Hardware cost evaluation of the posit number system

Luc Forget, Yohann Uguen, Florent de Dinechin

Compas'2019 : Parallélisme/ Architecture / Système
LIUPPA - IUT de Bayonne, France, du 24 au 28 juin 2019

Hardware cost evaluation of the posit number system

Luc Forget and Yohann Uguen and Florent de Dinechin

Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France
(first).(last-name)@insa-lyon.fr

Abstract
The posit number system is proposed as a replacement of IEEE floats. It encodes floating-point values with tapered precision: numbers whose exponent is close to 0 have more precision than IEEE floats, while numbers with high-magnitude exponents have lower precision, because their encoding takes bits from the significand. In addition, the posit standard mandates the presence of the "quire", a Kulisch-like large accumulator able to perform exact sums of products. Several works have demonstrated that posit arithmetic can provide improved accuracy at the application level. However, the variable-length fields of posit encoding impacts the hardware cost of posit arithmetic. Existing comparisons of posit hardware versus float hardware are unconvincing, and the overhead of the exact accumulator has not been studied in detail so far. This work aims at filling this gap. To this purpose, it introduces an open-source tool to compare the respective costs of floats and posits on an application basis. A C++ templatized library compatible with Vivado HLS implements operators for custom size posits and their associated quire. These architectures are evaluated on recent FPGA hardware and compared to their IEEE-754 counterpart. The standard 32 bits posit adder is found to be twice as large as the corresponding floating-point adder. Posit multiplication requires about 7 times more LUTs and a few more DSPs for a latency which is 2x worst than the IEEE-754 32 bit multiplier. Furthermore, the cost of the posit 32 quire is shown to be the same as a 32 bits floating-point Kulisch accumulator.

1. Introduction

Most implementations of real number arithmetic rely on underlying floating-point units. The ease-of-use of the floating-point format made its popularity. However, this hides complex hardware defined by the IEEE-754 standard [1].
The posit number system (described in details in [9]) aims at replacing the IEEE-754 floating-point representation. The first posit claim is that a part of the floating-point bits are lost to encode exponent bits. Indeed, when the exponent only requires a few bits of encoding, the rest of the bits could be used to extend the precision. The second claim is that there is no floating-point tool to avoid cancellations. To that end, the posit standard [8] requires the use of a quire, a variant of the Kulisch accumulator [13] revamped to fit the posit number system.
Most of the evaluation on posits is performed from a functional standpoint through software simulation [5]. The C/C++ SoftPosit library [1] (among others [2]) implements the latest posit stan-

[1] `gitlab.com/cerlane/SoftPosit`
[2] `posithub.org/docs/PDS/PositEffortsSurvey.html` as of march 6, 2019

dard and allows for direct accuracy comparison with floating-points. However the hardware cost is not yet completely known as none of the known projects implements an evaluable quire. Most advanced projects generates custom posit adders and multipliers in HDL [3] or using Intel OpenCL SDK complianttemplatized C++ operators [15]. Application specific hardware implementations of posits include machine learning [10, 12] or a matrix multiply algorithm [4]. This work enables a complete hardware cost evaluation of the posit number system, including the quire. A Vivado HLS compliant templatized C++ library with custom size posits is provided. The current implementation allows for standalone posit operators such as the adder, the subtracter and the multiplier. The quire can receive exact posit products and perform their addition or subtraction. The library is extendable to other operators and built on a custom internal representation. Section 2 provides details on the architectures used in the library.
The benefits of the posit 8 and 16 bits posits are already recognized [5]. Indeed, for such small formats, the Kulisch-like accumulator is very cheap [2] and the precision is increased over standard floating-point. However, the posit 64 quire is so expensive it is rarely even considered. The focus of this study is then to evaluate the cost and benefits of replacing 32 bits floats by posits with a quire. Section 3 shows synthesis results of standalone operators compared to Vivado's floating-point IPs and a templatised soft floating-point operators library (without support for subnormals) [16]. It also evaluates the quire against IEEE floating-point accumulation loops and custom floating-point Kulisch accumulators.

2. Architecture

In order to perform computations on a posit number, one must beforehand decode it to an internal format. As posits are stored using a two's complement representation, the format we chose is a custom two's complement floating-point representation. Such a value is named a *posit value* and is obtained by *decoding* a posit. Similarly, a *posit value* can be *encoded* back to a posit. A first part describes in details the fields of a *posit value* and explains the choices made. Then, the architectures of the different components that we built to evaluate posits are described. Each component has been validated using the C/C++ SoftPosit library.

2.1. A custom internal format: *posit value*

A *posit value* is a custom floating point format used to represent the posit-encoded number with fixed size fields. The significand is stored in two's complement so the decoding overhead from posit encoding is limited. The exponent is stored as the offset from posit minimum exponent to simplify arithmetic operators architectures. This is similar to the biased exponent of IEEE floats. Due to two's complement representation, negative power of two are expressed as -2×2^{k} whereas positive ones are expressed as 1×2^{k}. Thus the minimal exponent is $-(N-2)\times2^{W_{es}}-1$ which is the exponent of the smallest posit. The bias is the opposite of this value. The width of the significand is $W_f = N-(W_{es}+3)$, the significand size of the most precise posits. W_{es} is the exponent width of the considered posit format, which for our architectures is defined as $\log_2\left(\frac{N}{8}\right)$ following the posit standard [8]. As the maximum exponent of a posit is $N\times2^{W_{es}}$, $W_e = \log_2(N)+W_{es}+1$ bits are needed to represent all possible exponents. Finally, five extra bits are used in the *posit value* representation. The isNaR bit is used tosignal Not a Real posit value. S and I bits encode the sign and the implicit bit respectively. Lastly, the sticky and guard bits are used to avoid double rounding before conversion back to posit.
Posit decoder to *posit value*: The posit decoder used is described in Figure 1. The expensive part of this architecture comes from: the `OR reduce` over N-1 bits to detect NaR numbers; and the leading zero or one count (`LZOC + Shift`) that consumes the regime bits while aligning

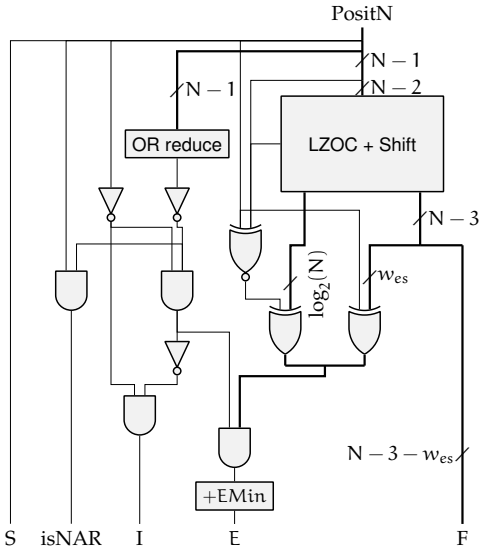

Figure 1: Architecture of a posit decoder.

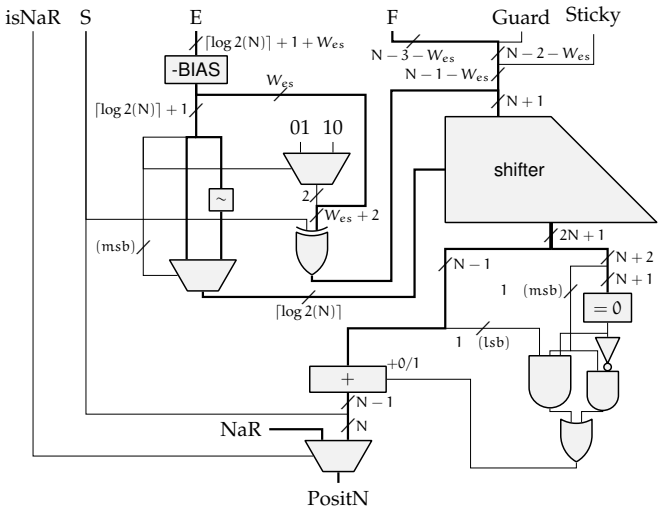

Figure 2: Architecture of a posit encoder.

the significand. The `+EMin` aligns the exponents to simplify following operators. This decoding cannot be compared to an IEEE floating-point equivalent as no decoding is needed.

Posit encoder from *posit value*: Due to the encoding properties of the posits, the rounding can only happen during this final conversion. Indeed, the guard and sticky bits are relative to the number of bits available to encode the posit. Therefore, the encoder (depicted in Figure 2) embeds quite some logic.

The fraction is first shifted to include the regime bits and es. Once shifted, the first $N-1$ bits represent the unrounded posit without sign. The remaining bits of the shitfed fraction is used to extract the guard bit and compute the sticky bit. At that point, the rounding can occur. Finaly, we chose between the computed value and NaR.

2.2. Posit adder/subtracter and multiplier

The architectures of the exact posit adder/subtracter (Figure 3 (top)) and multiplier (Figure 4 (top)) implement a custom floating-point operation. What realy differs from the IEEE is the rounding part (bottom part of the two previous Figures). In our case, the exact result is converted to a *posit value* that will later be encoded to a posit. For both operators, the exact significand must be realigned, correcting the exponent accordingly. This costs a leading zero/one counter (`LZOC + Shift`) that is the size of the exact significand. The guard and sticky bits are also computed as the posit standard requires round to the nearest binary value. This means that the remaining bits of the exact significand are compared to zero to compute the sticky. Additionnaly the computation the NaR bit is basically free.

2.3. Quire

The posit quire is able to perform exact sums and sums of products. Therefore, the input format of the quire is defined as the output of the exact multiplier from Figure 4 (top). To perform sums in the quire, a *posit value* must be converted to this exact multiplier format (not shown here).

The quire, as defined by the standard specifies NaR as a special value. Instead this work proposes to add a contaminant special bit that signals that the value hold in the quire is NaR. This

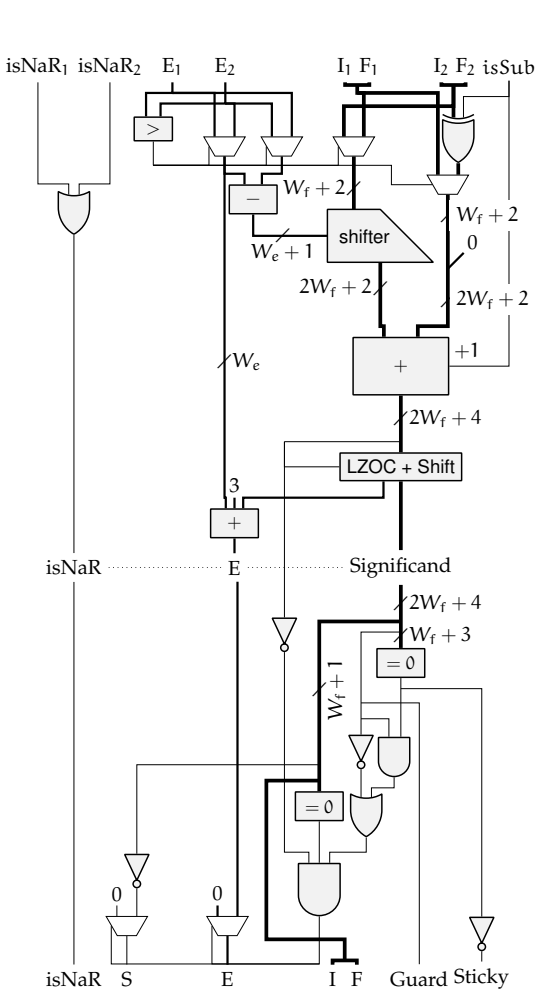

Figure 3: Architecture of a posit adder.

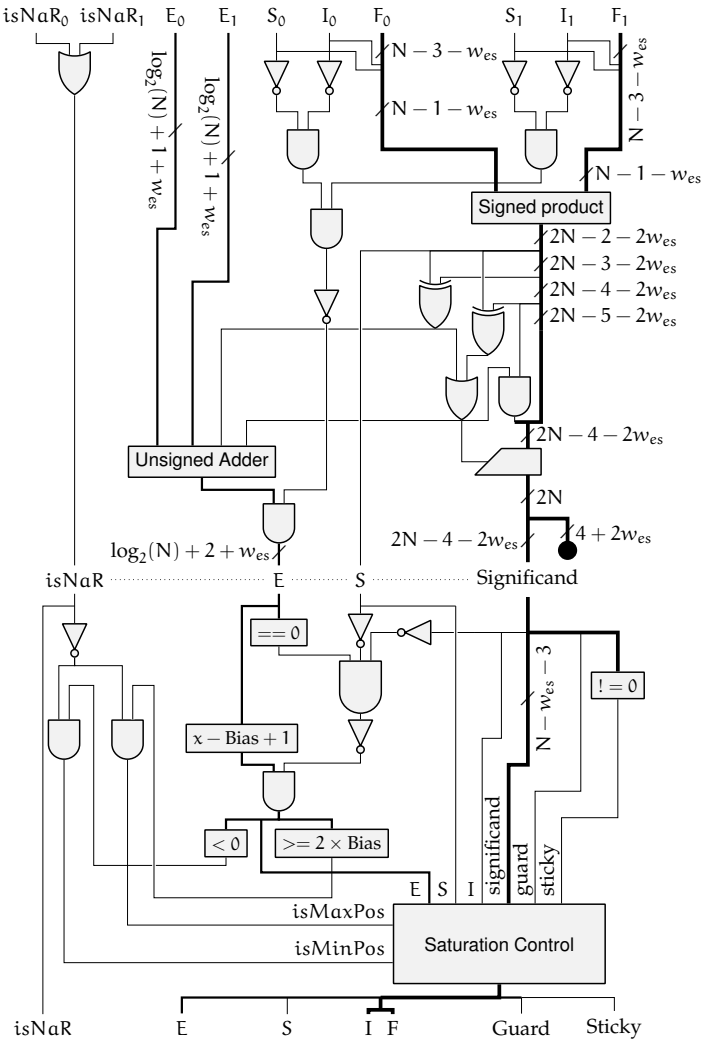

Figure 4: Architecture of a posit multiplier.

bit is set when NaR is added to the quire and stays set until the end of the computation. The standard motivates that the quire size should fit in a cache line. Therefore this extra bit can replace one carry bit. A more expensive alternative is to force NaR value when exporting the quire to memory. The implemented quire architecture is depicted in Figure 6.

2.3.1. Addition of products to the quire

The simplest implementation of the quire addition/subtraction is depicted in Figure 6 where the quire structure is as Figure 5. An exact posit product fraction is shifted to the correct place to the quire format according to its exponent. A large adder then performs the addition with the previous quire value. The subtraction is performed at very little cost using the same method as in the posit adder/subtracter.

The long carry propagation delay of the addition in this architecture will restict the maximum frequency achievable. Therefore, one may implement a segmented quire [17]. However, doing so, recovering the data hold by the quire will take several cycles. Indeed, the carries must be

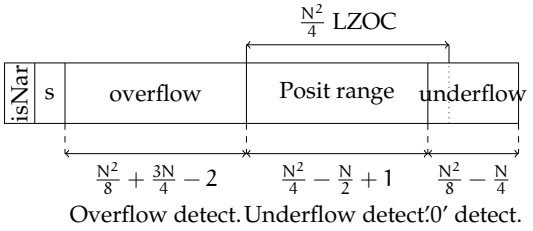

Figure 5: Quire conversion to *posit value*.

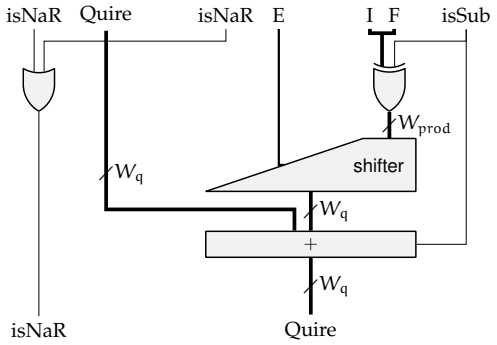

Figure 6: Architecture of a posit quire addition/subtraction.

Table 1: Synthesis results of posit and IEEE floating-point adders and multipliers.

		Adder/subtracter					Multiplier				
	Size	LUTs	Regs.	DSPs	Cycles	Delay	LUTs	Regs.	DSPs	Cycles	Delay
Float	32	341	467	0	9	2.529	80	193	3	7	2.201
	64	641	1098	0	11	2.562	196	636	11	17	2.568
Soft FP [16]	32	389	530	0	13	2.409	67	228	2	9	2.193
	64	1161	1544	0	19	2.712	259	651	9	10	3.299
Posit	16	436	394	0	17	2.357	269	292	1	16	2.361
	32	1044	1031	0	22	2.587	544	710	4	21	2.421
	64	2248	3383	0	38	2.854	1501	2410	16	42	2.816

propagated throught the different segments of the accumulator.
Our implementation converts the quire to a *posit value* (not shown here) before encoding to a posit. However, it requires a large LZOC of $\frac{N^2}{4}$ bits and two large XOR reduce to detect overflows and underflows/compute the sticky bit as shown in Figure 5.

3. Evaluation

The hardware cost evaluation of the posit number system is divided in two parts. We first evaluate standalone posit operators that are by nature more expensive than IEEE floating-points', trading precision bits around one. In order to validate the efficiency of our posit designs, we cross validated them against [3]. Using the same FPGA target, for the posit 32 adder, our architecture needs 948 LUTs and 39ns when they report 981 LUTs ans 40ns. Regarding the posit 32 multiplier, our architecture needs 479 LUTs + 4 DSPs with a 28ns delay where they need 572 LUTs + 4 DSPs and a delay of 33ns.
Then, we discuss the cost of the quire, and its floating-point equivalent, the Kulisch accumulator. All synthesis results given in this work are obtained using Vivado HLS and Vivado 2018.3 targeting 3ns delay for a Kintex 7 FPGA (xc7k160tfbg484-1).
Synthesis results for the posit adder/subtracter and posit multiplier are given in Table 1. To compare these results, we provide Vivado's floating-point IP synthesis. We also provide the

Table 2: Synthesis results for a sum of 1000 products using the quire, a taylored floating-point kulisch accumulator and regular floating-point hardware.

		LUTs	Regs.	DSPs	Cycles	Delay(ns)
Posit 16 quire	Unsegmented	1409	1763	1	1028	3.215
	Segment 32	1239	1431	1	1031	2.643
	Segment 64	1185	1555	1	1030	2.756
Posit 32 quire (512 bits)	Unsegmented	5068	6256	4	1040	8.850
	Segment 32	4394	4779	4	1055	2.854
	Segment 64	3783	4564	4	1047	2.961
Kulisch Float 32 (559 bits)	Segment 32	4446	5290	2	1050	2.875
	Segment 64	4365	5276	2	1041	2.854
Float 32		460	806	3	10011	2.676
Float 64		892	1999	11	12021	2.737

synthesis of the templatised soft floating-point operators (without support for subnormals).
The posit version of the adder requires more than 2x ressources and needs 2x the latency of Vivado's IP for similar frequencies. As expected, for every format, the posit variant of the multiplier is more expensive and has a greater latency. We observe about x7 more resources needed and more DSPs for about a x2 latency at similar frequencies than both Vivado's IPs and the soft floating-point library.
The synthesis results for the quire are given in Table 2 where we perform 1000 sums of product and return the result as a posit. Each posit quire is presented in its unsegmented version along with two segmented segmented versions (32 and 64 bits). One can observe that the unsegmented 16 and 32 bits posit quire are not able to achieve 3ns due to the long carry propagation. Therefore, we focus on the segmented version of the posit 32 in comparison with the Kulisch accumulator for 32 bits floats. The Kulisch accumulator used in this paper is from [17] to which we added the same rounding policy as posits. The implementation has been validated against the C/C++ MFPR library [7]. Synthesis results of floating-point sums of products are also given even though the non associativity of floating-points makes their latency much higher.
The posit 32 quire and the Kulisch for 32 bits floats costs are almost identical and can achieve similar frequencies. This comes from the fact that the posit decoding cost is comparable to the floating-point subnormals handling.

4. Conclusion

This work evaluates the cost of the hardware to support posits as a replacement for floating-point and evaluates the overall benefits of posits. To that end, a Vivado HLS templatized C++ library implements the posit number system, including the quire.
Should standard floating-point be replaced with posits? The accuracy win in many situations should not hide the loss of some properties at the core of classical error analysis [5], which requires further studies. In this work, we also find that standalone posits operators are considerably larger and slower.
Still, posits are a good alternative for small formats such as 8 or 16 bits. This study also raises again the popular question of including a Kulisch accumulator in modern processors [11, 14, 6].

Bibliographie

1. *IEEE standard for binary floating-point arithmetic*. – Institute of Electrical and Electronics Engineers. Note: Standard 754–1985.
2. Brunie (N.). – Modified fused multiply and add for exact low precision product accumulation. – In *IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pp. 106–113. IEEE, 2017.
3. Chaurasiya (R.), Gustafson (J.), Shrestha (R.), Neudorfer (J.), Nambiar (S.), Niyogi (K.), Merchant (F.) et Leupers (R.). – Parameterized Posit Arithmetic Hardware Generator. – In *36th International Conference on Computer Design (ICCD)*, pp. 334–341. IEEE, 2018.
4. Chen (J.), Al-Ars (Z.) et Hofstee (H.). – A matrix-multiply unit for posits in reconfigurable logic leveraging (open)CAPI (online). – pp. 1–5, 03 2018.
5. De Dinechin (F.), Forget (L.), Muller (J.-M.) et Uguen (Y.). – Posits: the good, the bad and the ugly. 2019.
6. Fiolhais (L.) et Neto (H.). – An Efficient Exact Fused Dot Product Processor in FPGA. – In *28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 327–3273. IEEE, 2018.
7. Fousse (L.), Hanrot (G.), Lefèvre (V.), Pélissier (P.) et Zimmermann (P.). – MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, n2, 2007, p. 13.
8. Group (P. W.). – *Posit Standard Documentation*. – juin 2018. Release 3.2-draft.
9. Gustafson (J. L.) et Yonemoto (I. T.). – Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, vol. 4, n2, 2017, pp. 71–86.
10. Johnson (J.). – Rethinking floating point for deep learning. *arXiv preprint arXiv:1811.01721*, 2018.
11. Koenig (J.), Biancolin (D.), Bachrach (J.) et Asanovic (K.). – A hardware accelerator for computing an exact dot product. – In *IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pp. 114–121. IEEE, 2017.
12. Kudithipudi (D.). – Performance-efficiency trade-off of low-precision numerical formats in deep neural networks. – In *Conference for Next Generation Arithmetic (CoNGA)*, 2019.
13. Kulisch (U.). – *Computer arithmetic and validity: theory, implementation, and applications*. – Walter de Gruyter, 2013.
14. Lutz (D. R.) et Hinds (C. N.). – High-precision anchored accumulators for reproducible floating-point summation. – In *IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pp. 98–105. IEEE, 2017.
15. Podobas (A.) et Matsuoka (S.). – Hardware Implementation of POSITs and Their Application in FPGAs. – In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 138–145. IEEE, 2018.
16. Thomas (D.). – Templatised soft floating-point for high-level synthesis. – In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019.
17. Uguen (Y.) et De Dinechin (F.). – Design-space exploration for the Kulisch accumulator (Online). 2017.