.

The IEEE-754 floating-point standard is the de facto standard for implementing floating-point number systems.[3] However, this data representation is not always the most appropriate numerical type, and it frequently affords little flexibility to domain scientists in overcoming hurdles such as the I/O bottleneck and memory and power constraints. Rather, it is part of the problem. Heterogeneous systems, which contain more than one type of processor, offer programmers the opportunity to shift computationally intense operations off of the CPU and on to more specialized devices such as GPUs or FPGAs. Utilizing coprocessors allows for the design of systems that software can more fully exploit, which, in turn, enables developers to increase performance, but this exacerbates the problems posed by the I/O bottleneck.

As scientists are given the opportunity to tailor their hardware to suit their application needs, some have begun to explore the use of alternative numerical representations in the interest of increasing precision while maximizing throughput and minimizing power consumption and application run times.

Alternatives to traditional floating-point types are currently used in domain-specific applications; however, for the most part, they are derivative of IEEE-754 floating-point and share many of the same drawbacks, despite leaving a smaller memory footprint and decreasing power. They may not offer the dynamic range, precision, or numerical stability required for the task at hand. Recently, posits have been proposed as a drop-in replacement for IEEE-754 floating-point types. This numerical representation offers a greater dynamic range and precision than IEEE floats without many of the downsides. There has been increasing interest in exploring applications of this new type, and results from early research have been promising in demonstrating the superiority of posits over traditional floating-point representations. Nevertheless, there is still much to explore in order to fully exploit the advantages of this unique numeric type, including numerical analysis, application studies, and fully realizing the posit number system in hardware. This article provides a summary of the current state of research pertaining to posits and other numerical representations, and how these nonstandard types are being leveraged to overcome challenges the computing world now faces.

2 | FIXED AND FLOATING-POINT NUMBER SYSTEMS

We first provide background on fixed- and floating-point numerical representations before elaborating on the details of the IEEE-754 Floating Point Standard.

2.1 | Fixed-point

In a fixed-point number system the decimal point does not move. A number has a fixed number of digits reserved for the integer portion and a fixed number of digits reserved for the fraction portion. Typically a number is stored as an integer, and then the decimal point is shifted by some implicit scaling factor.[4]

A fixed-point number system $\mathbb{F}$ is given by $\langle n, d\rangle$, where n is the number of digits used to represent a number, and d denotes the offset of the decimal point starting from the least significant digit. For example, in a binary fixed-point number system with $n = 5$, and $d = 2$, the value 10101_2 is interpreted as $101.01_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} = 5.25_{10}$.

Fixed-point numbers are similar to integer representations. In a binary fixed-point system, values can be represented internally as integers, and the scaling can be implemented as fast and inexpensive bit shifts. This makes fixed-point number systems appealing where an executing processor might not have a floating-point unit (FPU), such as in embedded designs. Note that in fixed-point number systems, the difference between two consecutive representations is the same for all numbers in the system, that is, fixed-point numbers are uniformly distributed throughout their range.

The drawback of the fixed-point type is that only a limited number of values can be represented, and the dynamic range is relatively narrow when compared with floating-point representations. There is always a trade-off between the precision with which a value can be represented and the overall range of values that can be included in the numerical system. For example, in the unsigned $\langle 5, 2\rangle$ system from the previous example, the integer component has a maximum value of 7_{10}, and the fraction is only precise to a quantum of 0.25_{10}. The precision can be increased to 0.125 by using a $\langle 5, 3\rangle$ fixed-point system at the cost of the integer component dropping to a maximum value of 3.

2.2 | Floating-point

In contrast to fixed-point representations, floating-point number systems allow the decimal point to move, or *float*, to allow for a greater dynamic range. This requires taking the appropriate elements of the underlying base β, dividing them by some power of β, summing all of those values, then multiplying the summand by β raised to some power. This shifts the decimal point in the correct direction to obtain the desired value.

A floating-point number system $\mathbb{F}$ is given by a quadruple of integers $\{\beta, p, [L, U]\}$, where β is the base, p is the precision (or the number of significant digits of base β), and $L$and U define the exponent range.[5] A value in such a system has the form

FIGURE 1 Floating-point system given by $\{2, 3, [-1, 1]\}^5$

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{p-1}}{\beta^{p-1}} \right) \beta^E,$$

where d_i is an integer such that

$$0 \leq d_i \leq \beta - 1, \quad i = 0, \ldots, p - 1,$$

and $E$is an integer such that

$$L \leq E \leq U.$$

Some floating-point systems are normalized. With $\beta = 2$ this means that there is always an implicit 1 to the left of the decimal point. Normalization guarantees that the representation of every number in the system is unique[6] and avoids wasting bit patterns on leading zeros. This also saves space in hardware implementations, since the (implied) 1 is always assumed to be there.

While floating-point systems offer a greater dynamic range than fixed-point, this is at the expense of the precision with which a number can be represented. As with fixed-point numbers, there is a finite number of digits used to represent a value. More digits allocated to the mantissa necessitates fewer being used for the exponent, resulting in a smaller dynamic range; more digits in the exponent field results in less precision but wider dynamic range.

Unlike fixed-point representations, floating-point numbers are not uniformly distributed throughout their range. Instead, they are equally spaced only between successive powers of β.[5] This property can be seen in Figure 1, which represents a normalized floating-point system with $\beta = 2, p = 3$, and $[L, U] = [-1, 1]$.

3 | IEEE-754 FLOATING-POINT STANDARD

There were no standards for implementing floating-point number systems in hardware prior to 1985. Instead, each vendor used its own system based on the importance of dynamic range versus precision[7] for their intended applications. Arithmetic rounding operations and exception conditions (such as dividing by zero) were resolved depending on how the manufacturer saw fit.[8] This led to compatibility and reproducibility issues when porting programs across systems, creating a burden for programmers, who had to manage the complexities of different floating-point number systems.

The IEEE-754 floating-point standard was developed in 1985, which standardized how floating-point numbers are implemented. The standard precisely defines floating-point formats and describes valid operations on floating-point values. It sets minimum requirements for floating-point implementations and for portability and reproducibility between systems, including specifying valid rounding modes, specifying how exceptions should be handled, and mandating that the results of certain mathematical operations be bit-exact across all systems.[3] The standard also provides recommendations for other number formats, operations, and implementations. Today, IEEE-754 is the de facto standard for the implementation of floating-point number systems in hardware.

3.1 | Properties of IEEE-754 floating-point number systems

An IEEE-754 floating-point number has three fields: a sign bit s, an unsigned integer exponent e, and a fraction field also known as the *mantissa* or *significand*. There is an implicit *bias*in the exponent value which is determined by the fixed number of exponent bits E as $2^{E-1} - 1$. The IEEE-754 floating-point number system is normalized. An IEEE-754-compliant floating-point number can be numerically interpreted as

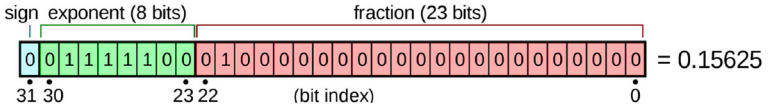


FIGURE 2 Structure of an IEEE-754 single-precision float[9] [Colour figure can be viewed at wileyonlinelibrary.com]

$$(-1)^s \times 2^{(\text{exponent} - \text{bias})} \times \left(1 + \sum_{i=1}^{M} b_{M-i} 2^{-i}\right),$$

where $b \in [0, 1]$ and $M$is the number of bits in the mantissa. Figure 2 shows an example of a single-precision IEEE-754 float.

3.2 | Different IEEE-754 types

IEEE-754 binary, floating-point types come in multiple sizes, which are summarized in Figure 3. The dynamic range and number of decimal digits with which a number can be accurately represented increases as the size of the IEEE floating-point type increases. As the number of bits used to represent a floating-point increases, the standard allocates significantly more bits for the significand field than it does for the exponent field, realizing the trade-off of greater dynamic range for more precision.

The most common floating-point types are binary32 and binary64, commonly called single- and double-precision, respectively. A binary32 float typically offers the dynamic range and precision desired for most calculations. A binary64 is used when the precision or range exceeds what binary32 offers, such as in scientific computing applications. If a greater dynamic range or precision is required, the width of an IEEE floating-point type can be arbitrarily increased to meet the needs of the programmer. If less precision or dynamic range suffices, or if maximizing throughput and/or minimizing power is important,[10] smaller types (such as binary16) may be used.

In some instances it is pragmatic to use multiple floating-point types within a program. For example, consider a subroutine that accumulates binary32 values over several iterations. If the difference in magnitude between two values is sufficiently large, the smaller value will be shifted down to zero during alignment, preventing it from being accumulated. One solution would be to store all values in binary64 format, but this increases memory usage and power consumption. A better solution is to store only the accumulated value in binary64 format, which can ensure that shifting does not cause this cancelation. Such *mixed-precision computing* approaches are commonly used not just to avoid issues of cancelation, but also to reduce application memory footprint, run time, and power consumption.[11,12]

3.3 | Limitations of IEEE-754

The shortcomings of the IEEE-754 standard are well documented.[13] While the standard resolved many issues with early floating-point system implementations, many issues remain. For example, identical results across all systems are still not

Parameter	binary16	binary32	binary64	binary128	binary{k} ($k \geq 128$)
k, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$k - \text{round}(4 \times \log_2(k)) + 13$
emax, maximum exponent e	15	127	1023	16383	$2^{(k-p-1)} - 1$
Encoding parameters					
bias, $E - e$	15	127	1023	16383	*emax*
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	$\text{round}(4 \times \log_2(k)) - 13$
t, trailing significand field width in bits	10	23	52	112	$k - w - 1$
k, storage width in bits	16	32	64	128	$1 + w + t$

FIGURE 3 IEEE-754 binary interchange format parameters[3]

guaranteed. This is an artifact of the standard allowing for different rounding modes. IEEE mandates that only basic math functions are rounded in such a way as to be bit-accurate across systems. The rounding of many math library functions are implementation defined[3] and thus can differ from system to system.

Another limitation is redundant representations. The IEEE floating-point system has positive zero and negative zero, which makes no sense mathematically. Additionally, any bit pattern with all exponent bits set to 1 is used to represent an invalid NaN (not a number) value. This design decision allows encoded information in the returned NaN value to assist the programmer in debugging, but it is often the case that the programmer either does not use this information or has no access to it. This wastes bit patterns that could otherwise be used to represent valid numerical values. Further, subnormal and exceptional values are handled at the hardware level, which burdens floating-point operations.[14]

4 | MIXED PRECISION ARITHMETIC AND OTHER FLOATING-POINT TYPES

The end of Moore's law and Dennard scaling has led to a paradigm shift in computing.[15,16] Previously, the desire for maximum precision caused domain scientists to use larger floating-point types to ensure correct results (with little regard for the overhead incurred). HPC workloads are generating increasingly large volumes of data, creating an I/O bottleneck that must now be addressed.[17] The advent of the Internet of Things (IoT) has led to increasing deployment of more complex machine learning (ML) models on smaller architectures, and thus ML applications and frameworks have hit a wall due to constraints on edge-device model size and the power required to run those embedded models. The end result is increased application run time and power utilization that is not sustainable in the age of exascale and edge computing.

While not always suitable for HPC applications, binary16 is often used to reduce throughput and training times in ML applications, even though this data type was not designed with deep learning applications in mind.[18] Studies have repeatedly shown that neural networks are far more sensitive to the size of the exponent than to the size of the mantissa,[19] that is, dynamic range is more important than precision. With this information in mind, several alternatives to IEEE floating-point representations have been designed to sacrifice precision for greater dynamic range. Two such examples are Google's bfloat16 and NVIDIA's TensorFloat-32. Figure 4 shows a comparison of these representations with traditional IEEE floating-point types.

4.1 | Bfloat16

Brain floating-point, or bfloat16, is a custom 16-bit floating-point format developed by Google Brain, Google's artificial intelligence research group.[19] A bfloat16 is a truncated version of binary32 and is comprised of one sign bit, eight exponent bits, and seven mantissa bits. By maintaining the same number of bits in the exponent field, bfloat16 has a dynamic

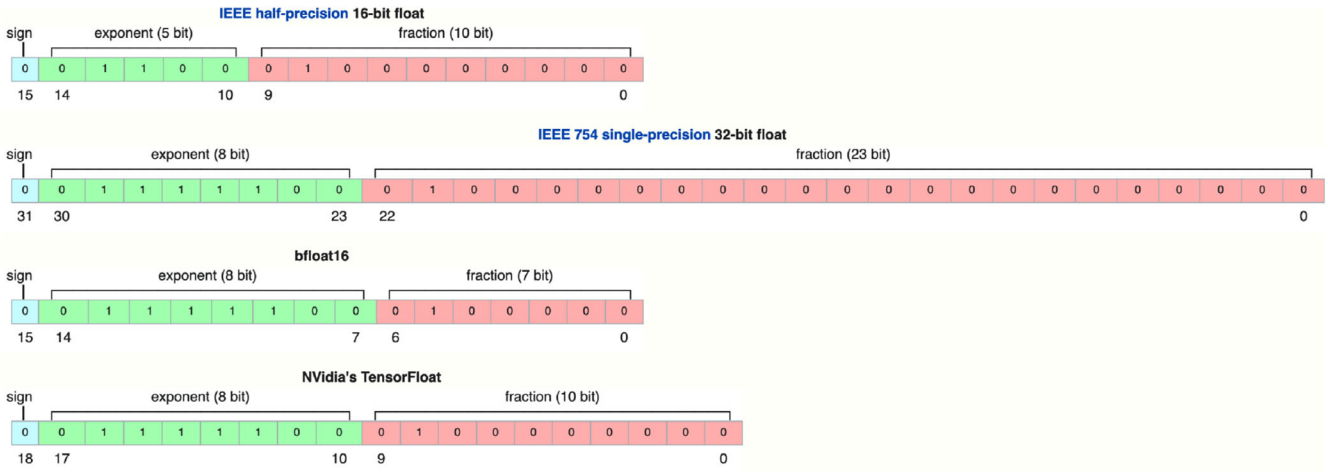


FIGURE 4 A comparison of IEEE types with bfloat16 and TensorFloat-32[20] [Colour figure can be viewed at wileyonlinelibrary.com]

range identical to that of binary32, but a lower precision than both binary32 and binary16. Keeping the same number of exponent bits ensures that the handling of underflow, overflow, and NaNs is the same as in the IEEE standard. This means that bfloat16 is subject to many of the drawbacks discussed in Section 3.3. Bfloat16 handles denormalized numbers differently from IEEE types: they are flushed to zero rather than through being reduced by gradual underflow.

Bfloat16s accelerate high-performance ML on Google's Cloud Tensor Processing Units (TPU). Cloud TPUs use this type within systolic array architectures, where each multiply-accumulate operation in a matrix multiplication uses bfloat16 for the multiply and binary32 for the accumulate. Another benefit of using bfloat16 is that binary16, due to its narrow dynamic range, typically requires special techniques (such as loss scaling) to preserve small gradient values,[21] whereas bfloat16 performs almost as a drop-in replacement for binary32 when training and running deep neural networks[19] (while halving the memory footprint).

The TPU software stack provides automatic conversion for mixed-precision training, allowing values to be seamlessly converted between bfloat16 and binary32. Automatic conversion allows model values to be stored in full 32-bit format, which eliminates concerns regarding portability across hardware platforms. In addition to TPUs, bfloat16 is supported on Intel Xeon processors, Intel FPGAs, and TensorFlow, making it an attractive data type for mixed-precision computing.

4.2 | TensorFloat-32

TensorFloat-32 (TF32) is NVIDIA's custom floating-point type designed to accelerate AI and HPC workloads.[22] A TF32 is comprised of one sign bit, eight exponent bits, and 10 mantissa bits. This data type has the same dynamic range as binary32 and the same precision as binary16. TF32 thus offers benefits similar to bfloat16 but with increased precision.

TF32 can be used on Tensor Core GPUs based on the Ampere architecture,[23] such as the A100 GPU. TF32 cores operate on binary32 inputs and produce binary32 results. By default, they use the 19-bit TF32 type for intermediary matrix calculations, so the user benefits from this data type with no code changes. Tensor cores also further integrate mixed precision with other types via enhanced math capabilities that support binary16 and bfloat16. Additionally, automatic mixed precision can be enabled to seamlessly switch between binary32 and binary16. This improves throughput and performance,[24] but loss-scaling is required to maintain AI model fidelity due to the issues with the dynamic range of binary16.

5 | POSITS

A posit number system $\mathbb{P}$ is given by $\langle n, es\rangle$, where n is the bit-width of a posit and $es \geq 0$ is an upper bound on the size of the exponent field. Figure 5 shows the internal structure of a posit, which consists of a sign bit s, one or more (up to $n-1$) regime bits, an optional unsigned integer exponent e, and an optional mantissa.

The sign bit and the optional fraction bits are interpreted as in IEEE-754. If the sign is negative, then the two's complement of the remaining $n-1$ bits is taken prior to decoding. The regime field uses run-length encoding consisting of a run of identical bits terminated by an occurrence of the opposite bit. To decode the regime,[25] let m be the number of identical bits in the regime. If the first bit in the regime is zero, the number of zeros represents a negative value $k = -m$. Otherwise the number of consecutive bits represents the positive value $k = m - 1$. The regime indicates a scale factor of $useed^k$, where $useed = 2^{2^{es}}$.

The posit number system is normalized. A posit number can be numerically interpreted as:

$$(-1)^s \times useed^k \times 2^e \times \left(1 + \sum_{i=1}^{M} b_{M-i} 2^{-i}\right).$$

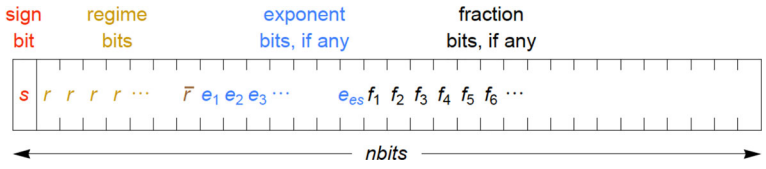


FIGURE 5 Structure of a posit[25] [Colour figure can be viewed at wileyonlinelibrary.com]

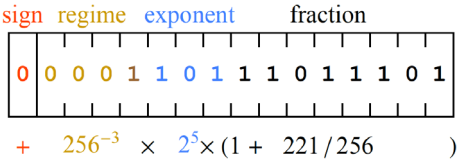
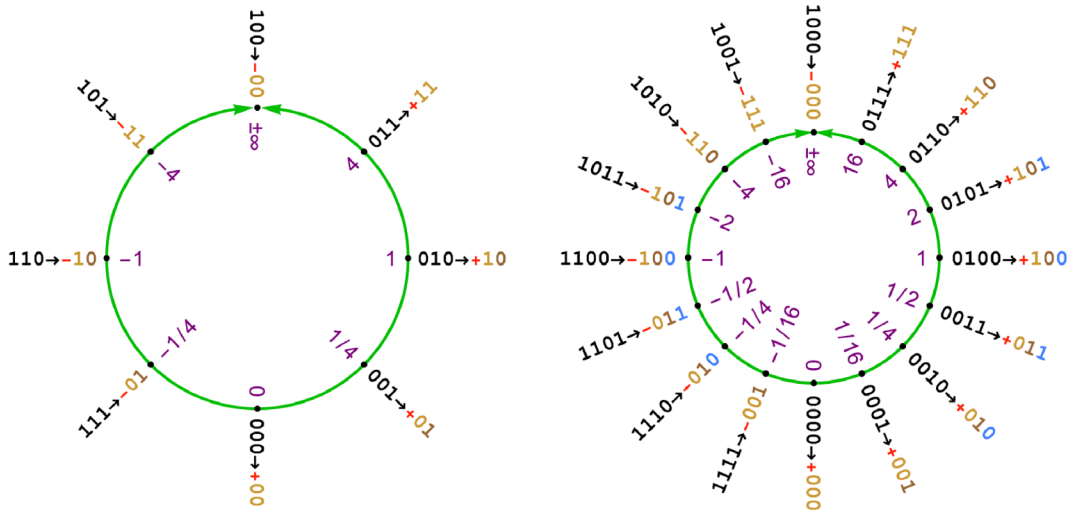


FIGURE 6 A ⟨16, 3⟩ posit representing $(3.55393 \times 10^{-6})_{10}{}^{25}$ [Colour figure can be viewed at wileyonlinelibrary.com]



(A) Posit ⟨3, 1⟩ (B) Posit ⟨4, 1⟩

FIGURE 7 Real projective line of the posit format.[26] The color-coding corresponds to the fields as seen in Figure 5 [Colour figure can be viewed at wileyonlinelibrary.com]

If there are no fraction bits, the equation simplifies* to $(-1)^s \times useed^k \times 2^e$. A posit is interpreted as above with the exception of two reserved bit patterns: $\vec{0} = 0$ and $-2^{n-1} = \pm\infty$. The smallest positive posit value, *minpos*, is given by $useed^{2-n}$. The largest positive posit value, *maxpos*, is given by $\frac{1}{minpos}$, or $useed^{n-2}$. Every posit number is an integer multiple of *minpos*. Figure 6 shows an example of a ⟨16, 3⟩ posit.

5.1 | Properties of posits

Posit number systems have only one value for 0. Additionally, $+\infty$ and $-\infty$ share the same bit pattern. The rationale behind this design decision is that a calculation should always return a real-valued number. With traditional floats, expressions evaluated with a $\pm\infty$ term either return $\pm\infty$ or NaN, which cannot be used in further evaluations.[13] A programmer needing to compute with infinity should instead use a symbolic numerical system that can handle those calculations correctly.[27] For these reasons, the $\pm\infty$ value is also referred to as Not a Real (NaR) and is the only exception value in a posit number system. There are no subnormal values in the posit format.

Traditional floating-point number systems are based on the real number line. In contrast, posits are based on the projective real number line. Figure 7 provides a visual representation of two posit number systems, ⟨3, 1⟩ and ⟨4, 1⟩, on the projective real number line. When *es* is fixed and *n* is increased, the bit patterns for existing values do not change—they simply have appended zeros. Conversely, appending a one to a posit creates a new value in the system. For example, in Figure 7A the binary value of 1 is 010 and in Figure7B it is 0100. This makes it very easy to switch between posit systems of different sizes.

As with signed integers, obtaining the negation of a value in a posit number system requires taking the two's complement to reflect about the vertical axis of the posit ring. Obtaining a reciprocal requires reflecting about the horizontal axis

*The equation simplifies similarly if there are no exponent bits.

TABLE 1 Comparison between various floating-point representations

Number format	Pros	Cons	Use cases
binary64	High degree of precision and a large dynamic range	Large memory footprint and high bandwidth penalty	HPC and scientific computing applications where precision is of high importance
binary32	Lower memory footprint and bandwidth consumption than binary64. Fully supported on most GPUs	Insufficient precision for many scientific applications. Memory and bandwidth constraints are still a problem	ML training and inference. HPC and scientific computing applications where precision is not a concern
binary16	Low memory overhead and high throughput offer significantly reduced execution times. Supported on most modern GPUs	Lower precision and less dynamic range than other IEEE types. Less suitable for ML than other representations	ML training and inference. Also used for post-training quantization for faster inference
bfloat16	Dynamic range of binary32. Highly portable across systems. Supported by several deep learning accelerators	Lower precision than binary32, binary16, and TF32	ML training and inference. May be able to be leveraged for higher-precision computations[89,90]
TF32	Dynamic range of binary32 and precision of binary16. Ability to use binary16 and bfloat16 alongside TF32 is built-in	Larger memory footprint than binary16 and bfloat16. Currently limited to GPU implementations	AI training. HPC applications as an accelerator for linear solvers[23]
Posits	Customize precision, dynamic range, and data size for a given task	Limited hardware support	ML training and inference. HPC and scientific computing applications

by taking the two's complement of the posit and excluding the sign bit. For powers of two, posits can always represent perfect reciprocals. For nonpowers of two, the reciprocal of a value x ranges from $\frac{1}{x}$ to $\frac{1.125}{x}$ in all cases. These properties hold for any $\langle n, es\rangle$ posit configuration.[27] A comparison of posits and the previously discussed numerical representations is available in Table 1.

5.2 | The posit standard and the quire

The posit standard[28] is still in draft form, and thus there are no finalized requirements for implementing posit number systems. As of this writing, an implementation is compliant with the standard if it supports full functionality of at least one precision (suggested n of 8, 16, or 32) and the *quire*, a fixed-point format storing exact sums and differences of products of posits. A posit-compliant system may be realized in software, hardware, or a combination thereof. If an implementation supports more than one posit type then it must support conversions among them. The posit standard defines a binary interchange format with $es = 2$, regardless of the size of n, for portability across systems. When $es = 2$, *minpos* is given by 2^{-4n+8} and *maxpos* is given by 2^{4n-8}.

The posit standard has one rounding mode: if a value x is exactly representable, then it remains unchanged; if $|x| > maxpos$, then $x = sign(x) \times maxpos$; if $0 < |x| < minpos$, then $x = sign(x) \times minpos$; otherwise, x is rounded to the nearest representable posit value. If two posits are equally near x, the posit with binary encoding ending in 0 is chosen (*round to nearest, tie to even*).

The quire is a fixed-point two's complement value which behaves like a Kulisch accumulator.[29] It can be thought of as a dedicated register that permits dot products, sums, and other operations to be performed with rounding error deferred to the very end of the calculation.[30] Using the quire to evaluate long sequences of arithmetic operations can allow for binary64 to be replaced by 16-bit posits in certain instances.[30]

For an n-bit posit system, the size of the quire is given by $16n$. If an implementation supports the posit binary interchange format, this size enables the product of two posits to be added or subtracted in the quire up to at least $2^{31} - 1$ times without rounding or overflow. The structure of the quire can be seen in Figure 8. Similar to a regular posit, if the sign bit is 1 and all other bits are 0 then the value in the quire evaluates to NaR. Otherwise the value in the quire

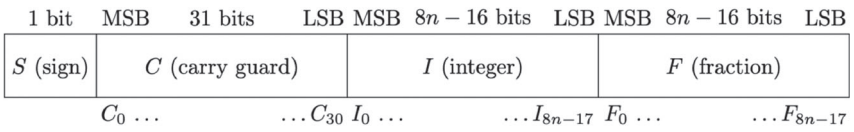


FIGURE 8 Binary quire format[28]

evaluates to 2^{16-8n} times the two's complement signed integer represented by all of the bits concatenated.[28] Expressions can be evaluated exactly in the quire and then rounded to posit format to create a fused expression. The quire may be used as a scratch space for fused operations, but this is not part of the draft posit standard.

5.3 | Posits in software

There are several implementations of posits in software. One of the more well-known versions is SoftPosit.[31] SoftPosit is written in C and supports $\langle 8, 0\rangle$, $\langle 16, 1\rangle$, and $\langle 32, 2\rangle$ posit types, as well as $\langle n, 2\rangle$ types for $n \in [2, 32]$. A python wrapper for SoftPosit is also available.[32] SoftPosit-Math[33] defines several mathematical functions to be used with the SoftPosit library.

bfp (beyond floating-point)[34] is a C/C++ posit library similar to SoftPosit but with less functionality. cppPosit[35] is a template-based C++ library inspired by bfp that supports posits with $n \in [4, 64]$ with any valid exponent size. This library provides several features not included in the bfp library. The Universal Numbers Library[36] is an open source header-only C++ template library for universal number arithmetic. The library implements several number systems for numerical analysis and computational mathematics, including a variety of posit configurations. PERI[37] recently presented a modification to the RISC-V GNU C compiler (GCC) to enable compilation of floating-point literals into $\langle 32, 2\rangle$ and $\langle 32, 3\rangle$ posits.

5.4 | Posits in hardware

Many efforts focus on realizing the posit number system in hardware. No exception handling exists at the hardware level, and because there is only one rounding mode, hardware implementations are expected to yield simpler designs and consume less power than floating-point implementations.[25] There is no standardized posit processing unit, so much of the research on hardware-level posits involve FPGA or ASIC implementations.

Traditional floating-point types have fixed sizes for exponent and mantissa, which simplifies hardware design. Unlike these types, the regime, exponent, and mantissa fields of a posit are determined at run time. This run-time variation poses a design challenge for hardware implementations. Jaiswal and So[38,39] present the first parameterized Verilog HDL implementation of posits for FPGAs. Their open-source design[40] includes a floating-point-to-posit converter, a posit-to-floating-point converter, and adder/subtractor and multiplier architectures. Jaiswal[41] adds support for a posit division generator using the Newton–Rhapson method, and he proposes a pipelined architecture for $\langle 32, 6\rangle$ posit addition/subtraction, multiplication, and division units.[42]

Chaurasiya et al.[43] note that FPUs have massive area and energy footprints, and they explore the feasibility of replacing them with posit arithmetic units (PAUs). They present a parameterized PAU generator that realizes adders and multipliers. Designs are synthesized for both FPGAs and ASICs for 8-, 16-, and 32-bit posits. The designs are then compared with IEEE-compliant adders and multipliers, and also with an earlier posit architecture proposed by Jaiswal and So.[38] Results show that the area and energy of the proposed PAU adder and multiplier are comparable to their same-width IEEE counterparts. The PAU architecture also outperforms previous posit implementations for smaller bit widths, but this design is not pipelined, and thus it results in low operating frequencies for large bit widths. The authors show that the superior numerical accuracy of posits will allow an n-bit IEEE adder and multiplier to be safely replaced by an m-bit PAU adder and multiplier, where $m < n$.

Xiao et al.[44] address the shortcomings of previous implementations in References 38,39,41,43,45 by minimizing the circuit area of a posit processing unit for embedded devices. They note that Jaiswal's work[38,39] utilizes both a leading ones detector and a leading zeros detector when encoding and decoding posit numbers, which requires redundant chip area. Chaurasiya[43] addresses this issue, but neither of his designs fully leverages the mathematical properties of the posit

number system in hardware (while trying to occupy minimal area). Both Jaiswal and Chaurasiya decode a posit to sign and magnitude representation to simplify logic, rather than leaving the value in two's complement form. Xiao et al.[44] propose new encoding/decoding modules that maintain the two's complement representation, and they present new algorithms for posit adder/subtractor, multiplier, divider, and square root operations. Rather than using the Newton–Rhapson method for division and square root as in Jaiswa's work,[41] they instead implement the alternating addition and subtraction method, which allows them to have the division and square root operations in the same module. Results show that their division and square root design yields area and power utilization similar to that of a posit adder. Additionally, their multiplier implementation shows significant reductions in maximum net delay compared with the designs of Jaiswal.[41] Both of these are noteworthy improvements over prior work, but the area reduction for the division and square root module comes at the cost of higher latency—it only produces one result bit every clock cycle, which is 3–5× slower than in Jaiswal's design.[41]

RISC-V[46] is a commercial-grade open-source ISA. It is highly extensible and customizable, which makes it a suitable platform for experimenting with posit implementations.[47] In PERI,[37,48] Tiwari et al. present the first posit-enabled RISC-V core which integrates the posit numeric type as a functional unit within a RISC-V processor. They also present a modification of the RISC-V GCC to support compilation of posits, bypassing the need to convert floating-point values to posits in hardware and enabling direct execution on the RISC-V core. The proposed posit FPU is incorporated into the RISC-V core as both a tightly coupled execution unit and a coprocessor. The design includes support for comparison, negation, absolute value, fused multiply-add (FMA), and square root for posits in addition to the previously mentioned operations from prior work; however, it does not support the quire. The PERI core also allows for *dynamic switching* between $\langle 32, 2\rangle$ and $\langle 32, 3\rangle$ posits at run-time with minimal overhead. Integrating the posit FPU as a tightly coupled execution unit reuses the existing floating-point infrastructure (including the floating-point register file) of the RISC-V architecture, which limits the system to only using either 32- or 64-bit posits. Additionally, this approach prevents posits and IEEE-754 floats from coexisting in the same core and disallows extending or modifying existing functionality for custom use. The authors present an alternative methodology for implementing the posit FPU as an accelerator using custom opcodes. This approach eliminates the need to use floating-point register file, allowing posits and floats to coexist in the same design and enabling further extension of the ISA to add more complex posit functionality.

Clarinet[49] is a RISC-V-based framework for posit arithmetic empiricism. The motivation behind this framework is to allow posit and floating-point types to coexist in experiments. The Clarinet framework includes support for FMA with quire functionality, the first of its kind for the RISC-V architecture. The reuse of the floating-point register file in PERI limited the core to implementing 32- or 64-bit posits and also precluded the use of both floats and posits simultaneously when not integrated as an accelerator. Clarinet addresses both of these issues by including a posit register file; this allows for the use of posits of any size and for posits to be used alongside floats in a mixed-precision environment. A major limitation of the Clarinet framework is that it can only execute applications that contain multiply, multiply-add, and multiply-accumulate operations on posits. There is no support for division or square root, and there are no dedicated instructions for posit addition, subtraction, and multiplication when they are not used with the quire. This limits application analysis unless these operations are implemented in software. Rather, the usage model focuses on utilizing the quire to accumulate values which have been converted from floating-point to posit, which are then converted back to floating-point. This mitigates the effect of rounding errors but limits the functionality of the framework.

6 | APPLICATIONS OF POSITS

For posits to gain broad adoption in a diverse set of areas, research must be undertaken to show how they perform in real-world applications in areas such as AI/ML and scientific computing. In this section, we survey how posits are being used in applications today.

6.1 | Machine learning

Gustafson[25] shows that $\langle 8, 0\rangle$ posits can be especially useful in approximating a sigmoid function. One example is the logistic sigmoid function $\frac{1}{e^{-x}+1}$, commonly used as an activation function in artificial neural networks (ANNs). This function is expensive with IEEE-style floats, requiring over 100 clock cycles to compute due to the division and library calls required to evaluate $\exp(x)$. With posits, this function can be quickly and cheaply approximated by

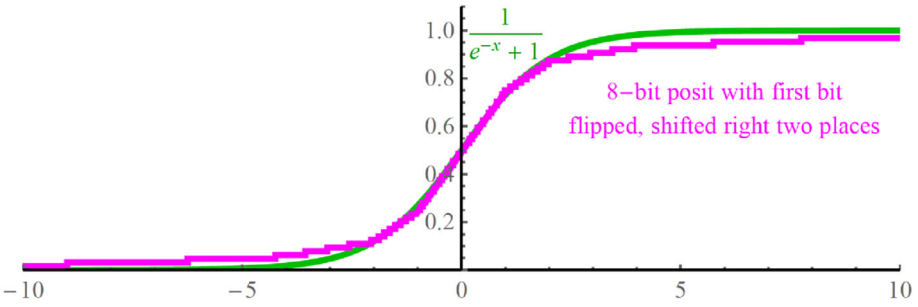


FIGURE 9 Fast sigmoid function using posit representation[25] [Colour figure can be viewed at wileyonlinelibrary.com]

flipping the first bit of the posit (representing x) and logically shifting twice to the right. Figure 9 illustrates the resulting posit function, which very closely resembles the actual sigmoid function and has the correct slope when intersecting the y-axis.

Building on Gustafson's work, Cococcioni et al.[50,51] explore using posits with $es = 0$. They present several fast, approximated versions of operations using only low-level bit string manipulation. These functions include hyperbolic tangent and extended linear unit, which are commonly used in deep learning. They use these new activation functions for inference, and they test them on the GTRSB and MNIST datasets. Results show that $\langle n, 0\rangle$ posit configurations yield slightly less accurate output when compared with binary32, but they perform better with respect to the mean sample inference time of the network. Since these operations need only integer arithmetic and logic operations, they can be executed straightforwardly in most ALUs, saving power consumption and increasing computational efficiency. Cococcioni et al. note that being able to write functions as a sequence of arithmetic-logic operations allows for posit operations to be vectorized via integer SIMD engines, avoiding the considerable resource demands of GPUs or floating-point SIMD processors.

Smaller data types are increasingly being used for ML, since deep learning models can tolerate lower numerical precision. Neural networks are more sensitive to exponent field size than that of the mantissa field, making posits an attractive data type: their size is user-defined, and they offer fine-grained control over the scaling factor for dynamic range (via the choice of es). This is important for operations that are bound by memory bandwidth, meaning on-chip memory bandwidth determines how much time is spent computing the output. Storing operands and outputs in smaller number formats reduces transferred data, thereby improving overall performance via increased throughput.

Fixed-point number systems are an ideal choice when working with memory-constrained embedded devices for performing deep learning inference,[52-55] but they are not well suited for deep learning applications, in general.[53] In recent literature, Langroudi et al. explore the use of posits in deep convolutional neural network inference compared with fixed-point representations.[56] Posits consistently outperform fixed-point number systems with fewer bits when evaluated on the MNIST, CIFAR-10, and ImageNet datasets. Results suggest that posits could be used with more complex datasets such as LeNet, ConvNet, and AlexNet with less than 1% accuracy loss while reducing memory footprints by up to 36% (compared with a fixed-point representation).

Autonomous driving exemplifies a limited-resource environment in which floating-point number systems are frequently used. Autonomous vehicles have many input sensors (e.g., cameras, radar, and lidar) that provide data to be clustered and classified in real time by embedded on-board devices.[57] Researchers working on autonomous driving applications[58] show that 16-bit posits achieve the same accuracy as binary32 in a k-NN classifier, and they further demonstrate that eight-bit posits outperform binary16 on the sift-128-euclidean, mnist-784-euclidean, and fashion-mnist-784-euclidean datasets.[59] The authors show that an LUT implementation of a posit processing unit for eight-bit posits requires only 64-KB storage, which satisfies requirements for memory-constrained devices.

Much research uses posits for the inference stage of deep learning, since performing this step in resource-constrained devices is challenging.[45,57,58,60-63] Newer research[64] explores the use of posits in both the training and inference phases of ML. Deep PeNSieve[26] is a deep learning framework designed to perform both training and inference for DNNs. Murillo et al.[26] use a $\langle 16, 1\rangle$ posit to obtain a 4% higher top 1% accuracy than binary32 on the CIFAR-10 dataset. Results for other datasets are within 1%. They demonstrate that posits converge similarly to binary32 while presenting lower error throughout training.

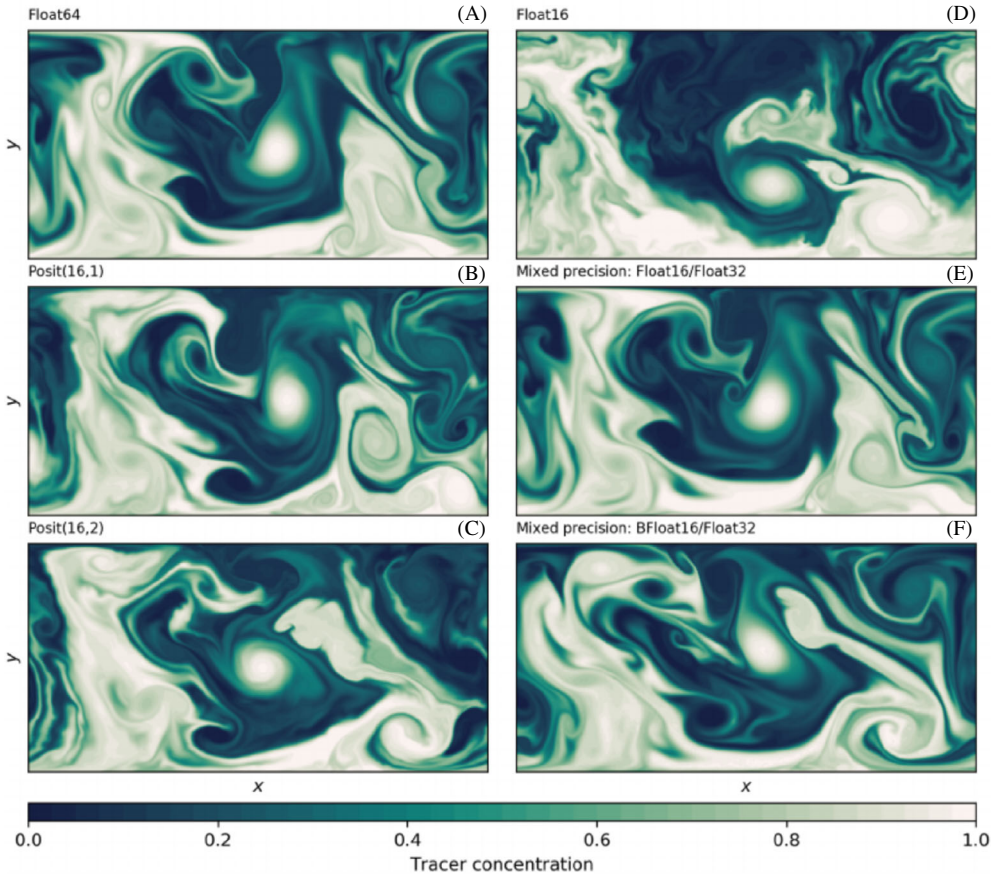


FIGURE 10 Comparison of shallow water simulations of vigorous turbulence interacting with a zonal current using different numerical representations[65] [Colour figure can be viewed at wileyonlinelibrary.com]

6.2 | Numerical simulation

Posits are gaining traction in other areas of HPC. One such field is numerical climate simulation. The work of Klöwer et al.[65,66] explores posits in weather and climate models. In their work, weather and climate models of low and medium complexity are simulated to compare posits not just to IEEE floating-point types, but also to a mixed-precision environment with binary16/binary32 and bfloat16/binary32. The target application uses Navier–Stokes equations to simulate vigorous turbulence interacting with a zonal current. The simulation is advanced several days, and Figure 10 shows that 16-bit posits very closely approximate the binary64 control while also demonstrating that the posit type reduces accumulated rounding errors compared with other types. The use of 16-bit posits as an alternative to double-precision floats reduces the memory footprint of the application by 75%.

7 | OPEN PROBLEMS AND FUTURE WORK

Although much work has been done to develop and explore the ideas of posits theoretically, in hardware, and in software, other important areas and use-cases of posits need to be addressed before they are ready for wide adoption. In this section, we detail some open questions about the use and applicability of posits.

7.1 | Limitations of current research

There remains much work in realizing the posit number system. One problem with the current state of affairs is that there is no complete posit math library,[67] so the capabilities and limitations[68] of the posit type have not yet been fully explored

and compared with different numerical representations. In particular, there is limited work comparing posit to non-IEEE types such as bfloat16 and TF32. Additionally, much effort has been dedicated to leveraging the structure of floating-point numerical systems to optimize algorithms and reduce error.[69] Posits do not share many of the properties of traditional floating-point types, such as having a bound on the amount of error introduced in each step of a computation.[70] Thus, posits must be considered from a numerical analysis perspective to develop new algorithms to mitigate error and optimize performance.

Hardware implementations of posits are continuously improving; however, many of the designs today are either incomplete, high-latency, or utilize too much area when synthesized. Furthermore, there is limited open source compiler support for posits which creates another hurdle when using this type. Much of the research pertaining to the applications of posits has been emulated on conventional CPUs, which limits additional analysis. While there have been efforts toward creating a toolchain to allow posits to be used side-by-side with other floating-point types, such an environment does not exist today.

7.2 | Future research directions

Outside of addressing the limitations discussed above, there are many aspects of posits that have not been addressed in the current literature, and they deserve attention.

The I/O bottleneck is becoming increasingly prohibitive in HPC applications. Embedded devices for edge computing are also facing challenges with the size of numerical representations due to memory constraints. Compression is one approach to solving this problem; however, floating-point numbers are notoriously hard to losslessly compress.[71] Error-bounded lossy compression offers much higher compression ratios for floating-point types at the expense of information loss, and is increasingly being utilized to address this problem.[1,71-74] From a data reduction standpoint, posits are ideal because they offer reduced size without necessarily decreasing accuracy or range. To date, no efforts have been undertaken to study the compressibility of posits, nor are there compression algorithms leveraging the posit type.

Another problem facing the computing community which has only recently begun to be addressed with posits[75] is resilience. As HPC systems continue to grow in size, so does the need to make them more resilient.[76,77] Errors can arise from many sources, such as hardware failures, voltage fluctuations, cosmic particles, and deliberate fault injection.[76,78] This is not something that a change of numerical representation can address, and so it is important to understand how transient errors impact applications using posits from a fault tolerance perspective. There have been many studies investigating the impact of soft errors on floating-point numbers,[79] how those errors can propagate through a simulation,[80] and how to make the systems utilizing those types and the algorithms that run on those systems more fault tolerant.[81-83] Additionally, studies have explored leveraging mathematical properties from coding theory[84-86] and other heuristic techniques[87,88] as a means to overcome multibit errors beyond the capabilities of error-correcting codes.

8 | CONCLUSION

Growing constraints on memory utilization, power consumption, and I/O throughput have increasingly become limiting factors with respect to the advancement of HPC and edge computing applications. IEEE-754 floating-point types, the de facto standard for floating-point number systems, have shown to be prohibitive when it comes to addressing these problems. Alternative numerical representations have recently been gaining traction for these reasons. Recently, posits have been proposed as a drop-in replacement for IEEE-754 floating-point representation. The development and standardization of the posit type is still ongoing; however, posits appear to be a promising alternative to traditional floating-point number systems, both as a stand-alone replacement and in a mixed-precision environment. There is still much research that needs to be done to explore the applications of posits, how they can be used in different domains, and where they fit in with other numerical representations in use today.

ACKNOWLEDGMENT

This material is based on work supported by the National Science Foundation under Grant No. SHF-1910197 and SHF-1943114.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Jon C. Calhoun https://orcid.org/0000-0001-7191-4422

REFERENCES

1. Cappello F, Di S, Li S, et al. Use cases of lossy compression for floating-point data in scientific data sets. *Int J High Perform Comput Appl*. 2019;33(6):1201-1220.
2. Gubbi J, Buyya R, Marusic S, Palaniswami M. Internet of Things (IoT): a vision, architectural elements, and future directions. *Future Gener Comput Syst*. 2013;29(7):1645-1660. https://www.sciencedirect.com/science/article/pii/S0167739X13000241, https://doi.org/10.1016/j.future.2013.01.010
3. IEEE standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008); 2019:1-84. https://doi.org/10.1109/IEEESTD.2019.8766229
4. So H. Introduction to fixed point number representation. https://inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html. Accessed 2021.
5. Heath MT. *Scientific Computing: An Introductory Survey*. 2nd ed. SIAM-Society for Industrial and Applied Mathematics; 2018.
6. The GNU C library reference manual appendix A.5.3.1. https://www.gnu.org/software/libc/manual/html_node/Floating-Point-Concepts.html. Accessed 2021.
7. Severance C, Dowd K. *High Performance Computing*. OpenStax CNX . 2010. https://cnx.org/contents/u4IVVH92@5.1:zWRe5Cf_@3/History-of-IEEE-Floating-Point-Format
8. Severance C. *An Interview with the Old Man of Floating-Point*. IEEE Computer; 1998. https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html
9. Stannered. https://commons.wikimedia.org/w/index.php?curid=3357169. Accessed 2021.
10. Sampson A, Dietl W, Fortuna E, Gnanapragasam D, Ceze L, Grossman D. EnerJ: approximate data types for safe and general low-power computation. Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '11. Association for Computing Machinery; 2011:164-174; New York, NY. https://doi.org/10.1145/1993498.1993518
11. Gupta G. What's the difference between single-, double-, multi- and mixed-precision computing? 2019. https://blogs.nvidia.com/blog/2019/11/15/whats-the-difference-between-single-double-multi-and-mixed-precision-computing/
12. Training with mixed precision. https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html. Accessed 2021.
13. What every computer scientist should know about floating-point arithmetic; 2000. https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
14. Hauser JR. Handling floating-point exceptions in numeric programs. *ACM Trans Program Lang Syst*. 1996;18(2):139-174. https://doi.org/10.1145/227699.227701
15. Esmaeilzadeh H, Blem E, Amant RS, Sankaralingam K, Burger D. Dark silicon and the end of multicore scaling. Proceedings of the 2011 38th Annual International Symposium on Computer Architecture (ISCA); 2011:365-376.
16. Theis TN, Wong HSP. The end of Moore's law: a new beginning for information technology. *Comput Sci Eng*. 2017;19(2):41-50. https://doi.org/10.1109/MCSE.2017.29
17. Harrington P, Yoo W. Diagnosing parallel I/O bottlenecks in HPC applications. Nakajima K, *Supercomputing 2017 ACM Student Research Competition Poster*. ACM; 2017.
18. Johnson J. Making floating point math highly efficient for AI hardware; 2018. https://engineering.fb.com/2018/11/08/ai-research/floating-point-math/
19. Bfloat16: the secret to high performance on cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus. Accessed 2021.
20. bfloat16 floating-point format. https://handwiki.org/wiki/Bfloat16_floating-point_format. Accessed 2021.
21. Micikevicius P, Narang S, Alben J, et al. *Mixed Precision Training*. 2018.
22. Kharya P. What is the tensor-float-32 precision format? https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/. Accessed 2021.
23. NVIDIA Ampere architecture in-depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/. Accessed 2021.
24. Automatic mixed precision for deep learning. https://developer.nvidia.com/automatic-mixed-precision
25. Gustafson J, Yonemoto I. Beating floating point at its own game: posit arithmetic. *Supercomput Front Innov*. 2017;4(2):71–86. https://doi.org/10.14529/jsfi170206
26. Murillo R, Barrio AAD, Botella G. Deep PeNSieve: a deep learning framework based on the posit number system. *Digit Signal Process*. 2020;102:102762. https://doi.org/10.1016/j.dsp.2020.102762
27. Gustafson J. Stanford seminar: beyond floating point: next generation computer arithmetic. https://www.youtube.com/watch?v=aP0Y1uAA-2Y. Accessed 2021.
28. Posit standard documentation release 4.9-draft. technical report posit working group; 2020. https://00248909357804553642.googlegroups.com/attach/7967db941064b/PositStandard4.9.pdf?part=0.1.1&view=1&view=1&vt=ANaJVrEkctbXTCx1pBtrQErZYo7WTfi1Fibi2-XyqKJvwCLf2M4Z9Lb9qnpEvMH32qY6_tYJY7m3CyN1T7JDU404U6xil7kZXoqXxmBHJPZKb6_uSiuByTc. Accessed 2021.

29. Uguen Y, Forget L & de Dinechin F Evaluating the hardware cost of the posit number system. Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications (FPL); 2019:106-113.
30. Gustafson J. *Posit Arithmetic*. PositHub; 2017.
31. SoftPosit. https://gitlab.com/cerlane/SoftPosit. Accessed 2021.
32. SoftPosit-Python. https://gitlab.com/cerlane/SoftPosit-Python. Accessed 2021.
33. SoftPosit-Math. https://gitlab.com/cerlane/softposit-math. Accessed 2021.
34. Guérin C. bfp: beyond floating-point. https://github.com/libcg/bfp. Accessed 2021.
35. Ruffaldi E. cppPosit. https://github.com/eruffaldi/cppPosit. Accessed 2021.
36. Omtzigt ETL, Gottschling P, Seligman M, Zorn W. Universal numbers library: design and implementation of a high-performance reproducible number systems library; 2020. arXiv:2012.11011.
37. Tiwari S, Gala N, Rebeiro C, Kamakoti V. PERI: a configurable Posit enabled RISC-V core. *ACM Trans Archit Code Optim*. 2021;18(3):1–26. https://doi.org/10.1145/3446210
38. Jaiswal MK, So HK. Architecture generator for type-3 unum posit adder/subtractor. Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS); 2018:1-5.
39. Jaiswal M, So H. Universal number posit arithmetic generator on FPGA. Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition; 2018:1159-1162
40. Jaiswal MK. https://github.com/manish-kj/Posit-HDL-Arithmetic. Accessed 2021.
41. Jaiswal MK, So HKH. PACoGen: a hardware posit arithmetic core generator. *IEEE Access*. 2019;7:74586-74601. https://doi.org/10.1109/ACCESS.2019.2920936
42. Jaiswal MK. https://github.com/manish-kj/PACoGen. Accessed 2021.
43. Chaurasiya R, Gustafson J. Parameterized posit arithmetic hardware generator. Proceedings of the 2018 IEEE 36th International Conference on Computer Design (ICCD); 2018:334-341.
44. Xiao F, Liang F, Wu B, Liang J, Cheng S, Zhang G. Posit arithmetic hardware implementations with the minimum cost divider and SquareRoot. *Electronics*. 2020;9(10). https://doi.org/10.3390/electronics9101622
45. Johnson J. Rethinking floating point for deep learning; 2018. http://arxiv.org/abs/1811.01721
46. Waterman A, Lee Y, Patterson DA, Asanovia K. The RISC-v instruction set manual, volume I: user-level ISA, version 2.0. technical report UCB/EECS-2014-54, EECS Department, University of California, Berkeley; 2014. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html
47. Gustafson J. RISC-V proposed extension for 32-bit posits; 2018. https://posithub.org/docs/RISC-V/RISC-V.htm
48. Tiwari S, Gala N, Rebeiro C, Kamakoti V. PERI: a posit enabled RISC-V core; 2019. arXiv:1908.01466 [cs] 2019. http://arxiv.org/abs/1908.01466
49. Jain R, Sharma N, Merchant F, Patkar S, Leupers R. CLARINET: a RISC-V based framework for posit arithmetic empiricism; 2020. https://arxiv.org/abs/2006.00364.
50. Cococcioni M, Rossi F, Ruffaldi E. A fast approximation of the hyperbolic tangent when using posit numbers and its application to deep neural networks; 2020. https://doi.org/10.1007/978-3-030-37277-4_25
51. Cococcioni M, Rossi F, Ruffaldi E, Saponara S. Fast approximations of activation functions in deep neural networks when using posit arithmetic. *Sensors*. 2020;20(5):1515. https://doi.org/10.3390/s20051515
52. Judd P, Albericio J, Hetherington T, Aamodt TM, Jerger NE, Moshovos A. Proteus: exploiting numerical precision variability in deep neural networks. Proceedings of the 2016 International Conference on SupercomputingICS '16. Association for Computing Machinery; 2016; https://doi.org/10.1145/2925426.2926294
53. Gysel P, Motamedi M, Ghiasi S. Hardware-oriented approximation of convolutional neural networks. *CoRR*. 2016;abs/1604.03168.
54. Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y. Quantized neural networks: training neural networks with low precision weights and activations. *J Mach Learn Res*. 2017;18(1):6869-6898.
55. Mishra AK, Marr D. Apprentice: using knowledge distillation techniques to improve low-precision network accuracy. ArXiv 2018; abs/1711.05852.
56. Fatemi Langroudi SH, Pandit T, Kudithipudi D. Deep learning inference on embedded devices: fixed-point vs posit. Proceedings of the 2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2); 2018:19-23
57. Cococcioni M, Ruffaldi E, Saponara S. Exploiting posit arithmetic for deep neural networks in autonomous driving applications. Proceedings of the 2018 International Conference of Electrical and Electronic Technologies for Automotive; 2018:1-6
58. Cococcioni M, Rossi F, Ruffaldi E, Saponara S. Novel arithmetics to accelerate machine learning classifiers in autonomous driving applications. Proceedings of the 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS); 2019:779-782.
59. Bernhardsson E, Martin Aumueller M. Erikbern/ann-benchmarks: benchmarks of approximate nearest neighbor libraries in python .
60. Langroudi HF, Carmichael Z, Gustafson JL, Kudithipudi D. PositNN framework: tapered precision deep learning inference for the edge. Proceedings of the 2019 IEEE Space Computing Conference (SCC); 2019:53-59.
61. Carmichael Z, Langroudi HF, Khazanov C, Lillie J, Gustafson JL, Kudithipudi D. Performance-efficiency trade-off of low-precision numerical formats in deep neural networks. Dimitrov V, *Proceedings of the Conference for Next Generation Arithmetic 2019CoNGA'19*. Association for Computing Machinery; 2019. https://doi.org/10.1145/3316279.3316282
62. Carmichael Z, Langroudi HF, Khazanov C, Lillie J, Gustafson JL & Kudithipudi D Deep positron: a deep neural network using the posit number system. Proceedings of the 2019 Design, Automation Test in Europe Conference Exhibition; 2019:1421-1426

63. Langroudi HF, Karia V, Gustafson JL, Kudithipudi D. Adaptive posit: parameter aware numerical format for deep learning inference on the edge. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops; 2020.
64. Lu J, Fang C, Xu M, Lin J, Wang Z. Evaluations on deep neural networks training using posit number system. *IEEE Trans Comput*. 2021;70(2):174-187. https://doi.org/10.1109/TC.2020.2985971
65. Klöwer M, Düben PD, Palmer TN. Number formats, error mitigation, and scope for 16-bit arithmetics in weather and climate modeling analyzed with a shallow water model. *J Adv Model Earth Syst*. 2020;12(10). https://doi.org/10.1029/2020MS002246
66. Klöwer M, Düben PD, Palmer TN. Posits as an alternative to floats for weather and climate models. Proceedings of the Conference for Next Generation Arithmetic 2019CoNGA'19. Association for Computing Machinery; 2019; ACM, New York, NY.
67. Lim JP, Shachnai M, Nagarakatte S. Approximating trigonometric functions for posits using the CORDIC method. Proceedings of the 17th ACM International Conference on Computing Frontiers CF '20; 2020:19-28; Association for Computing Machinery, New York, NY. https://doi.org/10.1145/3387902.3392632
68. de Dinechin F, Forget L, Muller JM, Uguen Y. Posits: the good, the bad and the ugly. Proceedings of the Conference for Next Generation Arithmetic 2019CoNGA'19. Association for Computing Machinery; 2019; ACM, New York, NY. https://doi.org/10.1145/3316279.3316285
69. Muller JM. *Elementary Functions: Algorithms and Implementation*. Birkhauser; 2005.
70. Muller JM, Brunie N, de Dinechin F, et al. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser; 2018.
71. Di S, Cappello F. Fast error-bounded lossy HPC data compression with SZ. Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2016:730-739.
72. Liang X, Di S, Tao D, et al. Improving performance of data dumping with lossy compression for scientific simulation. Proceedings of the 2019 IEEE International Conference on Cluster Computing (CLUSTER); 2019:1-11.
73. Liang X, Di S, Tao D, et al. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. Proceedings of the 2018 IEEE International Conference on Big Data (Big Data); 2018:438-447.
74. Lindstrom P. Fixed-rate compressed floating-point arrays. *IEEE Trans Vis Comput Graph*. 2014;20(12):2674-2683. https://doi.org/10.1109/TVCG.2014.2346458
75. Alouani I, Ben Khalifa A, Merchant F, Leupers R. An investigation on inherent robustness of posit data representation. Proceedings of the 2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID); 2021:276-281.
76. Snir M, Wisniewski RW, Abraham JA. Addressing failures in exascale computing. *Int J High Perform Comput Appl*. 2014;28(2):129-173.
77. Sridharan V, DeBardeleben N, Blanchard S, et al. Memory errors in modern systems: the good, the bad, and the ugly. Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating SystemsASPLOS '15. Association for Computing Machinery; 2015:297-310; New York, NY. https://doi.org/10.1145/2775054.2694348
78. Avizienis A, Laprie JC, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans Depend Secur Comput*. 2004;1(1):11-33. https://doi.org/10.1109/TDSC.2004.2
79. Elliott J, Mueller F, Stoyanov M, Webster C. *Quantifying the Impact of Single Bit Flips on Floating Point Arithmetic. Technical Report*. Oak Ridge National Laboratory; 2013.
80. Calhoun J, Snir M, Olson LN, Gropp WD. Towards a more complete understanding of SDC propagation. Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing HPDC '17. Association for Computing Machinery; 2017:131-142; ACM, New York, NY. https://doi.org/10.1145/3078597.3078617
81. Kuang-Hua H, Abraham JA. Algorithm-based fault tolerance for matrix operations. *IEEE Trans Comput*. 1984;C-33(6):518-528. https://doi.org/10.1109/TC.1984.1676475
82. Cappello F, Al G, Gropp W, Kale S, Kramer B, Snir M. Toward exascale resilience: 2014 update. *Supercomput Front Innov Int J*. 2014;1(1):5-28. https://doi.org/10.14529/jsfi140101
83. Gupta S, Patel T, Engelmann C, Tiwari D. Failures in large scale systems: long-term measurement, analysis, and implications. Proceedings of the International Conference for High Performance Computing, Networking, Storage and AnalysisSC '17; 2017; Association for Computing Machinery, New York, NY. 10.1145/3126908.3126937
84. Gottscho M, Schoeny C, Dolecek L, Gupta P. Software-defined error-correcting codes. Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W); 2016:276-282.
85. Poulos A, Wallace D, Robey R, et al. Improving application resilience by extending error correction with contextual information. Proceedings of the 2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS); 2018:19-28.
86. Wallace D, Jones WM, Robey R, Monroe L, Grové T, DeBardeleben N. Impact of contextual error correction techniques in CLAMR. In: Proceedings of the 2020 SoutheastCon; 2020:1-2.
87. Fang B, Guan Q, Debardeleben N, Pattabiraman K, Ripeanu M. LetGo: a lightweight continuous framework for HPC applications under failures. Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed ComputingHPDC '17. Association for Computing Machinery; 2017:117-130; ACM, New York, NY.
88. Fang B, Halawa H, Pattabiraman K, Ripeanu M, Krishnamoorthy S. BonVoision: leveraging spatial data smoothness for recovery from memory soft errors. Proceedings of the ACM International Conference on Supercomputing ICS '19; 2019:484-496; Association for Computing Machinery, New York, NY.
89. Henry G, Tang PTP, Heinecke A. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations; 2019.
90. Defour D, De Oliveira CP, Istoan M, Petit E. Shadow computation with BFloat16 to compute numerical accuracy. Proceedings of the IEEE 28th Symposium on Computer Arithmetic (ARITH) Virtual Conference; 2021; France.

AUTHOR BIOGRAPHIES

Alexandra Poulos is a graduate Computer Engineering Ph.D. student at Clemson University. She received a B.S. in Computer Science and a B.S. in Applied Mathematics from Coastal Carolina University in 2018. Her research interests include fault tolerance and alternative floating-point representations.

Sally A. McKee received her bachelor's degree in Computer Science from Yale University, master's from Princeton University, and doctorate from the University of Virginia. Her dissertation advisor was Bill Wulf, with whom she worked on memory systems architecture. Together they coined the now-common term the "memory wall" to describe a situation in which processors are always waiting on memory, and CPU performance is therefore entirely limited by memory performance. McKee joined the University of Utah's School of Computing as a Research Assistant Professor in 1998. She joined Cornell University's Computer Systems Lab within the School of Electrical and Computer Engineering in 2002. She moved to the Department of Computer Science and Engineering at Chalmers University of Technology in 2008, and she joined the Holcombe Department of Electrical and Computer Engineering at Clemson University in 2018.

Jon C. Calhoun is an Assistant Professor in the Holcombe Department of Electrical and Computer Engineering at Clemson University. He received a B.S. in Computer Science from Arkansas State University in 2012, a B.S. in Mathematics from Arkansas State University in 2012, and a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2017. His research interests lie in fault tolerance and resilience for high-performance computing (HPC) systems and applications, lossy and lossless data compression, scalable numerical algorithms, power-aware computing, and approximate computing.

How to cite this article: Poulos A, McKee SA, Calhoun JC. Posits and the state of numerical representations in the age of exascale and edge computing. *Softw Pract Exper*. 2021;1–17. https://doi.org/10.1002/spe.3022