Provably Correct Posit Arithmetic with Fixed-Point Big Integer

Shin Yee Chung
SpeedGo Computing
Singapore
shinyee@speedgocomputing.com

ABSTRACT

Floating-point number format is used extensively in many applications, especially scientific software. The applications rely on efficient hardware floating-point support to perform arithmetic operations. With the advent of multicore CPUs and massively parallel GPUs, the memory bandwidth of a computer system is increasingly limited for each of the compute cores. The limited memory bandwidth is a serious bottleneck to the system performance. The posit number format [12] is a promising approach to improve the accuracy of the arithmetic operations with more efficient use of bit storage, hence, reducing memory contention. However, robust and reliable software implementations of posit arithmetic libraries in C/C++ or Python are not readily available. In this paper, we seek to develop provably correct posit arithmetic based on fixed-point big integers. A robust and reliable implementation can then serve as a reference for other hardware-optimized implementations, as a test bed for applications to experiment with different posit bit configurations, and to analyze the relative errors of using smaller bit sizes in the posit numbers compared to using the native 32-bit or 64-bit floating-point numbers.

CCS CONCEPTS

• **Mathematics of computing** → **Arbitrary-precision arithmetic**; • **Software and its engineering** → *Software verification and validation*; • **Computer systems organization** → Architectures;

KEYWORDS

posit, arithmetic, arbitrary-precision, fixed-point, floating-point

ACM Reference Format:
Shin Yee Chung. 2018. Provably Correct Posit Arithmetic with Fixed-Point Big Integer. In *CoNGA 2018: Conference for Next Generation Arithmetic, March 28, 2018, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3190339.3190341

1 INTRODUCTION

Modern computer systems have widely adopted floating-point arithmetic [7] in processor hardware. The hardware supports provide fast execution of basic arithmetic operations such as addition, subtraction, multiplication, and division. A floating-point number, or *float*, represents a value as a signed fraction times an integer power of 2. This allows the representations of extremely large or extremely small numbers with a limited number of bits, trading off the ability to represent a certain range of values exactly.

A floating-point number format can be configured to have different number of bits for the fraction part (called the mantissa bits), and the exponent part (called the exponent bits). Converting between floating-point number formats with different configurations is difficult and inefficient. The IEEE 754 floating-point standard [15] has been widely adopted to standardize on the configurations of the bit allocations, the rounding mode when a value cannot be represented exactly in the particular configuration, and the handling of invalid arithmetic operations such as a divide by zero operation.

However, the proposed IEEE 754 standard is not without its limitations. The limitations include:

- a wastefully large number of bit patterns that redundantly represent "Not A Number", NaN;
- regular and expensive software checks or hardware traps to ensure NaN robustness;
- limited numerical precision resulting from the fixed number of bits allocated for the exponent and mantissa;
- the hardware cost of handling denormalized numbers;
- rounding causes failure of the associative and distributive laws of algebra.

Gustafson proposed what we now call unum Type I numbers [11] that have variable bit sizes; unum Type III numbers [12], *posits*, use fixed bit sizes more efficiently to obtain higher accuracy than floats. Posit format simplifies the hardware and software implementations with just two special cases (zero and $\pm\infty$), simple rounding, and no denormalized numbers to handle. However, posit format is currently not supported by contemporary processors. Software libraries are becoming available for experiments to assess the precision impacts on applications, but software libraries that correctly implement posit arithmetic for various popular programming languages are currently lacking.

In this paper, we implement a posit arithmetic library using *big integer* libraries that allow arbitrarily large integer representation. It is important to have a robust and correct first implementation widely available for experiments with different applications. It helps to bootstrap more hardware-optimized implementations subsequently. Hence, correctness takes priority over speed for early implementation of the range of legitimate posit configurations.

The use of big integer libraries helps to handle the large integer arithmetic operations transparently. Hence, we can focus on getting the relations between posits and big integers right. Big integer has a long history with number theory [4] and cryptography. Software

Table 1: Mathematical notations

Symbol	Representation
P	A posit number.
$nbits$	The total number of bits in P.
N	The same as $nbits$.
es	The maximum number of exponent bits in P.
U	An integer, $2^{2^{es}}$.
s	The sign bit of P.
k	The power of U of P.
e	The power of two of P.
f	The fractional part of P.
h	The number of bits in the fractional part, f.
B	A fixed-point number, $B = x2^m$
x	The integer part of B.
m	The exponent part of B.

libraries for big integer [6, 9] are generally available on most platforms. Scripting languages, such as Python [17] and Ruby [5], often have big integer built-in.

To represent decimal numbers with integers, we adopt a fixed-point number approach. We use an integer to represent the sign and the magnitude, and another integer to represent the shift of the binary point. With both integers, we implicitly represent the value of a number. In the following sections, we use the notations in Table 1 in our formulations, unless explicitly specified otherwise.

2 RELATED WORK

Floats are widely supported in modern systems. However, the bit sizes of the floating-point numbers supported on the hardware are often fixed. On embedded systems, due to limited hardware resources, floating-point hardware may not be available. For certain applications, different precisions may be desired due to the performance needs, the storage limits, or the accuracy requirements. In these cases, a software implementation of floating-point arithmetic fills the gaps.

The GMP [9] open source software library provides a C interface for arbitrary precision signed integer, rational number, and floating-point numbers. The software library automatically and dynamically allocates more memory storage when the numbers get large. The MPFR [6] open source software library extends the GMP with more mathematical functions available for floating-point numbers. The precision of the floating-point numbers can be set at run time. However, unlike big integer, the floating-point arithmetic operations do not allocate more bits to maintain accuracy higher than the working level of precision. Hence, if we require high precision for only a very small part of an application, programmers typically set all the floating-point numbers in the application to the highest precision desired. The numbers beyond the set precision will round to the higher or lower floating-point numbers that can be represented in the format depending on the rounding mode used.

There are a number of software implementations of the posit format. Leong [13] implemented decimal number and posit number conversions for a number of selected bit sizes. Guérin [10] implemented posit arithmetic in C++ supporting bit sizes up to 32-bit. Unfortunately, its comparison operators produce incorrect results with negative numbers. Yonemoto [18] implemented posit arithmetic backed by the native floating-point numbers. His approach converts posits to and from native floating-point numbers before and after each arithmetic operation. The correctness of the posit calculations for that approach has only been verified up to 16-bit posit numbers.

Ruffaldi [16] implemented posit arithmetic with C++ templates supporting up to 64-bit posits; however, correct posit round-to-nearest was not included, and bits beyond the posit precision are simply truncated. Omtzigt [14] implemented posit arithmetic in C++ [12]. At the time of writing, his implementation of the multiplication and the division arithmetic with posit rounding has only been completed recently. The division operation computes the quotient with an excessive amount of bits to ensure small truncations. The remainder of the division is not kept for subsequent posit binary encoding, so there is no assurance that the posit rounding with tie breaking conditions can be handled correctly. Certain legitimate posit configurations such as $\{nbits = 2, es = 0\}$ or $\{nbits = 3, es = 1\}$ are not supported.

Other implementations include posits implemented in Rust [2] and the early implementation of posits in Julia [19].

Robust, flexible implementations of posits with different bit sizes (especially bit sizes beyond the native data type sizes) are still lacking. Correct posit rounding is omitted in many implementations.

3 POSIT DEFINITIONS

This section provides a brief description of the posit number format based on [12] as the basis for the subsequent sections. For details of the binary format, the encoding and decoding of the posit bits, and the design and error analysis of posits, we refer the readers to the article [12].

Definition 3.1. A *posit configuration* is the pair of numbers $nbits \geq 2$ and $es \geq 0$. All the posit numbers of a specific posit configuration form a *domain*. Each of the numbers in the domain has a total of $nbits$ bits with a maximum of es exponent bits.

The es integer is used to select the exponential scaling of the represented numbers. A larger es value increases the dynamic range of the numbers in the domain.

Definition 3.2. The posit number binary format defines zero and $\pm\infty$ with special bit patterns. The bit pattern that consists of all zero bits represents a zero. The bit pattern that consists of a one bit at the most significant bit and all zero bits elsewhere represents $\pm\infty$ or "NaR" for "Not-a-Real," used to express indeterminate forms such as $0/0$ as well as results that are complex such as square roots of negative numbers.

The special numbers do not follow the general posit number formulation of the represented number. Hence, these numbers are often specially checked and handled.

Definition 3.3. A posit number, $P \notin \{0, \pm\infty\}$ represents the value of $(-1)^s U^k 2^e (1 + f/2^h)$, where

s	is the sign bit value, one or zero
U	is an integer, $2^{2^{es}}$
k	is an integer defined by the regime bits
e	is an integer $\in [0, 2^{es} - 1]$ defined by the exponent bits
f	is an integer $\in [0, 2^h - 1]$ defined by the fractional bits
h	is the number of bits allocated for the fractional part

Definition 3.4. A posit number consists of a sign bit at the most significant bit, followed by one or more regime bits, followed by zero or up to *es* exponent bits, followed by zero or more fractional bits. Bits are allocated in priority order first to the sign bit and regime bits, then the exponent bits, lastly the fractional bits.

Definition 3.5. The regime bits of a positive posit number are encoded using their run length, L. From the first regime bit, r, immediately after the sign bit, the length of the run is the number of consecutive bits (including the first r bit) that have the same bit value as the r bit, before a negated r bit is encountered or the end of the bit stream is encountered. The run of r bits and zero or one negated r bit form the regime bits.

The run length L and the r bit value determine the k value in the posit number formulation in Definition 3.3. If the r bit is a one, then $k = L - 1$. If the r bit is a zero, then $k = -L$.

Note that when the r bit is a zero, there must be a negated r bit after the run. Otherwise, the bit pattern represents one of the two exception values, either zero or NaR.

Definition 3.6. The exponent bits of a positive posit number follow immediately after the regime bits up to a maximum of *es* bits. The exponent bits are rounded and truncated when the number of remaining bits after the regime bits is less than *es*. The truncated exponent bits are implicitly zero bits. The exponent value e in Definition 3.3 equals the unsigned integer of the *es* exponent bits by including the truncated exponent bits (if any).

Definition 3.7. The fractional bits of a positive posit number follow immediately after the exponent bits, taking up all the remaining bits (if any). The fractional value f in Definition 3.3 equals the unsigned integer of the fractional bits, and equals zero if there are no fractional bits.

Definition 3.8. A negated posit number, $-P$, has the bit pattern of the two's complement of the posit number P.

Definition 3.9. When the numbers of a posit configuration cannot represent a value V exactly, the value is rounded to the nearest posit number P, and breaks to even on a tie. A posit bit pattern is *even* if the least significant bit is a zero bit. When one or more fractional bits are to be rounded, the nearest posit number has the smallest absolute difference $|P - V|$. When one or more exponent bits are to be rounded, the nearest posit number has the same sign and the smallest ratio $|\lg(P/V)|$. Thus, posits never round to zero or $\pm\infty$ (NaR).

Definition 3.10. Rounding exponent bits refers to reducing the number of exponent bits from *es* to $es - t$, where t is the number of exponent bits to be rounded, and $0 < t \leq es$. Similarly, rounding fractional bits refers to reducing the number of fractional bits from h to $h - t$, where t is the number of fractional bits to be rounded, and $0 < t \leq h$.

4 FIXED-POINT BIG INTEGER MODEL

We define the basic requirements of the fixed-point big integer model, B, that we use to implement the posit arithmetic. To bootstrap a reliable implementation of the posit number, we assume a powerful backing big integer model.

Definition 4.1. A *big integer* is a signed integer with basic arithmetic capability of addition, subtraction, multiplication, and division. The division operation implements a floor division.

The Python language adopted floor division while C and C++ languages adopted division rounded towards zero. Note that for a division rounded towards zero and positive divisor, when the remainder is negative, it can be converted to a floor division by incrementing the remainder by the divisor, and decrementing the quotient by one.

Definition 4.2. A big integer allocates resources for representing large numbers dynamically.

Note that for posit configurations with small *nbits* and *es*, such as $\{nbits = 16, es = 1\}$, a native 64-bit long integer may suffice as long as the integers do not overflow in all the operations. Bounding the minimum number of native integer bits required for each of the posit configurations is beyond the scope of this paper, but it is potentially useful in practice.

Definition 4.3. A big integer provides accesses to the binary bits of the integer. Bit operations such as left shift '$\ll$', right shift '$\gg$', bit-wise AND '&', bit-wise OR '|', bit-wise XOR '$\oplus$', and bit-wise negation '~', may be performed on the integer. The format is two's complement, so bit shifting operations maintain the sign of the integer with sign extensions.

Definition 4.4. A *fixed-point number* is of the form $B = x2^m$, where x and m are big integers. If x is an odd integer, the fixed-point number is a *normalized* fixed-point number.

5 PROVABLY CORRECT POSIT ARITHMETIC

In this section, we consider posit arithmetic with decoded posit numbers. We are given the *type* of the posit number $P \in \{0, \pm\infty, \text{normal}\}$ where a normal posit is in $\mathbb{R} \setminus \{0\}$; if the posit number P is normal, we are given the values of $\{N, es, s, k, e, f, h\}$, such that by Definition 3.3, $P = (-1)^s U^k 2^e (1 + f/2^h)$.

5.1 Decoded posit and fixed-point conversions

Lemma 5.1. *A posit number $P \notin \{\pm\infty\}$ to a fixed-point number B conversion can be computed with only integer operations.*

Proof. If the posit number P is a zero, then $B = x2^m = 0$ with $x = 0$. Otherwise, given that $P \notin \{\pm\infty\}$, by Definition 3.3, the posit number

$$\begin{aligned} P &= (-1)^s U^k 2^e \left(1 + \frac{f}{2^h}\right) \\ &= (-1)^s 2^{2^{es}k} 2^e 2^{-h} 2^h \left(1 + \frac{f}{2^h}\right) \\ &= (-1)^s \left(2^h + f\right) 2^{2^{es}k+e-h} = B \end{aligned}$$

where $B = x2^m$ with $x = (-1)^s(2^h + f)$ and $m = 2^{es}k + e - h$. Since the computation of x and m involves only integer operations, they can be computed using big integers for the posit numbers with arbitrarily large bit sizes. ■

We need not represent $\pm\infty$ (NaR) in a fixed-point number B. NaR serves a role similar to "NaN" in floating-point arithmetic, and the result of any arithmetic operation on NaR returns NaR. This ensures that any indeterminate calculations are revealed in the final output. While it is tempting to allow $1/\pm\infty$ to be treated as zero, that behavior is reserved for the interval version of posits, *valids*.

Lemma 5.2. *A fixed-point number B to a posit number P conversion can be computed with only integer operations.*

Proof. If $B = x2^m$ is a zero with $x = 0$, by Definition 3.2, P is a bit string of all zeros. Otherwise, by Definition 3.3 and Definition 4.4, the fixed-point number B and the posit number P

$$\begin{aligned} B &= x2^m \\ &= x2^{-g}2^{m+g}, \text{and} \\ P &= (-1)^s 2^{2^{es}k} 2^e \left(1 + \frac{f}{2^h}\right) \\ &= (-1)^s \left(1 + \frac{f}{2^h}\right) 2^{2^{es}k+e}. \end{aligned}$$

Let $g = h$, matching the fraction part of the posit number P and the fixed-point number B. Then

$$\begin{aligned} s &= \begin{cases} 0 & \text{if } x \geq 0 \\ 1 & \text{otherwise} \end{cases} \\ |x|2^{-g} &= 1 + \frac{f}{2^h} \\ |x| &= 2^h + f \\ h &= \lfloor \lg |x| \rfloor \quad \text{since } f < 2^h \\ f &= |x| - 2^h. \end{aligned}$$

Note that $\lfloor \lg |x| \rfloor$ of an integer x can be computed with just bit shifts and countings. With $g = h$, matching the exponent parts of the posit P and the fixed-point number B,

$$\begin{aligned} m + h &= 2^{es}k + e \\ k &= \left\lfloor \frac{m+h}{2^{es}} \right\rfloor \\ e &= (m+h) \bmod 2^{es}. \end{aligned}$$

Since the divisor 2^{es} is greater than zero, $0 \leq e < 2^{es}$ as described in [8]. Thus, the computed value e fits the posit number requirements defined in Definition 3.3. Both e and k can be computed with only integer operations. ■

5.2 Addition and subtraction

Lemma 5.3. *The addition of two fixed-point numbers $B_a = x_a2^{m_a}$ and $B_b = x_b2^{m_b}$ produces $B_c = x_c2^{m_c}$ where $m = \max(m_a, m_b)$, $x_c = x_a2^{m-m_b} + x_b2^{m-m_a}$, and $m_c = m_a + m_b - m$.*

Proof. Given the fixed-point numbers B_a and B_b,

$$\begin{aligned} m &= \max(m_a, m_b) \\ B_a + B_b &= x_a2^{m_a} + x_b2^{m_b} \\ &= \frac{x_a}{2^{m-m_a}}2^{m_a}2^{m-m_a} + \frac{x_b}{2^{m-m_b}}2^{m_b}2^{m-m_b} \\ &= \left(\frac{x_a}{2^{m-m_a}} + \frac{x_b}{2^{m-m_b}}\right)2^m \\ &= \left(x_a2^{m-m_b} + x_b2^{m-m_a}\right)2^{m_a+m_b-m} \\ &= x_c2^{m_c}. \end{aligned}$$

■

Corollary 5.4. *The subtraction of two fixed-point numbers B_a and B_b produces $B_c = x_c2^{m_c}$ where $m = \max(m_a, m_b)$, $x_c = x_a2^{m-m_b} - x_b2^{m-m_a}$, and $m_c = m_a + m_b - m$.*

Proof. It follows the proof of Lemma 5.3, with x_b negated. ■

5.3 Multiplication and division

Lemma 5.5. *The multiplication of two fixed-point numbers $B_a = x_a2^{m_a}$ and $B_b = x_b2^{m_b}$ produces $B_c = x_c2^{m_c}$ where $x_c = x_ax_b$ and $m_c = m_a + m_b$.*

Proof. Given the fixed-point numbers B_a and B_b,

$$\begin{aligned} B_a \times B_b &= x_a2^{m_a} \times x_b2^{m_b} \\ &= (x_ax_b)2^{m_a+m_b} \\ &= x_c2^{m_c}. \end{aligned}$$

■

Lemma 5.6. *The division of two positive fixed-point numbers $B_a = x_a2^{m_a}$ and $B_b = x_b2^{m_b}$, $B_a/B_b = (Q + r_h/y_b)2^m$ where*

$$\begin{aligned} Q &= 2^h \sum_{i=0}^{h} 2^{-i}q_i \\ m &= m_a - m_b + h_a - h_b - h - g \\ q_i &= \begin{cases} \left\lfloor \frac{2r_{i-1}}{y_b} \right\rfloor & \text{if } i > 0 \\ 1 & \text{if } i = 0 \end{cases} \\ r_i &= \begin{cases} 2r_{i-1} \bmod y_b & \text{if } i > 0 \\ 2^g y_a \bmod y_b & \text{if } i = 0 \end{cases} \\ g &= \begin{cases} 1 & \text{if } y_a < y_b \\ 0 & \text{otherwise} \end{cases} \\ h_a &= \lfloor \lg x_a \rfloor \\ h_b &= \lfloor \lg x_b \rfloor \\ h &= \text{number of fractional bits required} \\ H &= \max(h_a, h_b) \\ y_a &= x_a2^{-h_a+H} \\ y_b &= x_b2^{-h_b+H} \end{aligned}$$

Proof. Given $x_a, x_b \in \mathbb{Z}^+$, let

$$\begin{aligned} h_a &= \lfloor \lg x_a \rfloor, \\ h_b &= \lfloor \lg x_b \rfloor, \text{and} \\ H &= \max(h_a, h_b). \end{aligned}$$

Then

$$\begin{aligned} x_a 2^{-h_a} &\in [1,2) \\ x_b 2^{-h_b} &\in [1,2). \end{aligned}$$

To ensure that only integer operations are required, let

$$\begin{aligned} y_a &= x_a 2^{-h_a+H} \in \mathbb{Z}^+ \\ y_b &= x_b 2^{-h_b+H} \in \mathbb{Z}^+ \\ \frac{y_a}{y_b} &\in \left(\frac{1}{2}, 2\right) \end{aligned}$$

Dividing B_a by B_b gives

$$\begin{aligned} \frac{B_a}{B_b} &= \frac{x_a}{x_b} 2^{m_a-m_b} \\ &= \frac{2^g x_a 2^{-h_a+H}}{x_b 2^{-h_b+H}} 2^{m_a-m_b+h_a-h_b-g} \\ &= \frac{2^g y_a}{y_b} 2^{m_a-m_b+h_a-h_b-g}, \text{ where} \\ g &= \begin{cases} 1 & \text{if } y_a < y_b, \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

The factor 2^g ensures that the division $2^g y_a / y_b \in [1,2)$ is in scientific form, matching the fractional part of a posit number. The division computes a quotient and a remainder such that

$$\begin{aligned} 2^g y_a &= q_0 y_b + r_0 \\ q_0 &= 1 \text{ and } r_0 = 2^g y_a \bmod y_b, \text{ since } 2^g y_a \geq y_b. \end{aligned}$$

Expand the remainder r_0 using long division:

$$\begin{aligned} 2^g y_a &= q_0 y_b + \frac{1}{2} 2r_0 \\ &= q_0 y_b + \frac{1}{2}\left(\left\lfloor \frac{2r_0}{y_b} \right\rfloor y_b + 2r_0 \bmod y_b\right) \\ &= q_0 y_b + \frac{1}{2}\left(\left\lfloor \frac{2r_0}{y_b} \right\rfloor y_b + \frac{1}{2}\left(\left\lfloor \frac{2r_1}{y_b} \right\rfloor y_b + 2r_1 \bmod y_b\right)\right) \\ &= y_b \sum_{i=0}^{h} 2^{-i} q_i + \frac{r_h}{2^h} \\ &= \left(y_b 2^h \sum_{i=0}^{h} 2^{-i} q_i + r_h\right) 2^{-h} \\ &= (y_b Q + r_h)\, 2^{-h}, \text{ where} \\ q_0 &= 1, \\ r_0 &= 2^g y_a \bmod y_b, \\ q_{i>0} &= \left\lfloor \frac{2r_{i-1}}{y_b} \right\rfloor = \begin{cases} 1 & \text{if } 2r_{i-1} \geq y_b, \\ 0 & \text{otherwise,} \end{cases} \\ r_{i>0} &= 2r_{i-1} \bmod y_b = \begin{cases} 2r_{i-1} - y_b & \text{if } 2r_{i-1} \geq y_b, \\ 2r_{i-1} & \text{otherwise, and} \end{cases} \\ h &= \text{ number of fractional bits.} \end{aligned}$$

$$\therefore \quad \frac{B_a}{B_b} = \frac{2^g y_a}{y_b} 2^{m_a-m_b+h_a-h_b-g} = \left(Q + \frac{r_h}{y_b}\right) 2^m.$$

■

5.4 Posit arithmetic and rounding

Lemma 5.7. *For a posit number with a residual, which may not be represented as a posit number, $P' = (-1)^s 2^{2^{es}k} 2^e (1 + (f + r/y)/2^h)$, where $0 \leq r < y$ and $y \geq 2$, rounding t exponent bits has the tie breaking point at 2^{t-1}, $f = 0$, and $r = 0$. Similarly, rounding t fractional bits has the tie breaking point at 2^{t-1} and $r = 0$. Rounding the residual r/y has the tie breaking point at $2r = y$.*

Proof. The tie breaking point of a rounding is equally near to the rounded up and the rounded down posit numbers. Given a posit number with a residual P', the posit number with t exponent bits rounded down

$$P_e^- = (-1)^s 2^{2^{es}k} 2^{\lfloor \frac{e}{2^t} \rfloor 2^t}$$

and the posit number with t exponent bits rounded up

$$P_e^+ = (-1)^s 2^{2^{es}k} 2^{(\lfloor \frac{e}{2^t} \rfloor + 1) 2^t}$$

The tie breaking point is equally near to the two posit numbers

$$\begin{aligned} P_{T_e} &= (-1)^s 2^{2^{es}k} 2^{(\lfloor \frac{e}{2^t} \rfloor + \frac{1}{2}) 2^t} \\ &= (-1)^s 2^{2^{es}k} 2^{\lfloor \frac{e}{2^t} \rfloor 2^t + 2^{t-1}} \end{aligned}$$

The tie breaking point P_{T_e} corresponds to having $T_e = 2^{t-1}, f = 0, r = 0$, where

$$T_e = e - \left\lfloor \frac{e}{2^t} \right\rfloor 2^t$$

Similarly, rounding t fractional bits we have

$$\begin{aligned} P_f^- &= (-1)^s 2^{2^{es}k} 2^e \left(1 + \frac{\left\lfloor \frac{f}{2^t} \right\rfloor 2^t}{2^h}\right) \\ P_f^+ &= (-1)^s 2^{2^{es}k} 2^e \left(1 + \frac{\left(\left\lfloor \frac{f}{2^t} \right\rfloor + 1\right) 2^t}{2^h}\right) \\ P_{T_f} &= (-1)^s 2^{2^{es}k} 2^e \left(1 + \frac{\left\lfloor \frac{f}{2^t} \right\rfloor 2^t + 2^{t-1}}{2^h}\right) \\ T_f &= f - \left\lfloor \frac{f}{2^t} \right\rfloor 2^t. \end{aligned}$$

The tie breaking point P_{T_f} corresponds to having $T_f = 2^{t-1}, r = 0$.

Rounding the residual r/y we have

$$\begin{aligned} P_r^- &= (-1)^s 2^{2^{es}k} 2^e \left(1 + \frac{f}{2^h}\right) \\ P_r^+ &= (-1)^s 2^{2^{es}k} 2^e \left(1 + \frac{f+1}{2^h}\right) \\ P_{T_r} &= (-1)^s 2^{2^{es}k} 2^e \left(1 + \frac{f + \frac{1}{2}}{2^h}\right). \end{aligned}$$

The tie breaking point P_{T_r} corresponds to having $r/y = 1/2$, that is $2r = y$. ■

Lemma 5.8. *Using the formulations in Lemma 5.7, rounding t exponent bits should be rounded down when $T_e < 2^{t-1}$, rounded up when $T_e > 2^{t-1} \vee (T_e = 2^{t-1} \wedge (f \neq 0 \vee r \neq 0))$ and perform tie breaking when $T_e = 2^{t-1} \wedge f = 0 \wedge r = 0$. Similarly, rounding t fractional bits should be rounded down when $T_f < 2^{t-1}$, rounded up when $T_f > 2^{t-1} \vee (T_f = 2^{t-1} \wedge r \neq 0)$ and perform tie breaking when $T_f = 2^{t-1} \wedge r = 0$. Rounding the residual r/y should be rounded down when $2r < y$, rounded up when $2r > y$, and perform tie breaking when $2r = y$.*

Proof. Based on Lemma 5.7, rounding t exponent bits has the tie breaking point at $T_e = 2^{t-1} \wedge f = 0 \wedge r = 0$. Since

$$f < 2^h \wedge \frac{r}{y} < 1,$$

when $T_e < 2^{t-1}$, the exact value P' is less than the tie breaking point P_{T_e}, hence should be rounded down. Since

$$f \geq 0 \wedge \frac{r}{y} \geq 0,$$

when $T_e > 2^{t-1} \vee (T_e = 2^{t-1} \wedge (f \neq 0 \vee r \neq 0))$, the exact value P' is greater than the tie breaking point, hence should be rounded up.

Similarly, rounding t fractional bits, we have

$$P' = P_{T_f} \text{ when } T_f = 2^{t-1} \wedge r = 0$$
$$P' < P_{T_f} \text{ when } T_f < 2^{t-1}$$
$$P' > P_{T_f} \text{ when } T_f > 2^{t-1} \vee (T_f = 2^{t-1} \wedge r \neq 0)$$

Rounding the residual r/y, we have

$$P' = P_{T_r} \text{ when } 2r = y$$
$$P' < P_{T_r} \text{ when } 2r < y$$
$$P' > P_{T_r} \text{ when } 2r > y$$

■

Theorem 5.9. *Using the formulations in Lemma 5.7, the residual r/y with arbitrarily large r and y can be reduced to $R \in \{0, 1, 2, 3\}$ while preserving the correctness of the posit rounding.*

Proof. Given a posit number with a residual, P', rounding of the exponent bits or the fractional bits requires the condition whether $r = 0$ or $r \neq 0$, not the exact value of r. Let

$$R = \begin{cases} 0 & \text{if } 2r < y \wedge r = 0 \\ 1 & \text{if } 2r < y \wedge r \neq 0 \\ 2 & \text{if } 2r = y \\ 3 & \text{if } 2r > y \end{cases}$$

Note that because $y \geq 2$, then $r \neq 0$ for both cases $2r = y$ and $2r > y$. During the rounding of the exponent bits or the fractional bits,

$$R = 0 \iff r = 0$$
$$R \neq 0 \iff r \neq 0$$

During the rounding of the residual r/y,

$$R < 2 \iff 2r < y$$
$$R = 2 \iff 2r = y$$
$$R > 2 \iff 2r > y$$

Therefore, reducing the residual r/y to R gives sufficient information for the subsequent posit rounding. ■

Table 2: Special posit number arithmetic

		Arithmetic results			
Type(a)	Type(b)	+	−	×	÷
normal	normal	0 ($a = -b$)	0 ($a = b$)	normal	normal
normal	±∞	±∞	±∞	±∞	±∞
normal	0	a	a	0	±∞
±∞	normal	±∞	±∞	±∞	±∞
±∞	±∞	±∞	±∞	±∞	±∞
±∞	0	±∞	±∞	±∞	±∞
0	normal	b	$-b$	0	0
0	±∞	±∞	±∞	±∞	±∞
0	0	0	0	0	±∞

Corollary 5.10. *A posit division can be augmented with a reduced residual R to ensure correct posit rounding on the result.*

Proof. It follows from Lemma 5.6 that a division produces a residual r/y, in addition to a fixed-point number B which can be converted to a posit number P. Using Theorem 5.9, the result of the division can be rounded correctly by the posit rounding requirements. ■

Lemma 5.11. *Posit arithmetic with special numbers* 0 *or* ±∞ *can be computed with the input types.*

Proof. Table 2 shows all the cases that involve special numbers 0 or ±∞ and their results. The arithmetic operations can be computed by checking the types of the input posit numbers. Notice that ±∞ behaves as NaR, not as unsigned infinity, so that indeterminate operations cannot misleadingly produce a normal posit number result. ■

Theorem 5.12. *Posit addition, subtraction, multiplication, and division can be computed using big integers.*

Proof. Given two posit numbers, the arithmetic operations with special posit numbers 0 or ±∞ can be computed with Table 2. The arithmetic operations with normal posit numbers can be computed by first converting the input posit numbers to fixed-point numbers using big integers, performing the arithmetic operations, and converting the results back to posit numbers. For a division operation, the result of the fixed-point division is augmented with a reduced residual R for the posit encoding to round correctly. ■

6 POSIT BINARY CODING

The posit design [12] defines the binary format of posit numbers. In a posit decoding algorithm, the regime bits, exponent bits, and fractional bits are extracted from a given posit bit pattern for a specific posit configuration as defined in Definition 3.4. The extracted bit values can be used to construct the value represented by the posit, as defined in Definition 3.2 and Definition 3.3.

A posit encoding algorithm does the reverse of the decoding algorithm in general, but not exactly. The number to be encoded can be any numbers represented by the posit formulation in Definition 3.3 without the total bit size restrictions. The number may be a result of an arithmetic operation that violates the total bit size constraint. The number may also be augmented with rounding information from the division operation. The encoding algorithm has to perform posit rounding as defined in Definition 3.9 to reduce the total bit size to the specific posit configuration.

6.1 Posit decoding

Algorithm 1 is a posit decoding algorithm that computes the posit number type and the values $\{s, k, e, f, h\}$ of a given bit pattern for a posit configuration $\{nbits, es\}$.

Line 1 and line 2 check the specific bit patterns for a zero and $\pm\infty$. Line 3 extracts the sign bit from the most significant bit. Line 4 applies two's complement to the bit pattern for a negative posit number. Subsequently, the decoding works the same for a positive posit number. Line 5 extracts the first regime bit after the sign bit. The variable 'i' starting from line 6 keeps track of the number of remaining bits to be decoded. Line 7 to line 9 counts the run length of the regime bits. Line 10 and line 11 compute the k value based on Definition 3.5. Line 12 computes the number of remaining bits after the regime bits. These bits are used for the exponent bits and the fractional bits. Line 14 extracts both the exponent bits and the fractional bits at the same time. Line 15 computes the number of remaining bits, h, for the fractional bits after allocating bits to the exponent bits. Line 16 extracts the fractional bits, f, after the exponent bits. Line 17 extracts the exponent bits, e. Line 18 scales up the exponent bits, e, by the number of truncated exponent bits. The truncated exponent bits are implicitly zeros.

6.2 Posit encoding

Algorithm 2 is a posit encoding algorithm for positive posit numbers. The s value is ignored in the encoding. Given the values $\{k, e, f, h\}$, the posit configuration $\{nbits, es\}$, and the rounding information R, the encoding algorithm computes and returns the bit pattern in posit binary format defined in [12].

The Mask(n) function used in the algorithm computes a bit mask with n number of one bits. Line 1 pre-computes the bit pattern of *maxpos*, defined as the maximum number that can be represented in the posit configuration. This is used in the rounding process to prevent rounding to $\pm\infty$. Starting from line 2, the variable 'n' keeps track of the number of remaining bits to be encoded. Line 5 to line 7 encodes a positive k when there are enough bits for the regime bits without rounding. Line 7 appends the negated r bit only if there is at least one remaining bit. Line 9 rounds the value to maxpos, with the corresponding bit pattern Mask($N-1$) or 'maxp' at line 1, as the value is too large to be represented in the posit configuration, and posit rounding never rounds to $\pm\infty$. Line 11 to line 13 encode a negative k with regime zero bits terminated by a one bit (the negated r bit). When the value is too small to be represented in the posit configuration, the regime bits correspond to the bit pattern of *minpos*, the minimum positive number that can be represented in the posit configuration, and it marks the number as rounded, where the number of remaining bits will be zero.

Algorithm 1: DecodePosit: Decode posit binary

```
input  : Posit binary bits, and configuration (N, $es$)
output : Posit number $P = (-1)^s 2^{2^{es}k} 2^e (1 + f/2^h)$, and
         Type(P)
1  if bits == 0 then return Type(P) ← zero type
2  if bits == (1 ≪ (N − 1)) then return Type(P) ← ±∞ type
3  s ← (bits ≫ (N − 1)) & 1
4  if s == 1 then bits ← −bits
5  rbit ← (bits ≫ (N − 2)) & 1
6  i ← N − 2
7  while i ≥ 0 AND ((bits ≫ i) & 1) == rbit do
8      i ← i − 1
9  nrs ← N − 2 − i
10 if rbit == 0 then k ← −nrs
11 else k ← nrs − 1
12 nefbits ← max(0, N − nrs − 2)
13 efmask ← (1 ≪ nefbits) − 1
14 ef ← bits & efmask
15 h ← max(0, nefbits − es)
16 f ← ef & (efmask ≫ es)
17 e ← ef ≫ h
18 e ← e ≪ max(0, es − nefbits)
19 Type(P) ← normal posit number type
20 return Type(P), s, k, e, f, h
```

Line 15 to line 23 encodes the exponent bits and the fractional bits when the number of remaining bits are more than sufficient. Line 19 appends zero bits to make up the total number of bits. Line 20 to line 23 performs posit rounding based on the rounding information. Line 21 computes rounding for a tie breaking condition. It rounds up only odd posit bits as that will give even posit bits, and it never rounds beyond maxpos. Rounding beyond maxpos would produce the bit pattern representing $\pm\infty$.

Line 25 and line 26 compute the number of exponent bits and the number of fractional bits to be truncated. Line 27 to line 36 computes the total amount of truncation incurred in the represented value due to the truncated exponent bits or fractional bits. The tie breaking points are computed based on the midpoint of two posit numbers that can be represented in the posit configuration. Line 37 to line 42 encode the exponent bits and the fractional bits with truncations if any. Line 43 to line 46 compute the rounding similar to the case before, but determine the tie breaking condition with the truncation value and the tie value.

Algorithm 3 encodes any posit numbers. The algorithm first checks and encodes the special posit numbers 0 and $\pm\infty$. Then, it uses the Algorithm 2 to obtain the bit pattern of the absolute posit number. If the posit number is a negative number, it applies two's complement operation to the bit pattern.

7 SOFTWARE TESTS AND VERIFICATIONS

Extensive software testing is the key to ensuring software robustness. Software development such as agile development, test-driven development, or lean software development are common practices

Algorithm 2: EncodeAbsPosit: Encode absolute posit

```
input  : A posit number $P = (-1)^s 2^{2^{es}k} 2^e (1 + f/2^h)$,
         $P \notin \{0, \pm\infty\}$, configuration (N, es), and rounding
         information R
output : The posit bits of |P|
maxp ← $2^{N-1} - 1$
n ← $N - 1$ // Number of remaining bits.
bits ← 0; rounded ← FALSE
if $k \geq 0$ then
    if n ≥ $k + 1$ then
        bits ← Mask($k + 1$); n ← n − ($k + 1$)
        if n > 0 then bits ← bits ≪ 1; n ← n − 1
    else
        bits ← Mask(n); n ← 0; rounded ← TRUE
else
    bits ← 1
    if n < $-k + 1$ then rounded ← TRUE
    n ← n − min(n, $-k + 1$)
if rounded ← FALSE then
    if n ≥ $es + h$ then
        bits ← (bits ≪ es) | e
        bits ← (bits ≪ h) | f
        n ← n − (es + h)
        bits ← bits ≪ n; n ← 0
        if R is tie breaking case then
            if bits & 1 AND bits < maxp then bits ← bits + 1
        else if R is rounding up case then
            if bits < maxp then bits ← bits + 1
    else
        ntebits ← max(0, es − n)
        ntfbits ← max(0, h − max(0, n − es))
        if ntebits > 0 then
            te ← (e ≫ ntebits) ≪ ntebits
            value ← $e \times 2^h + f$
            representedValue ← te × $2^h$
            truncation ← value − representedValue
            tie ← $2^{\text{ntebits}+h-1}$
        else if ntfbits > 0 then
            tf ← (f ≫ ntfbits) ≪ ntfbits
            truncation ← f − tf
            tie ← $2^{\text{ntfbits}-1}$
        w ← es − ntebits
        te ← e ≫ ntebits
        bits ← (bits ≪ w) | te; n ← n − w
        w ← h − ntfbits
        tf ← f ≫ ntfbits
        bits ← (bits ≪ w) | tf; n ← n − w
        if truncation == tie then
            if bits & 1 AND bits < maxp then bits ← bits + 1
        else if truncation > tie then
            if bits < maxp then bits ← bits + 1
return bits
```

Algorithm 3: EncodePosit: Encode posit to binary

```
input  : A posit number $P = (-1)^s 2^{2^{es}k} 2^e (1 + f/2^h)$, Type(P),
         configuration (N, es), and rounding information R
output : The posit bits of P
if Type(P) is zero then return 0
if Type(P) is ±∞ then return 1 ≪ (N − 1)
bits ← EncodeAbsPosit(P, N, es, R)
if s == 1 then
    bits ← −bits & Mask(N)
return bits
```

in the industry. Automated testing is a crucial component of software development.

In this section, we proposed several complementary methods to test the results of the posit arithmetic operations for different types of posit numbers and requirements.

7.1 Tests for special posit numbers

The special posit numbers 0 and $\pm\infty$ appear only in a small number of cases as depicted in Table 2. Tests with all the combinations of positive numbers, negative numbers, 0, and $\pm\infty$ for both input numbers to the arithmetic operations can be executed. The results can be checked by the posit number types based on Table 2. When an arithmetic operation on two normal posit numbers returns a zero, the arithmetic operation must be an addition with inputs $a = -b$, or a subtraction with inputs $a = b$.

Negative tests on normal posit numbers can be performed to ensure the arithmetic operations do not return 0 or $\pm\infty$, except for the cases with additions and subtractions with specific inputs.

For faster turnaround during software development, a small set of positive and negative numbers may be selected as the normal posit numbers to test on.

7.2 Tests for normal posit numbers

A small set of normal posit numbers may be hand-crafted to test arithmetic operations. Computed results are compared with the reference answers. This is practical only when the set is small.

We observe that posit numbers are strictly increasing when you add one to the bit patterns, and are strictly decreasing when you subtract the bit patterns by one, until the $\pm\infty$ bit pattern is encountered. The posit numbers with bit patterns, $bits \pm 1 \equiv P^{\pm}$ except 0 and $\pm\infty$, are the nearest normal posit numbers to the posit number with bit pattern, $bits \equiv P$.

Posit numbers, and the results of additions, subtractions, multiplications, and divisions, can be represented exactly using rational numbers. When a reliable posit decoding module is available, given two decoded posit numbers, we can construct the equivalent rational numbers with the decoded information and the posit formulation. Then, we test the arithmetic operation using both the posit numbers and the rational numbers. The result, V, computed by the rational numbers is an accurate reference answer. With the posit arithmetic computed result, P, we first verify that the posit number P and the value V have the same sign. Then, we compute both the absolute difference and the error ratio with respect to the reference answer using rational numbers,

$$\text{Absolute difference} = |P - V|$$

$$\text{Error ratio} = \frac{\max(P, V)}{\min(P, V)}$$

In addition, we construct the rational numbers from the bit patterns P^+ and P^- excluding the special numbers 0 and $\pm\infty$. Using these two posit numbers, compute both the absolute differences and the error ratios the same way with respect to the reference V. We consider if P^+ is a closer answer to the reference V. Using one of the numbers P and P^+ that is closer to zero, we examine if extending one more bit will be a regime bit, an exponent bit, or a fractional bit. If the extended bit is a regime bit or an exponent bit, we compare P and P^+ using the error ratios. The correct answer should have the lowest error ratio. If the extended bit is a fractional bit, we compare P and P^+ using the absolute differences. The correct answer should have the lowest absolute difference in this case. Similarly, we compare the number P and the number P^- using one of the metrics. When two different numbers have the same absolute difference or error ratio depending on the type of the extended bit, it must be a tie breaking condition, and the least significant bit can be checked for a zero bit.

Note that we are only testing with posit number inputs that are not expected to produce the special posit numbers using this method. The cases involved special posit numbers are covered in section 7.1.

7.3 Tests with checksum comparisons

The tests in section 7.2 can incur significant execution overhead with the use of rational numbers. For the development of a performance optimized posit arithmetic implementation, this may deem to be too slow to be productive. If we can deterministically and consistently generate the posit numbers to be tested, we can compute the checksum of the arithmetic operations once using a well-tested posit implementation. Subsequently, the checksum can be used to check if the new updates to the posit implementation produce different results. This also avoids producing and keeping huge reference answer sets for verification of a posit arithmetic implementation.

7.4 Tests on the posit encoding and decoding

Posit encoding and decoding is symmetry for all the bit patterns of a specific bit size. A number of selected posit numbers can be manually computed to check the encoding and decoding modules indeed produce the expected results. Then, all the bit patterns can be decoded, and subsequently be encoded, to check if the encoded bit patterns are the same as the bit patterns given to the decoding.

In addition, the decoded posit numbers can be used to check if they satisfy the property of posit numbers being strictly increasing and strictly decreasing with the $bits \pm 1$, except for the special posit number $\pm\infty$.

8 CONCLUSION

We showed the relationships between fixed-point numbers and posit numbers. We proved the formulations of the arithmetic operations. Posit binary encoding and decoding algorithms have been provided.

Since the big integer library is widely available for various computing platforms, and in view of the current state of the posit number software implementations, it makes sense to use the fixed-point approach with big integers to quickly bootstrap and implement a correct reference implementation.

We described our software test and verification methodology to ensure that it computes the posit arithmetic correctly by the posit design. The same methodology could be used to certify that existing posit implementations conform to the posit design. Using our Python implementation [3] of the posit arithmetic, we were able to find the incorrect results produced by other posit implementations.

In the future, posit implementations can be enhanced by extending the current work to derive posit arithmetic formulations from the posit parameters without explicitly converting to and from fixed-point numbers. Deriving the bounds on the minimum number of integer bits required for a specific posit configuration will be useful to determine if a native integer data type may be used.

Vectorizing the software computation is a big challenge, but potentially could unlock orders of magnitude more performance. The math library and BLAS [1] library are missing in current implementations. The development of these libraries will be of great value. Existing applications will then be more likely to be integrated with posit arithmetic and tested.

ACKNOWLEDGMENTS

The author would like to thank Dr. Hsu Wen-Jing for leading the way to exciting theoretical research work in Computer Science. His experience and guidance shaped the way for the author, striking a balance between theory and practice. The current work would not exist without his patience and persistence in mathematical writing.

Thanks to Prof. John L. Gustafson for his just-in-time offer to edit this paper. Also thanks for his effort in preparing the posit documentation with a great visual presentation.

Thanks to Dr. Vee Voon Yee for providing valuable feedback on the mathematical notations and writings. Thanks to Dr. Leong Siew Hoon for sharing her insights into the posit implementation and rounding issues.

REFERENCES

[1] 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (June 2002), 135–151. https://doi.org/10.1145/567806.567807
[2] Jorge Aparicio. 2017. A Rust implementation of the posit number system. (Aug 2017). Retrieved Feb 3, 2018 from https://github.com/japaric/posit
[3] Shin Yee Chung. 2018. Probably Correct Posit Prototype for Python. (Jan 2018). Retrieved Jan 14, 2018 from https://github.com/xman/sgpositpy
[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.
[5] David Flanagan and Yukihiro Matsumoto. 2008. *The Ruby Programming Language* (1st ed.). O'Reilly.
[6] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Software* 33, 2, Article 13 (Jun 2007). https://doi.org/10.1145/1236463.1236468
[7] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating Point Arithmetic. *Comput. Surveys* 23, 1 (1991), 5–48.
[8] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[9] Torbjörn Granlund. 2016. The GNU Multiple Precision Arithmetic Library. (Dec 2016). http://gmplib.org
[10] Clément Guérin. 2017. bfp - Beyond Floating Point. (Dec 2017). Retrieved Feb 3, 2018 from https://github.com/libcg/bfp

[11] John L. Gustafson. 2015. *The End of Error: Unum Computing.* CRC Press. https://books.google.com.sg/books?id=W2ThoAEACAAJ
[12] John L. Gustafson. 2017. Posit Arithmetic. (Oct 2017). Retrieved Feb 3, 2018 from https://posithub.org/docs/Posits4.pdf
[13] Siew Hoon Leong. 2017. Posit-Javascript. (Sep 2017). Retrieved Feb 3, 2018 from https://github.com/cerlane/posit-javascript
[14] Theodore Omtzigt. 2018. Universal Number Arithmetic. (Feb 2018). Retrieved Feb 3, 2018 from https://github.com/stillwater-sc/universal
[15] IEEE Task P754. 2008. *IEEE 754-2008, Standard for Floating-Point Arithmetic.* IEEE, New York, NY, USA. 58 pages. https://doi.org/10.1109/IEEESTD.2008.4610935
[16] Emanuele Ruffaldi. 2017. C++ Posit Implementation. (Dec 2017). Retrieved Feb 3, 2018 from https://github.com/eruffaldi/cppposit
[17] Guido van Rossum. 1995. *Python Reference Manual.* Technical Report. Amsterdam, The Netherlands, The Netherlands.
[18] Isaac Yonemoto. 2017. FastSigmoid. (Jul 2017). Retrieved Feb 3, 2018 from https://github.com/Etaphase/FastSigmoids.jl
[19] Isaac Yonemoto. 2018. Sigmoid Numbers for Julia. (Jan 2018). Retrieved Feb 3, 2018 from https://github.com/interplanetary-robot/SigmoidNumbers