Architecture Generator for Type-3 Unum Posit Adder/Subtractor

Manish Kumar Jaiswal[1], and Hayden K.-H So[2]
Dept. of EEE, The University of Hong Kong, Hong Kong;
Email: [1]manishkj@eee.hku.hk, [2]hso@eee.hku.hk

***Abstract*—This paper is aimed towards the hardware architecture aspect of a recently proposed *posit* number system under type-3 *unum* (universal number system). Here, an algorithmic flow for the posit addition/subtraction arithmetic is developed and its hardware architecture is designed. Compare to floating point, posit provides better dynamic range and accuracy over same word size, along with more accurate and exact arithmetic support. Posit format includes a run-time varying exponent component, provided by a combination of *regime-bits* (of run-time varying length) and *exponent-bits* (of size up to *ES* bits). Thus, the mantissa precision also varies at run-time. This provides a combination of dynamic range and precision under a given word size (N). This possible variation in format along dynamic range and precision may attract various applications with different(accuracy and dynamic range) requirement. However, this run-time variation in posit format also poses a hardware design challenge. So, this paper is aimed towards the construction of an open-source parameterized Verilog HDL (Hardware Description Language) generator for posit adder/subtractor arithmetic, with parameterized N and *ES*.**

***Keywords*-Unum, Posit, FPGA, Multi-Precision, Digital Arithmetic, Adder, Subtractor.**

I. Introduction

Unum (Universal number system) has been recently proposed by Gustafson et al. [1], [2], [3]. It is developed in a series of evolution as type-1 unum [1], [4], [5], type-2 unum [2], [6], and recently announced type-3 unum [3], [7]. The posit number system is proposed as type-3 unum. The unum proposal has created a significant amount of interest in the community. Unum is claimed to be developed as an alternative to the contemporary Floating point standard [8]. It is claimed to be providing various significant benefits over traditional Floating Point standard, including better dynamic range and accuracy over same bit field, more accurate and exact arithmetic computations. More details on these claims can be sought from above Unum references.

To the best of authors knowledge, no hardware validation is yet available for any format for unum. Thus, this paper is aimed towards the hardware generation of recently developed posit unum system. Currently, it is focused for addition/subtraction arithmetic.

Compare to type-1 and type-2 unum, posit is more closer to the floating point standard in terms of representation. However, it includes an extra field of *regime* bits along with exponent bits (of size *ES*), and both collectively contribute to the total effective exponent value. The *regime* bits size varies at run-time, which provides a run-time variation in exponent components as well as mantissa component positions. Due to this run-time variations, posit provides various options of dynamic range, and varying mantissa precision bits. As claimed and shown (with several example cases) by the developers, this kind of available choices would be beneficial for a variety of applications having a different set of requirements on dynamic range and accuracy. A brief detail on posit format is discussed in the next section.

The run-time variation in the posit format poses a key hardware design challenge for its packing and unpacking. The architecture requires a significant amount of dynamicity, and further, the inclusion of parameterization in its architectural modeling (along with other sub-components) includes another level of design challenge.

In view of above discussion, this paper is aimed for a hardware algorithmic flow for posit adder arithmetic and demonstrated its architectural implementation on FPGA as well as ASIC platforms. It is also aimed for an open-source Verilog HDL generator for posit adder arithmetic, which can generate Verilog HDL for desired word size (N) with desired exponent size (ES).

The main contributions of the present work can be summarized as follows:

- Proposed an algorithmic flow for hardware architecture of posit adder arithmetic.
- An open-source Verilog HDL generator is also modeled for it.
- Demonstrated the implementation details with 8-bit, 16-bit and 32-bit posit adder architecture with varying exponent size (ES).

II. Preliminary

A brief basic of posit is presented here which is directly based on the original posit literature [3]. An N-bit posit is defined by its exponent size (ES). The structure of posit format includes a Sign bit, Regime bits, Exponent bits, and Mantissa bits. Compare to floating point standard [8], posit includes an extra field as regime bits. The Sign bit in posit is 0 for positive numbers and 1 for negative numbers. For the case of negative number, first take 2's complement before decoding regime, exponent and mantissa bits.

$$\overbrace{s}^{Sign}\ \overbrace{r\ r\ r\ \cdots\ r\ r\ r\ \bar{r}}^{Regime\ bits}\ \overbrace{e_1\ e_2\ e_3\ \cdots\ e_{es}}^{Exponent\ bits,\ if\ any}\ \overbrace{f_1\ f_2\ f_3\ \cdots\cdots}^{Mantissa\ bits,\ if\ any} \tag{1}$$

Posit format has only one ZERO, represented by all bits with 0 value, and only one Infinity, represented by all, but Sign bit, with 0 value. It does not consider any NaN (not a number) representation. Also, posit does not consider sub-normal or de-normal representation, ie. all values are normalized numbers.

The regime bits are a binary string of either all 0 or all 1 terminated by an opposite bit. Regime bits can be of any length and its numerical value determined by the run length of the regime bits. With a string of m-bit 0 terminated by (m+1)th bit as 1 (-ve sequence), it gives a value of (-m) and with a string of m-bit 1 terminated by (m+1)th bit as 0 (+ve sequence), it gives a positive value of (m-1). Few examples, a regime bit sequence of 00001 gives a -4 value and a sequence of 11110 gives a value of +3. For a regime value of k, it contributes as $(2^{(2^{ES})})^k$, as part of total effective exponent value. The exponent bits are an unsigned integer with no BIASing, and it can be up to ES bits based on the availability at the right side of regime bits. With a value of e, it contributes as 2^e in total effective exponent. Mantissa bits function similar to the normalized floating point standard, and remaining bits (if available) after regime and exponent are occupied by it. Thus, with a regime value of k, exponent value of e and mantissa value of f (including hidden bit 1), the equivalent decimal value would be $s * (2^{(2^{ES})})^k * 2^e * f$, except the zero (000...000) and infinity (1000...000) representation. Few examples for decimal equivalent of 8-bit posit (P) are as follows (sign, regime, exponent and mantissa field are separated for easy understanding, and negative values are first converted into 2's complement before decoding). With ES=2: 0_0001_11_1 $= +(2^4)^{-3} \times 2^3 \times (1+\frac{1.0}{2})$, and 11101011 $\rightarrow$ 0_001_01_01 $= -(2^4)^{-2} \times 2^1 \times (1+\frac{1.0}{4})$, With ES=3: 0_110_101_1 $= +(2^8)^1 \times 2^5 \times (1+\frac{1.0}{2})$, and 10001111 $\rightarrow$ 0_1110_001 $= -(2^8)^2 \times 2^1 \times (1+0.0)$.

As seen in above discussion, for a given ES the mantissa size varies in posit representation. This variation in mantissa size with respect to ES is shown in Fig 1 for N=8 posit. It can be seen that posit representation founds more fraction bits around ± 1.

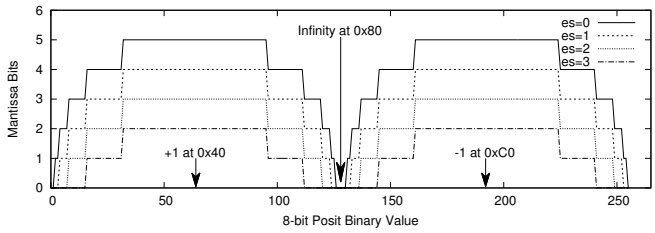

Fig. 1: Variation of Mantissa Size with respect to Exponent Size (ES) in 8-bit Posit format.

III. Proposed Posit Adder Algorithmic Flow and Architecture

The proposed parameterized algorithmic computational flow for posit addition is shown in algorithm-1. The major blocks in this include Data Extraction: which extracts the sign, regime-bit, exponent-bit, and mantissa information from the input operands; Core Adder Arithmetic Processing: which deals with the mantissa addition and final regime & exponent computation; followed by the Data Composition and Post-Processing: which combine sign, regime, exponent, mantissa, and perform rounding. This same computation flow can be used for posit subtraction also, after negating the second operand.

Algorithm 1 Proposed Adder Arithmetic Flow for Posit

1: **GIVEN:**
2: N: Posit Word Size
3: ES: Posit Exponent Field Size
4: RS: log_2 (N) (Posit Regime Value Store Space Bit Size)
5: **Input Operands:** $IN1$, $IN2$
6: **Data Extraction:Sign (S), Regime (R), Exponent (E), Mantissa (M), Exceptions (Infinity (Inf), Zero (Z))**
7: $Z \leftarrow Z1\&Z2$, ($Z1 = |IN1$, $Z2 = |IN2$, i.e. All bits of IN1, IN2 are 0)
8: $Inf \leftarrow Inf1|Inf2$, (All bits except MSB are 0)
9: Extraction from IN1:
10: S1$\leftarrow$ IN1[N-1]
11: $XIN1 \leftarrow S1$? $-IN1[N-2:0]$: $IN1[N-2:0]$
12: Regime Check (RC): ($RC1 \leftarrow XIN1[N-2]$, $RC2 \leftarrow XIN2[N-2]$)
13: Leading One Detection (LOD) of XIN1[N-2:0] ($\rightarrow K0$)
14: Leading Zero Detection (LZD) of XIN1[N-3:0] ($\rightarrow K1$)
15: Regime Value: $R1 \leftarrow RC1$? $K1$: $K0$
16: Regime Left Shift Amount: $Lshift \leftarrow RC1$? $K1+1$: $K0$
17: Left Shift XIN1 by Lshift amount $\rightarrow$ XIN1_tmp
18: E1 $\leftarrow$ MSB E-bits of XIN1_tmp
19: M1 $\leftarrow$ Remaining bits of XIN1_tmp (Append Hidden Bit as MSB)
20: Extraction from IN2: $\rightarrow$ S2, R2, E2, M2
21: **Core Adder Arithmetic Processing:**
22: Effective Operation: OP $\leftarrow$ S1 xor S2
23: Check for Large Operand and Small Operand
24: Large (L) Component: LS, LRC, LR, LE, and LM
25: Small (S) Component: SS, SRC, SR, SE, and SM
26: MANTISSA ADDITION:
27: Effective Exponent Difference (Ediff):
28: Ediff $\leftarrow$ ((LRC ? LR-(SRC ? SR : -SR) : SR-LR) « ES) +LE-SE
29: Right Shift SM by Ediff amount $\rightarrow$ SM_tmp
30: Add LM and SM_tmp:
31: Add_M $\rightarrow$ OP ? LM + SM_tmp : LM - SM_tmp
32: Mantissa Overflowed: Movf $\leftarrow$ Add_M[MSB]
33: Add_M $\leftarrow$ Movf ? Add_M : Add_M « 1
34: Normalization of Add_M:
35: LOD of Add_M $\leftarrow$ Nshift
36: Add_M $\leftarrow$ Add_M « Nshift, (Dynamic Left Shifting by Nshift)
37: Final EXPONENT (E_O) and REGIME (R_O) Computation:
38: LE_O $\leftarrow$ Combine LR, LE, Movf and Nshift
39: E_O: $\leftarrow$ Based on +/- of LE_O, Compute LSB ES bits from LE_O
40: R_O: $\leftarrow$ Based on +/- of LE_O, Compute MSB RS bits from LE_O
41: **Data Composition and Post-Processing:**
42: REGIME, EXPONENT and MANTISSA Packing:
43: REM $\leftarrow$ Append E_O at MSB of ADD_M
44: REM $\leftarrow$ Append REGIME bits at MSB, based on the value of R_O
45: REM $\leftarrow$ Shift Right by R_O
46: If (LS == 1): negate REM
47: Rounding
48: Final Output:
49: Combine LS with LSB (N-1) bit of rounded REM
50: Discharge Output while considering Exceptions

Based on the algorithm-1, a parameterized Verilog HDL is constructed which takes posit word size (N) and exponent size (ES) as its parameter and produces hardware for desired (N and ES) requirement. Since the regime bits can extend up to (N-1) bits, RS bits ($= Log_2 N$) can accommodate its maximum absolute numerical value. The proceeding architectural explanation refer to each related lines of algorithm-1.

A. Stage-1: Data-Extraction

Both operands are first checked for ZERO and Infinity value (Line 7 & 8 in algorithm-1). All bits with 0 leads to zero value, however, just with a true sign bit it leads to infinity.

The design challenge of posit data unpacking is solved here along lines 9-20 of algorithm-1 and proceeds as follows:

1) MSB of operands provides respective Sign bits (S1 and S2).

2) For true sign bit, operands undergo 2's complement conversion which produces XIN1 and XIN2, each of N-1 bits (except the respective sign bit).
3) The MSB of XINs depicts the sign of regime value and acts as Regime Check (RC) bit.
4) A leading zero detector (LZD) is employed to count a sequence of 1 with terminating 0 and a leading one detector (LOD) is used to count the sequence of 0 with terminating 1 (one less than the actual count of 1). RC determines either of K0 or K1 as R[RS-1:0] (absolute regime value) and regime left shift amount (Lshift) of respective operands.
5) To extract the exponent and mantissa, the respective XIN is dynamically left shifted by Lshift to push-out the entire regime bits and align exponent and mantissa at MSB. Now the MSB ES bit will act as the exponent and remaining bit would be mantissa bits.

The parameterized generation of LZD and LOD is based on their architecture shown in Fig. 2. Both are constructed in a similar hierarchical manner, except with a different respective basic 2:1 (LZD/LOD) building block, as shown in Fig. 2. For dynamic left shifting a parameterized barrel shifter is constructed with word width (W) and shifting amount (S) as the parameter. A barrel shifter requires one W-bit 2:1 MUX for each bit of S. So, here it requires RS numbers of 2:1 MUXs each of (N-1) bit size.

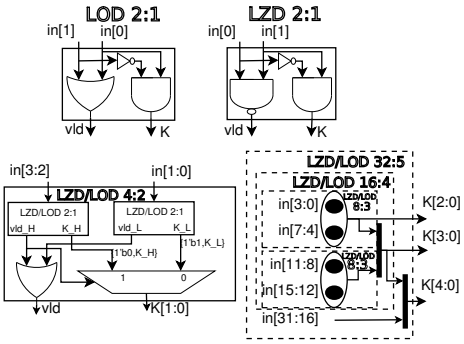

Fig. 2: LOD and LZD architectural design

B. Stage-2: Core Arithmetic

This stage involves the mantissa addition/subtraction, and final exponent and regime numerical value computation. The sign bits of both operands decide the effective operation (Line 22). A direct comparison of XIN1 and XIN2 gives the information of large and small operand. It requires an (N-1) greater-than-equal-to component. Based on this signal, the large and small components: the LS, SS; LRC, SRC; LR SR; LE, SE; and LM, SM are computed (Line 24-25), which require 2:1 MUXs of respective component bits.

To perform actual mantissa arithmetic both mantissa need to be aligned at the decimal point. For this, smaller mantissa is dynamically shifted right by the effective difference (Ediff) of large exponent and small exponent and produces SM_tmp (Line 29). For Ediff[BS:0] computation, first, an effective regime value difference (by taking their signs into account) is performed, which then shifted left by ES bits, and summed with exponent difference (Line 28) (this is based on the posit regime and exponent formation standard). Similar to the dynamic left shifter, a parameterized dynamic right shifter designed using barrel right shifter

The small shifted mantissa is then added/subtracted from large mantissa LM by using a N-bit add/sub unit and produces Add_M (Line 31). Add_M is checked for mantissa overflow (Movf) by checking its MSB and shifted 1-bit to left accordingly if found false, which requires an N-1 bit 2:1 MUX (Line 32-33). In case Movf is true, final effective exponent value needs to be incremented by one (Line 38).

In case LM and SM_tmp are two very close value and effectively undergone a subtraction operation, then Add_M may lose MSB bits, and it needs normalization. This is achieved by performing (N-ES) bit LOD operation on Add_M to get normalization shift (Nshift[RS-1:0]) amount (Line 35), and then perform dynamic left shifting of Add_M by Nshift amount (Line 36). The parameterized LOD and Dynamic left shift architecture is discussed above. Nshift amount is adjusted in final exponent output value (Line 38).

Under the computation of final exponent (E_O) and regime numerical value (R_O), first, the total effective large exponent output value (LE_O) is computed by combining LRC, LR, Movf, and Nshift. For LE_O < 0, it is negated as LE_ON. If LE_O is negative and LSB ES bits of LE_ON is non zero, then, E_O is computed as 2's complement of LSB ES bits of LE_ON, which is compensated by an increase in R_O, else LSB ES bits of LE_ON would become E_O. For R_O, if LE_O is positive (which will produce sequence of 1 with terminating 0 for regime bits) or LE_O is negative along with LSB ES bits of LE_ON is not zero, then, R_O would be an incremented value of LE_ON (after chopping off its LSB ES bits for exponent). These computations follows from (Line 37-40) and modeled as follows:

$$LE\_O = \{(LRC\ ?\ LR\ :\ -LR), LE\} + Movf - Nshift$$

$$LE\_ON = LE\_O[ES+RS]\ ?\ -LE\_O\ :\ LE\_O$$

$$E\_O = (LE\_O[ES+RS] \& (|LE\_ON[ES-1:0]))$$
$$?\ (1 << ES) - LE\_ON[ES-1:0]\ :\ LE\_ON[ES-1:0]$$

$$R\_O = !LE\_O[ES+RS] | (LE\_O[ES+RS] \& (|LE\_ON[ES-1:0]))$$
$$?\ LE\_ON[ES+RS-1:ES] + 1'b1\ :\ LE\_ON[ES+RS-1:ES]$$

C. Stage-3: Data Composition and Post-processing

This stage processes the another design challenge of posit components packing and it proceeds as follows:

1) Firstly, E_O and Add_M (after leaving MSB bit, the hidden mantissa bit) are combined to form an N-1 bit data (TMP).
2) Using sign of LE_O, a N+1 bit sequence of either 00…01 or 11…10 (for regime bits) is created and appended at the MSB of TMP. The sign of LE_O act as the terminating regime bit (1 for -ve and 0 for +ve exponent), and its opposite value sequence acts as regime sequence. Now, 2N bits TMP includes N+1 bits regime, ES bits exponent, and N-ES-1 bits mantissa (Line 43-44). These are constructed as below:

$$TMP = \{\overbrace{N\{!LE\_O[MSB]\}, LE\_O[MSB]}^{N+1\ Bits\ Regime\ Sequence}, \overbrace{E\_O, Add\_M[N-2:ES]}^{Exponent\ -\ Mantissa}\}$$

3) Actual composition of posit is obtained by dynamically right shifting TMP by R_O amount and taking LSB N-1 bits as

the pack of regime bits, exponent, and mantissa. This is done by a dynamic right shifter of word size 2N-bits and shifting size RS-bits (Line 45), which need RS numbers of 2N-bit 2:1 MUXs. The right side shifted out bits needed to be preserved for rounding purposes.

4) If large sign (LS) is true, shifted TMP requires being negated (Line 46), as per the requirement of -ve posit.
5) At this stage, rounding operation is performed. Rounding requires few logic gates for the computation of ULP (Unit at Last Place) and an incrementor. For simplicity, we have used round-to-zero method. Other methods can be introduced similar to the floating point standard.
6) The LSB N-1 bits of final TMP value is then combined with the large sign-bit (LS) to produce the final posit addition result. It is produced while considering ZERO and Infinity check of the input operands as discussed earlier.

All the above processing is parameterized for N and ES, and the source code of proposed posit adder generator is provided as open-source at [9].

IV. Implementation Results

A single cycle implementation of proposed posit adder is demonstrated on a Xilinx FPGA device (xc6vlx365t-3ff1759) and on UMC 90nm ASIC platform, for a range of ES values, and details are shown in Fig. 3. FPGA results are obtained after place and route, while ASIC results are obtained after synthesis. Period on FPGA platform is obtained by placing registers at the primary inputs and outputs to avoid IO pin delay. The functional verification is done by comparing the hardware simulation results with the Julia package for posit [10] provided by the posit developers. It is completely validated for 8-bit posit with varying ES value, and with over several millions random test cases for 16 and 32-bit posit.

For a given posit word size, the area and period are mostly similar due to a smaller difference in the hardware requirement across ES values variation. It mostly varies for exponent computation, which consists of few smaller adders/subtractors (mainly related to Lines 28, 38-39 in algorithm-1). The major components (LODs, LZD, Dynamic Left/Right Shifters, Mantissa Add/Sub, etc) are determined by the parameter N and RS ($Log_2\ N$). Marginally, it is also due a generic coding/implementation, which impedes any specific optimizations.

The proposed posit implementation is visualized against single precision (SP) FP adder to visualize on where does posit implementation stands, and also to provide a motivation for more exploration of posit arithmetic and its usage. As suggested by posit developer, 32 posit with ES=3 is the closest match to the SP format. Implementation details of a 32 bit posit (with ES=3) and SP adder is shown in Table-I. These literature are mostly targeted on older device (Virtex-2pro), nonetheless, it provides a reasonable picture on the position of posit adder. All the three prime FP adder algorithms, standard algorithm, LOP (leading one prediction) method and two-path FP adder method are included here. The standard method requires fewer resources but with higher delay/period, however, the two-path method requires more resources but with smaller delay/period. Thus, two-path method is used in most of commercial systems including AMD, Intel, and PowerPC [11], [12]. Positively, the posit implementation finds an average place among various FP adder methods.

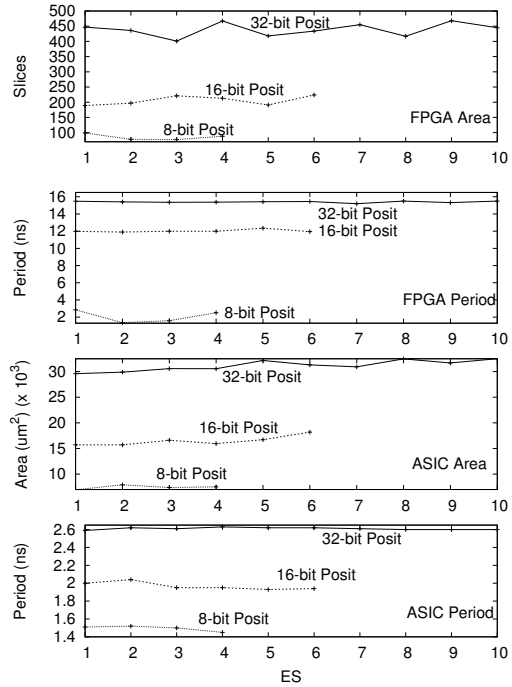

Fig. 3: FPGA and ASIC Implementation details for 8-bit, 16-bit, and 32-bit Posit Adder with varying ES

TABLE I: 32-bit Posit Adder (ES=3) vs SP FP adders

FP Adder Methods	Slices	Period (ns) × Latency(cycle)
Standard[13] (Normal)	551	4.0× 19
Standard[14] (Normal)	495	5.12× 13
Standard[15](Normal)	570	6.67× 10
Standard[11](Denormal)	541	27.06× 1
LOP [11](Denormal)	748	25.33× 1
Two-Path [11], [12](Denormal)	1018	21.82× 1
32 bit Posit Adder (ES=3)	401	15.353× 1

V. Conclusions

The Universal Number System (Unum) is an interesting development in the number system theory with posit as a recent development also called type-3 to Unum. This paper addressed the hardware architecture generation for the posit addition/subtraction arithmetic. It proposed a hardware algorithmic computational flow for posit adder and modeled its architectural, which addressed several key implementation challenges in posit. The entire modeling is carried out in a parameterized manner in order to facilitate others for its simple use under desired parameters. The entire modeling is made open-source. The implementation is demonstrated on FPGA and ASIC platforms, and functionality is exhaustively validated against software. Its hardware implementation metrics are on par with SP FP adder architectures.

VI. Acknowledgments

This work is party supported by the Research Grants Council of Hong Kong (Project ECS 720012E), and the Croucher Innovation Award 2013.

REFERENCES

[1] Gustafson, John L., *The End of Error: Unum Computing*, 1st ed. Chapman and Hall/CRC Press, 2015.

[2] J. Gustafson, "A radical approach to computation with real numbers," *Supercomputing Frontiers and Innovations*, vol. 3, no. 2, 2016. [Online]. Available: http://superfri.org/superfri/article/view/94

[3] Gustafson, John L. and Yonemoto, Isaac. (2017) Beating Floating Point at its Own Game: Posit Arithmetic. [Online]. Available: http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

[4] W. Tichy, "The end of (numeric) error: An interview with john l. gustafson," *Ubiquity*, vol. 2016, no. April, pp. 1:1–1:14, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2913029

[5] Rich Brueckner. (2015) Slidecast: John Gustafson Explains Energy Efficient Unum Computing. inside HPC. [Online]. Available: https://insidehpc.com/2015/03/slidecast-john-gustafson-explains-energy-efficient-unum-computing/

[6] W. Tichy, "Unums 2.0: An interview with john l. gustafson," *Ubiquity*, vol. 2016, no. September, pp. 1:1–1:16, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/3001758

[7] John L. Gustafson. (Feb 01, 2017) Beyond Floating Point: Next-Generation Computer Arithmetic. Stanford EE Computer Systems Colloquium. [Online]. Available: http://web.stanford.edu/class/ee380/Abstracts/170201.html,https://www.youtube.com/watch?v=aP0Y1uAA-2Y&feature=youtu.be

[8] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[9] Manish Kumar Jaiswal. (2017) Posit Adder HDL Arithmetic. [Online]. Available: https://github.com/manish-kj/Posit-HDL-Arithmetic/tree/master/Posit-Adder

[10] Yonemoto, Isaac. (2017) Sigmoid Numbers for Julia. [Online]. Available: https://github.com/interplanetary-robot/SigmoidNumbers

[11] A. Malik, D. Chen, Y. Choi, M. H. Lee, and S. B. Ko, "Design tradeoff analysis of floating-point adders in fpgas," *Canadian Journal of Electrical and Computer Engineering*, vol. 33, no. 3/4, pp. 169–175, Summer 2008.

[12] P. M. Seidel and G. Even, "Delay-optimized implementation of ieee floating-point addition," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 97–113, Feb 2004.

[13] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," in *Proceedings of 18th International Parallel and Distributed Processing Symposium*. IEEE, 2004, pp. 149–156.

[14] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, ser. FPGA '04. New York, NY, USA: ACM, 2004, pp. 171–180.

[15] P. Diniz and G. Govindu, "Design of a field-programmable dual-precision floating-point arithmetic unit," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Aug 2006, pp. 1–4.