Posit Arithmetic

Antoine Sébert
Department of Applied Mathematics and Computer Science
Danmarks Tekniske Universitet
Kongens Lyngby, Denmark
antoine.sb@orange.fr (ORCID)

***Abstract*—This electronic document is a "live" template and already defines the components of your paper [title, text, heads, etc.] in its style sheet. We show that the IEEE 754 standard is complex.**

Computer arithmetic, energy-efficient computing, floating point, posits, valid arithmetic

I. Introduction

The main idea of this paper is to provide an implementation of posits, a drop-in replacement for IEEE 754 floating point numbers [1]. Posits intent to provide an increased precision with faster computation at a lower cost, with a non-redundant representation.

Two other datatypes related to posits are also part of this new arithmetic: *valids*, representing ranges of real numbers, and *quires*, used to accurately represent results of computations. They will not be covered in the paper but are believed to be worth future investigation. As it seems, this computational model is quite complete, and addresses many of the problems that arise from the use of floats, and numbers in general.

Because of its novelty, there is no hardware support yet concerning general public products, but rather research-motivated configurable platforms such as FGPA [3] or electronic systems modelled using Verilog [4; 5]. Therefore, we propose a software implementation of posits to emulate their behaviour as a library. We choose to use the language Rust for its enforcement of correctness, low-level memory management and runtime speed. Since the upcoming standard is at the state of draft, we rely on publications of the author, assuming the format and ABI/API of the datatype will not significantly change and remain relevant for the next years.

In this paper, we:

- Show that IEEE 754 carries unnecessary complexity and improvable characteristics (section II)
- Present a new format that aims to represent not only floating-point values, but real numbers in general, with better semantics and correctness (section III)
- Compare accuracy and speed results of a simple implementation of posit8 against native float computations (section IV)

II. IEEE 754 format issues

We base our analysis of the standard on the last version, IEEE 754-2019. Unless indicated otherwise, all information in this section comes from this standard. If it has been widely used to represent floating-point numbers, and for good reasons [6], it is not exempt of design flaws.

A. Classes

Numbers in this standard belong into one of 10 classes: `signalingNaN`, `quietNaN`, `negativeInfinity`, `negativeNormal`, `negativeSubnormal`, `negativeZero`, `positiveZero`, `positiveSubnormal`, `positiveNormal`, `positiveInfinity`.

Although some classes share a degree of similarity, they have a different semantic and must be non-overlapping: a number belongs to one and only one class. We will not cover the details of those classes, but question whether they bring significant enough features to justify their existence.

For example, `quietNaN`, does not trigger an exception when encountered, while `signalingNaN` does; it would be simpler to move this logic at the operation level.

Concerning `negativeInfinity` and `positiveInfinity`, it allows to determine on which side or the range an infinite value was encountered, but the same cannot be said about `negativeZero` and `positiveZero`.

Finally, upon looking at the distinction between normal and subnormal numbers, its existence its existence is debatable. Subnormal numbers are not as precise as normalized numbers, which can be problem, considering that this characteristic is mostly invisible to the programmer. It is also worth noting that zero is neither normal nor subnormal.

B. Redundancy

If positive and negative zeroes can be useful in the context of functions limits, some others are more questionable: the number of *NaN* representations is equal to:

$$2^{23} * 2 = 16\ 777\ 214 \quad (1)$$

Roughly equivalent to 0.4% of all possible values, it could be replaced as a single or two fixed representations.

C. Representation errors

Some numbers, such as 0.1, do not have an exact representation, leading to errors in trivial operations:

$$0.1 + 0.2 \neq 0.3 \quad (2)$$

This can be a major issue regarding critical applications [7]. Of course, it is not possible to represent an infinite number of values using a finite set, but techniques to bypass this issue can be used, for example adding a data type those purpose is to store results of operations. It also appears that this aspect has not been extensively addressed in the literature, therefore possible convenient solutions are yet to be discovered

D. Rounding modes

IEEE 754 defines rounding-direction attributes:

- `roundTiesToEven`
- `roundTiesToAway`
- `roundTowardPositive` (ceiling)

- roundTowardNegative (floor)
- roundTowardZero (truncation)

Some other attributes may also have an impact on the rounding process. The rounding-direction attribute used is dependent on whether the operation inputs or outputs floats, unless either one of the operands or the result is *NaN*; in that case there is no rounding.

III. Arithmetic

We base our analysis of the current standard documentation draft [2] and the papers published by the author and collaborators [8, 9]. Unless indicated otherwise, all information in this section comes from these documents.

A. Format

Posits, also known as "type III Unum", are composed of four parts, in this order:

- Sign: The value +1 for positive numbers, -1 for negative numbers
- Regime: A subfield of a posit consisting of some number of identical bits terminated by the opposite bit or the end of the number, that contributes to the specification of the power-of-two scaling of the fraction
- Exponent: The part of the power-of-two scaling determined by the exponent bits
- Fraction: The component of a posit containing its significant binary digits after the binary point; $0 \leq fraction < 1$

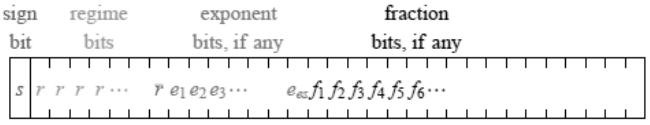

Fig. 1. Format of a generic *posit*.

The sizes of the Exponent and fraction depends on the Regime, which does not have a determined size. The overall size, *nbits*, is flexible, and specifies the precision. Four standard precisions are described: *posit8*, *posit16*, *posit32* and *posit64* (*posit32* is believed enough for most cases). From *nbits* it is possible to compute the smallest positive real number *minpos* and the largest positive real number *maxpos* that can be represented:

$$minpos = 2^{1/8 * nbits -(2 - nbits)} \quad (3)$$

$$maxpos = 2^{1/8 * nbits -(nbits - 2)} \quad (4)$$

Concerning the maximum range *m*, it is given by:

$$2_p < m < 2_p \quad (5)$$

Where *p* the maximum number of significand digits given by:

$$p = nbits - log(nbits) + 1 \quad (6)$$

Therefore, we can compute the value of a posit *x* with the following:

$$x = (-1)^b f 2^{e+k2^{\wedge}es} \quad (7)$$

With *b* the sign bit, *f* the fraction, *e* the exponent, *es* the maximum exponent size and *k* equals to *-m* or *m – 1*, depending on the regime's value.

B. Rounding and Operations

The only rounding mode available *is round to nearest, tie to nearest even*, as the default IEEE 754 rounding mode for floats.

Operations required to make this data type part of an arithmetic are also defined: *addition*, *subtraction*, *multiplication*, *division*, but also *negate*, *abs* and *round*; as well as trigonometric functions. The standard also enforces more complex functions such as *sqrt*, *exp* and *pow*. Moreover, posits support total ordering.

As a result, we can see posits as serious candidates for any real-life problem solving. Of course, with the hardware support being almost non-existent at the time, implementations using software emulation will certainly not be able to compete with IEEE 754 floats, but it is not an intrinsic characteristic of posits.

C. Valids

To summarize, valids are two posits of equal size bounding a range of real numbers, to which are appended an uncertainty bit *ubit*. Therefore, we have $[a, b]$ where *a* and *b* are floats, and $a \leq b$. The resulting interval can be open, closed or half-opened and, since *valids* operate on the possible representations of the posits format they are composed of, encompass ∞, such that $[x, \infty[\cup]-\infty, y]$ is a correct *valid*.

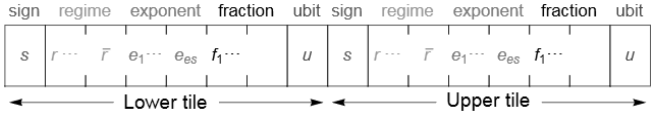

Fig. 2. Format of a *valid*.

D. Quires

Concerning *quires*, their purpose is to hold temporary values produced by mathematically exact distributive and associative operations. Hence, *quires* are not subject to rounding until the very end of a series of computations, when they are destroyed to recover the result as a *posit*. The size of a quire is given by:

$$nbits^2 / 2 \quad (8)$$

With *nbits* the size of the corresponding *posit*. Therefore, for the standard *posits* posit8, posit16, posit32 and posit64, we have the corresponding sizes 32, 128, 512, 2048.

IV. Results

Our implementation of posits is a software emulation in the form of a library, written in Rust. It is a limited attempt that aims to provide support for posit8 and its corresponding quire.

The main inconvenient is indeed that the library cannot use dedicated registers and processor instructions, letting alone running in kernel space. A solution to the lack of hardware support that prevents fair benchmarks might be testing a posit library against a float library.

V. Related works

There have been attempts to solve the issues related to IEEE 754 rather than creating concurrent standards. Transfloat arithmetic [10] removes the significant number of NaN representations mentioned in *section II* using transreal arithmetic [11]. Minifloat [12] trade correctness and range for speed and are suited for particular applications. While not being part of IEEE 754, they follow its principles, making minifloats a lower-resolution version of traditional IEEE floats. It is arguable that this format brings significant advantage besides its size and speed and could eventually be replaced by one of the standards posit data types. We can also mention Brain Floating Point (bfloat16) [13], a modified Half-precision float (binary16), targeted at machine learning [14]. While its dynamic range is like binary32, the precision is reduced.

Several formats are also competing with IEEE 754, sometimes as a domain-related solution for specific applications. To that extend, IBM mainframes support hexadecimal floating point (HFP), whose exponent is smaller, and fraction is larger, hence allowing a greater range at the cost of a lesser precision [15].

We also studied already existing implementations of posit arithmetic. A list of the indexed contributions can be found on the dedicated website [16, 17], encompassing both hardware and software works.

On the software side, we notice that integration in widely used frameworks, such as TensorFlow [18] and NumPy [19] has already made ground for further adoption in the field. As such, the ecosystem of the language Python seems to already contain several posit arithmetic libraries [20, 21, 22], suggesting that academic research and machine learning applications lead the current efforts. Next in order would be C/C++ [23, 24, 25], languages where memory is directly managed by the programmer. Libraries in these languages seems also to attract a larger public. Closer to our work, we find Rust implementations [26, 27]. While having a greater scope, they do not cover the entirety of the standard, thus an exhaustive implementation is yet to be written. One explication to this situation might the youth of the Rust language and thus its ecosystem. Other languages, such as C#, Julia, or JavaScript, have been used to write implementations of posits.

Hardware support mainly relies on FGPA [28] implementations and Verilog/HDL [29, 30]. There is no vendor support yet, but an increase of demand for such a format, if such demand would exist, is very likely to lead to its adoption by processor and microcontrollers manufacturers.

VI. Conclusions & future works

As shown in section III, posit arithmetic has advantages over IEEE 754 that makes it a good candidate for drop-in replacement, as its author intends it. This mainly comes from the dozens of years have been available to uncover and analyse the design flaws of IEEE 754, and investigate potential solutions.

While it makes little doubt that domain-specific formats will continue to exist and maybe appear with the rise of new computation needs, IEEE 754 should not be considered as a dead format, as the recent enhancements proves it [1]. It is likely to evolve again in the future, although its large adoption prevents breaking changes. Maybe that is, more than the current issues, its major drawback.

References

[1] "IEEE Standard for Floating-Point Arithmetic", in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019 doi: 10.1109/IEEESTD.2019.8766229

[2] Posit Working Group. "Posit Standard DocumentationRelease 3.2-draft", June 2018 [UNPUBLISHED]

[3] https://github.com/ChenJianyunp/Posit32-2-exact-multiply-accumulator

[4] JOHNSON, Jeff. Rethinking floating point for deep learning. *arXiv preprint arXiv:1811.01721*, 2018.

[5] https://github.com/manish-kj/PACoGen

[6] No evidence to support that claim could be found

[7] ZHIVICH, Michael et CUNNINGHAM, Robert K. The real cost of software errors. *IEEE Security & Privacy*, 2009, vol. 7, no 2, p. 87-90.

[8] GUSTAFSON, John L. et YONEMOTO, Isaac T. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 2017, vol. 4, no 2, p. 71-86.

[9] GUSTAFSON, John L. Posit Arithmetic. Online: https://posithub. org/docs/Posits4. pdf, 2017.

[10] ANDERSON, James ADW. Trans-floating-point arithmetic removes nine quadrillion redundancies from 64-bit ieee 754 floating-point arithmetic. 2014.

[11] ANDERSON, James ADW, VÖLKER, Norbert, et ADAMS, Andrew A. Perspex machine viii: Axioms of transreal arithmetic. In : *Vision Geometry XV*. International Society for Optics and Photonics, 2007. p. 649902.

[12] CERMELLI, C. A., RODDIER, D. G., BUSSO, C. C., *et al.* MINIFLOAT: A novel concept of minimal floating platform for marginal field development. In : *The Fourteenth International Offshore and Polar Engineering Conference*. International Society of Offshore and Polar Engineers, 2004.

[13] "BFLOATI6 - Hardware Numerics Definition White Paper", *Intel*, November 2018.

[14] BURGESS, Neil, MILANOVIC, Jelena, STEPHENS, Nigel, *et al.* Bfloat16 Processing for Neural Networks. In : *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019. p. 88-91.

[15] GERWIG, Guenter, WETTER, Holger, SCHWARZ, Eric M., *et al.* The IBM eServer z990 floating-point unit. *IBM journal of Research and Development*, 2004, vol. 48, no 3.4, p. 311-322.

[16] https://posithub.org/khub_community

[17] https://posithub.org/docs/PDS/PositEffortsSurvey.html

[18] https://github.com/xman/tensorflow/tree/posit

[19] https://github.com/xman/numpy-posit

[20] https://gitlab.com/cerlane/SoftPosit-Python

[21] https://pypi.org/project/sfpy/

[22] https://github.com/mightymercado/PySigmoid

[23] https://github.com/stillwater-sc/universal

[24] https://github.com/eruffaldi/cppposit

[25] https://github.com/libcg/bfp

[26] https://gitlab.com/burrbull/softposit-rs

[27] https://gitlab.com/cerlane/SoftPosit

[28] https://github.com/ChenJianyunp/Posit32-2-exact-multiply-accumulator

[29] https://github.com/manish-kj/Posit-HDL-Arithmetic

[30] JOHNSON, Jeff. Making floating point math highly efficient for AI hardware. Facebook Engineering, 2018.