Parameterized Posit Arithmetic Hardware Generator

Rohit Chaurasiya*, John Gustafson†, Rahul Shrestha*, Jonathan Neudorfer‡, Sangeeth Nambiar‡, Kaustav Niyogi‡, Farhad Merchant§, Rainer Leupers§
*Indian Institute of Technology, Mandi
†National University of Singapore, Singapore
‡Bosch Research and Technology Centre - India, Bangalore
§Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany
{d17018@students., rahul_shrestha@}iitmandi.ac.in, johngustafson@earthlink.net
{Jonathan.Neudorfer, Sangeeth.Nambiar, Kaustav.Niyogi}@in.bosch.com, {farhad.merchant, leupers}@ice.rwth-aachen.de

***Abstract*—Hardware implementation of Floating Point Units (FPUs) has been a key area of research due to their massive area and energy footprints. Recently, a proposal was made to replace IEEE 754-2008 technical standard compliant FPUs with Posit Arithmetic Units (PAUs) due to the greater accuracy, speed, and simpler hardware design. In this paper, we present the architecture of a parameterized PAU generator that can generate PAU adders and PAU multipliers of any bit-width pre-synthesis. We synthesize generated arithmetic units using the parameterized PAU generator for 8-bit, 16-bit, and 32-bit adders and multipliers and compare them with IEEE 754-2008 compliant adders and multipliers. Both, synthesis for Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) are performed. In our comparison of m-bit PAU units with n-bit IEEE 754-2008 compliant units, it is observed that the area and energy of a PAU adder and multiplier are comparable to their IEEE 754-2008 compliant counterparts where $m = n$. We argue that an n-bit IEEE 754-2008 adder and multiplier can be safely replaced with an m-bit PAU adder and multiplier where $m < n$, due to superior numerical accuracy of the PAU; we also compare m-bit PAU adders and multipliers with n-bit IEEE 754-2008 compliant adders and multipliers. As an application example, we examine performance in the domain of signal processing with and without PAU adders and multipliers, and show the advantage of our approach.**

***Index Terms*—computer arithmetic; floating point arithmetic; numerical error; posit arithmetic**

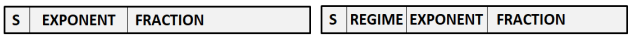
I. Introduction

Floating point arithmetic hardware architectures have been essential components in several domain specific and general purpose platforms due to demands set by the applications being executed on these platforms [1][2]. It is observed that around 50% of the chip area and energy footprints are consumed by a single-precision Floating Point Units (FPUs) in a simple scalar in-order processing element, while with a double precision FPU, the area and energy footprints can reach up to 60–70% of the energy footprint of the processing element [1][3]. Energy and area are not the only concerns with an IEEE 754-2008 compliant hardware FPUs; the low accuracy and high complexity of the standard are also major concerns. Lately, there have been several implementation errors in the realization of FPUs by major commercial vendors [4][5]. These anomalies are mainly due to intricacies in the IEEE standard that undermine the efforts of researchers and engineers who work toward the realization of IEEE-compliant FPUs. Recently, some researchers have taken a "U-turn" toward "approximate computing" where compliance with the IEEE 754 floating point standard is viewed as the wrong path to achieve energy efficiency. The expressive power of a string of bits is not efficiently exploited when implementations are held to IEEE 754 compliance. Ultimately, some of the major computer architecture vendors have decided to drop IEEE compliance while supporting arbitrary precision arithmetic, something not even mentioned in the IEEE 754-2008 technical standard document [6]. Despite several efforts in the literature by researchers to overcome complications of the IEEE technical standard for floating point arithmetic, the mentioned alternative implementations have not been advantageous enough to merit widespread adoption. Recently, however, there was a proposal by Gustafson based on the projective reals, called *posit arithmetic* [7]. A qualitative comparison of m-bit posit arithmetic and m-bit IEEE 754 arithmetic is shown in table I.

TABLE I
Comparison of Posit vs. IEEE 754-2008 Arithmetic

Arithmetic Format	IEEE 754-2008	Posit
Portability/Reproducibility	No	Yes
Accuracy	Lower	Higher
Dynamic Range	Lower	Higher
Hardware	More Complex	Simpler
Area and Energy Footprints	Higher	Lower
Overflow/Underflow	Yes	No
Redundant Representations	Many	None

It can be observed in table I that the posit arithmetic has uniform advantages over IEEE 754 floats, as opposed to offering tradeoffs. Preliminary studies show that posit compliant arithmetic hardware is less complex than IEEE 754 compliant floating point arithmetic hardware [8]. In this paper, we tackle the challenge of creating the first practical, parameterized Posit Arithmetic Unit (PAU) generator. The PAU generator is capable of generating adders and multipliers with different configurations. The generated units are exhaustively verified against standard software libraries. It is experimentally



(a) IEEE 754-2008 Format (b) Posit Format

Fig. 1. IEEE 754 and Posit Number Formats

shown that posit adders and multipliers attain higher numerical accuracy than their IEEE 754 compliant floating point unit counterparts. The Major contributions in this paper are as follows:

- We present the first complete parameterized, area-efficient and energy-efficient PAU generator that can generate posit adders and posit multipliers with different sizes and configurations.
- A detailed comparison of PAU generated units is presented, and it is shown that a PAU consumes less energy and occupies less area compared to state-of-the-art realizations.
- It is experimentally shown that an n-bit IEEE 754 compliant adder/multiplier can often be safely replaced by an m-bit posit adder/multiplier, where $m < n$.
- An application study with PAU-generated units is presented that shows reduction of the overall area and energy footprints of applications in posit arithmetic while maintaining higher accuracy.

II. Background

A. IEEE 754-2008 Technical Standard

The IEEE 754-2008 standard defines interchange formats, multiple rounding modes, required and optional operations, internal processor flag behavior, and exception handling. An IEEE 754-2008 compliant representation is divided into three parts as shown in fig. 1(a): a sign bit, exponent bits, and fractional bits. *Mantissa* and *significand* are alternative terms used in the literature for what we call the *fraction*, a value between 0 and 1. Here, we use the term significand to indicate the implied leading 1 plus the fraction, a value between 1 and 2 for normal floats. A summary of IEEE 754-2008 binary format for different precisions is shown in table II. The significand bits in the table I include an implicit `1` bit for normal numbers and a `0` bit for "subnormal" numbers (an exception case when all exponent bits are zero). The value x of a floating point number that does not fall into one of the many exception categories is given by

$$x = (-1)^{MSB} \times (1 + \sum_{i=1}^{fn-1} b_{fn-1-i} 2^{-i}) \times 2^{exp-bias} \quad (1)$$

where *fn* is the number of fraction bits, *exp* is the exponent bits interpreted as an unsigned integer, and $bias = 2^{exp\_bits-1} - 1$. Table I also shows the dynamic range of different precisions along with the percentage of area occupied by the arithmetic hardware when incorporated in a processor die of an in-order simple scalar processor. Note that for the higher precisions, the area occupied can go up to 75% of total processor die area and 80% of the energy [9][1]. Further details of IEEE 754-2008 can be found in [6] while a concise description of floating point arithmetic can be found in [10].

TABLE II
Summary of IEEE 754-2008 Basic and Interchange Formats

Format (bits)	Exp. Bits	Sig. Bits	Dynamic Range	%Area	%Energy
binary16	5	11	6×10^{-8} to 7×10^{4}	25-30%	40%
binary32	8	24	1×10^{-45} to 3×10^{38}	35-40%	50%
binary64	11	53	5×10^{-324} to 2×10^{308}	50-55%	60%
binary128	15	113	6×10^{-4966} to 1×10^{4932}	60-65%	70%
binary256	19	237	2×10^{-78984} to 2×10^{78913}	70-75%	80%

B. Posit Arithmetic

Recently, posit arithmetic has been proposed as a direct replacement for IEEE 754 floats. Posit arithmetic has several advantages over IEEE 754 compliant arithmetic. The posit number format is shown in fig. 1(b). There are only two exception cases: zero and Not-a-Real (NaR). For all other cases, the value x of a posit is given by

$$x = (-1)^{MSB} \times useed^{k} \times 2^{exp} \times (1 + \sum_{i=1}^{fn-1} b_{fn-1-i} 2^{-i}) \quad (2)$$

The regime indicates a scale factor of $useed^k$ where $useed = 2^{2^{es}}$ and *es* is the exponent size. The numerical value of k is determined by the *run length* of 0 or 1 bits in the string of regime bits. The use of run length encoding of the regime automatically allows more fraction bits for the more common values for which magnitudes are closer to 1, and thus provides tapered accuracy in a bit-efficient way. Further details about the posit number format can be found in [7].

III. Related Work

A. Posit Arithmetic-Based Implementations

Since the advent of posit arithmetic, there have been several software and hardware implementations of posits. A parameterized adder and multiplier is presented in [8]. Although the universal number posit arithmetic generator presented in [8] is the first parameterized implementation of a posit adder and multiplier, it is not a complete realization. Its generator lacks several features required for posit arithmetic; for example, it does not perform posit rounding (round to nearest, tie to even, or *unbiased rounding*) for addition or multiplication, making the implementation noncompliant and reducing accuracy. Furthermore, as per the Field Programmable Gate Array (FPGA) synthesis reports, the implementation presented in [8] occupies 30% more area than its IEEE 754 counterpart implementations. In our realization, we show that the area and the energy footprints of posit adders and multipliers are comparable to their IEEE 754 counterpart implementations. It means that for an m-bit posit adder/multiplier and an m-bit IEEE 754 compliant adder/multiplier, area and energy footprints are comparable but posits provide much better accuracy; for an m-bit posit adder/multiplier and an n-bit IEEE 754 compliant adder/multiplier where $m < n$, the m-bit posit adder/multiplier has a significant advantage in area and

energy footprint over a corresponding IEEE 754 compliant counterpart capable of providing similar accuracy. A posit adder architecture is presented in [11]. The implementations presented in [11] have limitations similar to those of the posit arithmetic generator presented in [8]. To the best of our knowledge, the exposition here is the first complete implementation of posit arithmetic units with a holistic approach that covers all the aspects of the scheme.

B. IEEE-754 2008 Technical Standard-Based Implementations

We cover IEEE 754 compliant implementations and transprecision implementations, where IEEE 754 non-compliant implementations are tailored for domain-specific computing platforms. IEEE 754 conforming implementations may not necessarily be IEEE 754-2008 compliant, but may support only a subset of the features advocated by the standard. VFLOAT is one such open source package developed for FPGAs that implements a majority of the recommendations of the IEEE 754-2008 technical standard, but omits support for subnormal numbers and the full set of rounding modes [12]. In such a system, it is possible for $x - y$ to produce a zero result even when $x \neq y$, creating additional hazards for programmers as well as for hardware designers who rely on the subtraction functional unit to set comparison flags.

Recently, *transprecision* hardware implementations have emerged as a viable solution to applications in the domain of approximate computing, where higher performance is attained at the cost of tolerated numerical error. One such recent implementation is presented in [13], where authors have proposed a transprecision computing paradigm capable of switching precision of the operations at runtime in the applications. In the transprecision computing paradigm, when numerical errors are encountered, the minimum precision of operations is set such that despite errors, the application has acceptable accuracy [1]. A similar approach, improving energy efficiency of overall system through transprecision arithmetic, is presented in [14] and [15]. While transprecision architectures are viable solutions to low power computing, we hypothesize that they can be improved further by employing posit arithmetic architectures instead of IEEE 754 floating point architectures to maximize information-per-bit. We examine this hypothesis with a simple study of IEEE 754 compliant floating point and posit compliant arithmetic architectures.

We have generated posit arithmetic units with different configurations by supplying parameters of word size N and Exponent Size *es* and compared them with several recent posit and IEEE 754 based implementations, and found that the area, and energy of our implementation is better than state-of-the-art realizations of posit arithmetic units, and comparable to IEEE 754 based arithmetic hardware units with better accuracy.

IV. Posit Arithmetic Unit Generator

This section presents the proposed PAU-generator algorithms for addition and multiplication operations.

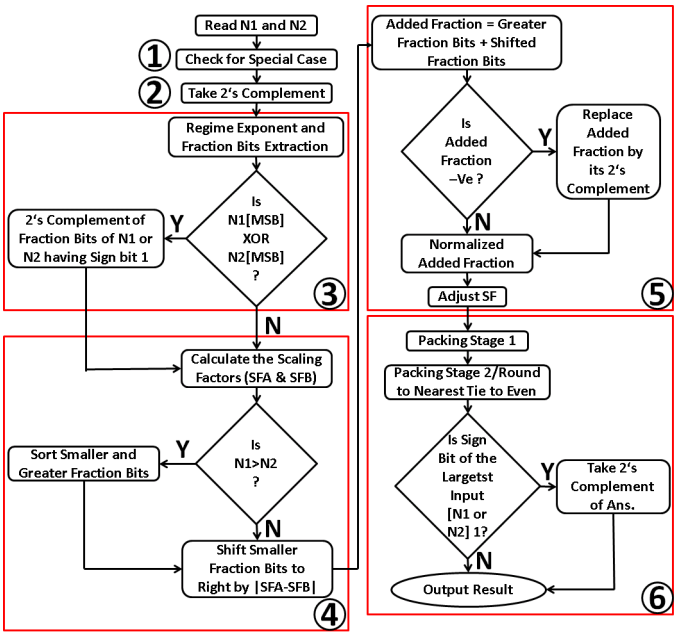


Fig. 2. Schematic Flow Diagram of the Proposed Posit Adder/Subtraction Algorithm

A. Adder Generator

We illustrate the complete flow diagram of the new posit add/substract process in fig. 2. The encircled numbers to the right bottom of the red blocks represent the subsection of IV-A that has detailed algorithmic description for each block:

1) Special Case: In posit arithmetic, several exceptions of IEEE 754 float are simplified into two cases: zero and NaR. Exceptional cases like imaginary numbers, multi-valued and indeterminate results, signed infinity and unsigned infinity are categorized as NaR. Thereby, checking for such exceptions is essentially performed by a reduction OR operator, as shown in algorithm 1 (line no. 6).

2) 2's Complement: Since posit arithmetic directly maps to 2's complement signed integers, an efficient way to obtain the regime and exponent values is to first determine its absolute value. Subsequently, the posit bit string is converted to the equivalent 2's complement representation provided its sign bit is 1, as shown in algorithm 1 (line no. 7).

3) Extraction of Regime, Exponent and Fraction Bits: Posit arithmetic is at an early stage in the hardware implementation domain. To the best of our knowledge, there is only one reported work in this domain [8]. It uses a leading one detector for decoding the regime and exponent bits of a posit bit string having a value greater than or equal to 1 (positive regime) and a leading zero detector for a posit bit string having a value less than 1 (negative regime). In this paper, we propose an implementation that requires only a leading zero detector for decoding the regime, exponent and fraction bits.

As presented in algorithm 1, the regime, exponent and fraction bits are extracted from OutputTwos $[N-2:0]$. It is to be noted here that the sign bit is excluded because we

are decoding a positive posit number. The pseudocode for the aforementioned process has been illustrated in algorithm 1 (lines no. 8–14) and its explanation is as follows:

- The OutputTwos $[N-2]$ bit is replicated $[N-1]$ times and XORed with OutputTwos $[N-2:0]$. This operation inverts all the bits of the OutputTwos and adds 1 to the least-significant bit (LSB) if the OutputTwos $[N-2]$ bit is 1 (+ve regime), else OutputTwos remains unchanged (line 8).
- In order to count the sequence of 0 bits terminating in a 1 bit, the value OutputTwos $[N-2:0]$ is applied to the leading zero detector module (line 9).
- For extracting exponent and fraction bits, OutputTwos $[N-2]$ is left shifted by ZC -1 (because the minimum value of ZC is 1). This step dynamically shifts the exponent bits towards the extreme left (lines 10).
- The fraction bits are extracted from the shifted bits. The implied 1 bit is attached before the fraction bits and three lower bits are appended after the LSB which act as the Guard (G), Round (R), and Sticky (S) bits needed for unbiased rounding. It is to be noted that the posit format obviates the need for subnormal values and hence, the implied bit before the fraction is always 1 (line 11).
- The exponent bits are extracted from the extreme left. The number of bits to be extracted depends upon the value of *es* (line 12).
- The OutputTwos $[N-2]$ bit aids to determine the regime value. If this bit is 1 then the regime value is (ZC -1) (+ve regime) else the regime value is $-$ZC (–ve regime) (line 13).
- If the operation to be performed is subtraction, then the 2's complement of the fraction bits for the input posit bit string having sign bit 1 is taken (line 14).

Algorithm 1 Regime, Exponent, Fraction Bits Extraction

1: Posit Size: N; Exponent Size: *es*
2: OutputTwos $[N-2:0]$ ▷ Input Size
3: Regime $[\log_2(N):0]$ ▷ Input Size
4: FractionBits $[N+1-es:0]$ ▷ Output Size
5: ExpBits$[es-1:0]$ ▷ Exponent Size
6: Atest ← InputA$[N-1]$ & $\sim$|InputA$[N-2:0]$ ▷ **Checking Special Case**
7: OutputTwos ← $\{[N]\{$InputA$[N-1]\}\}$ XOR InputA$[N-1:0]$ + InputA$[N-1]$ ▷ **2's Complement of Input**
8: InvertInput ← $\{[N-1]\{$OutputTwos $[N-2]$ $\}\}$ XOR OutputTwos $[N-2:0]$
9: ZC ← Leading Zero Detector (InvertInput)
10: temp$[N-4:0]$ ← OutputTwos $[N-4:0] \ll$ (ZC -1)
11: FractionBitsTemp ← $\{2'b01, \text{temp}, 3'b0\}$
12: ExpBits ← temp$[N-4:N-3-es]$
13: Regime ← OutputTwos $[N-1]$? (ZC -1) : –ZC
14: FractionBits ← ((InputA $[N-1]$ XOR InputB $[N-1]$) & InputA/B [MSB]) ? –FractionBitsTemp : FractionBitsTemp

4) Calculating the Scaling Factor (SF): This module computes the net scaling factor (SFA & SFB) (effective exponent) for both the inputs. Here we define net SF as $((2^{es})^{Regime}+es$ *bits*) which provides the amount of shift required for the smaller value to align its binary point with the larger value. The pseudocode for this process is presented in algorithm 2 and its brief description is given below.

- The regimeA value is left shifted *es* times (lines 1, 2).
- The amount by which the smaller significant needs to be shifted to the right for binary point alignment is the absolute difference of the net scaling factors (line 5).
- By comparing the input posit strings, the smaller and larger value significand bits and greater net scaling factor are obtained (lines 6–14).

Algorithm 2 Calculating Scaling Factor

1: ShiftRegimeA ← RegimeA $\ll$ *es*
2: ShiftRegimeB ← RegimeB $\ll$ *es*
3: SFA ← ShiftRegimeA + ExpBitsA
4: SFB ← ShiftRegimeB + ExpBitsB
5: ValuetoShift ← |SFA – SFB|
6: **if** InputA$>$ InputB **then**
7: GreaterFractionBits ← FractionBitsA
8: SmallerFractionBits ← FractionBitsB
9: GreatestScalingFactor ← SFA
10: **else**
11: GreaterFractionBits ← FractionBitsB
12: SmallerFractionBits ← FractionBitsA
13: GreatestScalingFactor ← SFB
14: **end if**

5) Adding Significand: This module adds the significands (the fraction bits with the implied 1 bit prepended) of the posit inputs, and provides the normalized significand bits with three extra bits (G, R and S). It also adjusts the greater net scaling factor if there is a carry from the addition of significand bits or subtractive cancellation of leading bits. Our proposed implementation performs a reduction OR operator over the result of added significand bits to check whether the result is zero (for $X+(-X)$). The pseudocode is presented in algorithm 3.

- The greater posit significand bits are added to the arithmetic right-shifted smaller posit significand bits. (line 1)
- For the case $X+(-X)$, the AnsIsZero flag returns 1.
- Renormalization of the significand bits and adjustment of the greater net scaling factor is analogous to that for floating point (lines 5–16).

6) Decoding Regime and Exponent Value, Encoding and Rounding: Depending on the sign of adjusted net scaling factor, this module extracts the exponent and regime values and encodes them, followed by rounding. The pseudocode is presented in algorithm 4 (lines 1–8).

Packing Stage 1: As the number of fraction bits present in the result is dynamic, in packing stage-1 extra fraction bits are

Algorithm 3 Adding Significand

1: SignificandAdd ← (GreaterFractionBits + (SmallerFractionBits ≫ ValuetoShift))
2: AnsIsZero ← ∼| SignificandAdd
3: AddOperation ← InputA$[N-1]$ OR InputB$[N-1]$
4:
5: **if** SignificandAdd[MSB] $= 0$ & AddOperation=0/1 **then**
6: ToNormalizedFraction ← SignificandAdd
7: Shift $= 0$
8: **else if** SignificandAdd[MSB] =1 & AddOperation=0 **then**
9: ToNormalizedFraction ← SignificandAdd ≫ 1
10: Shift $= 1$
11: **else if** SignificandAdd[MSB] =1 & AddOperation=1 **then**
12: ToNormalizedFraction ← –SignificandAdd
13: ZC1 ←Leading Zero Detector(ToNormalizedFraction)
14: NormalizedFraction ← ToNormalizedFraction ≪ ZC1
15: **end if**
16: Adjusted Scaling factor ← GreatestScalingFactor+Shift−ZC1

eliminated; this leaves behind the required fraction bits along with the G, R and S bits.

- For the +ve regime, the normalized fraction bits and exponent bits are concatenated with two bits `10`, followed by concatenating with the number of `1` bits equivalent to the maximum regime value. This avoids error in the calculation of S, the so-called "sticky" bit (line 10).
- For the –ve regime, the normalized fraction bits and scaling bits are concatenated with two bits '01', followed by concatenating with the number of 0 bits equivalent to the minimum regime value (which is same as the maximum regime value since posits have symmetric magnitude) (line 11).
- Necessary care is taken in the shifting amount for a corner case (lines 12–16).
- Lines 17–21 dynamically prepend the regime bits, i.e, a run of `1` bits ending with a `0` bit for +ve regime and a run of `0` bits ending with a `1` bit for –ve regime. The dynamic prepending of bits is achieved with an arithmetic right shifter.
- Based on the sign of adjusted net scaling factor, the tempAns1/2 is shifted right (lines 17–21).

Packing Stage 2/Unbiased rounding: There is only one rounding mode for posits: round to nearest, tie to even, sometimes called *unbiased rounding*. Rounding is performed by checking the G, R and S bits (lines 22–24). In the case of floats, if all the significand bits are `1` when performing the round operation, the exponent must be adjusted with a conditional check for overflow and underflow, whereas nothing needs to be done for posits; if all the significand bits are `1`, the exponent and regime bits get adjusted automatically (line 27). Analogous to the sign-magnitude representation used for floating point, the sign of the final answer is applied using 2's complement if necessary, based on the sign bit of the largest magnitude posit input (lines 30–32).

Algorithm 4 Decoding Regime and Exponent Value, Encoding and Rounding

1: Absolute SF ← Adjusted Scaling Factor [MSB] ? –Adjusted Scaling Factor : Adjusted Scaling Factor
2: **if** Adjusted Scaling Factor[MSB] $= 0$ **then**
3: ExpBits ← Absolute SF $[es-1:0]$
4: RegimeAns ← Absolute SF ≫ *es*
5: **else**
6: ExpBits ← Adjusted Scaling Factor $[es-1:0]$
7: RegimeAns1 ← Adjusted Scaling Factor ≫ *es*
8: RegimeAns ← –RegimeAns1
9: **end if**
10: TempAns1 ← {2'b10,ExpBits, NormalizedFraction, {[maxregime]{1'b0 } } ▷ **Packing Stage 1**
11: TempAns2 ← {2'b01,ExpBits, NormalizedFraction, {[maxregime]{1'b0 } }
12: **if** RegimeAns={log(N){1'b1}} **then**
13: Shiftnegexp ← RegimeAns
14: **else**
15: Shiftnegexp ← RegimeAns−1
16: **end if**
17: **if** Adjusted Scaling Factor[MSB] $= 0$ **then**
18: TempAns ← TempAns1 ≫ RegimeAns
19: **else**
20: TempAns ← TempAns2 ≫ Shiftnegexp
21: **end if**
22: Guard bit ← Extract from TempAns ▷ **Packing Stage 2**
23: Round ← Extract from TempAns
24: Sticky ← Extract from TempAns
25: LSB ← Extract from TempAns
26: checkround ← ((LSB & Guard) | (Guard & Round)) | (Guard & Round | Sticky)
27: IntermediateAns ← {1'b0,TempAns}+checkround
28: **if** AnsIsZero= 0 **then**
29: FinalAns ← 0
30: **else if** take2sans=1 **then**
FinalAns ← –IntermediateAns
31: **else**
FinalAns ← IntermediateAns
32: **end if**
33: AdditionResult ← Special ? inf : FinalAns

B. Multiplier Generator

The computational flow of a posit multiplier is almost the same as for floating point multiplication, i.e. the net SF of posits are added and their significands are multiplied and rounded. The pseudocode for multiplication is as shown in algorithm 5. In a posit multiplier, the reduction OR is used to check whether the inputs are 0 or not. For a posit multiplier, there is a minor difference in the extraction of regime, fraction and significand bits compared to their extraction in a posit adder. Since multiplication of significands will result in bit width, which is the sum of the individual significand widths,

there is no need to include extra three bits at the end of the significand result (line 3). The rest of the algorithm for packing and rounding is same as the one described for the posit adder in algorithms 1 through 4. While decoding regime and exponent values from the net adjusted scaling factor of the result, special care needs to be taken for cases such as (*maxpos* $\times$ *maxpos* or *minpos* $\times$ *minpos*) for which the effective exponent is greater than or less than what can be represented by the posit format. Lines 11 and 12 of algorithm 5 handle this.

Algorithm 5 Proposed Multiplier Algorithm

1: ZeroInputTest $\leftarrow \sim$|InputA$[N-1:0]$
2: **Regime, Exponent and Fraction bits Extraction** $\leftarrow$ Same as Algorithm 1 with the following difference:
3: FractionBitsTemp $\leftarrow \{1'b1, temp\}$
4: **Adding Scaling factor:**
5: AddedScalingFactor $\leftarrow$ {ShiftRegimeA + ExpBitsA} + {ShiftRegimeB + ExpBitsB}
6: **Significand Multiplication**:
7: Mul $\leftarrow$ (1+FractionBitsA)$\times$(1+FractionBitsB)
8: NormalizedFraction $\leftarrow$ Mul $\gg$ Mul[MSB]
9: Adjusted Scaling factor $\leftarrow$ AddedScalingFactor + Mul[MSB]
10: **Calculating Regime and Exponent value** $\leftarrow$ Same as Algorithm 4.
11: FinalRegime $\leftarrow$ RegimeAns[MSB] ? { $\log_2(N) - 1$ { {$1'b1$},{$1'b0$} }} : RegimeAns[$\log_2(N) - 1 : 0$]
12: FinalExp $\leftarrow$ RegimeAns[MSB] | (&FinalRegime) ? { *es* { {$1'b0$} }} : Exp_bits
13: **Packing Stage 1 and Rounding** $\leftarrow$ Same as Algorithm 4 with the following changes.
14: **if** RegimeAns = $\{\log_2(N)\{\{1'b1\}\}\}$ **then**
15: Shiftnegexp $\leftarrow$ RegimeAns-2
16: Shiftposexp $\leftarrow$ RegimeAns-1
17: **else**
18: Shiftnegexp $\leftarrow$ RegimeAns-1
19: Shiftposexp $\leftarrow$ RegimeAns
20: **end if**
21: **if** Adjusted Scaling Factor[MSB] $= 0$ **then**
22: TempAns $\leftarrow$ TempAns1 $\gg$ Shiftposexp
23: **else**
24: TempAns $\leftarrow$ TempAns2 $\gg$ Shiftnegexp
25: **end if**
26: **Packing Stage 2 and Convergent Rounding** $\leftarrow$ Same as Algorithm 4 from line 22.

V. Results and Discussion

The proposed posit arithmetic adder/subtracter is successfully implemented on a Zedboard with a Zynq-7000 SoC. Vivado 2017.4 is used for the FPGA synthesis results. To have a fair comparison with the work presented in [8], the code is obtained from [16] and synthesized using Vivado 2017.4.

A. Verification of PAU

As per [7], there is only one rounding mode for posits (i.e. unbiased rounding); the Julia package presented in [17] implements this mode. We created two implementations of posit: one with round to zero (RZ) to have fair comparison with [8], and another with unbiased rounding (RE). The functional simulation results for the adder and multiplier with unbiased rounding are verified with the Julia package for the posit ($N = 8, es = 4$) configuration. It is observed that for every input combination, the results of our implementation exactly matches the results of the Julia package.

B. FPGA Synthesis Results

Fig. 3(a) shows the resource utilization on an FPGA for a posit adder with a variable *es* size. It can be observed that the proposed posit adder with RZ has less resource utilization than [8]. Also, the resource utilization of the posit adder with RE is comparable to [8].

Fig. 3(b) shows the FPGA datapath delay for a posit adder with variable *es* size. We see that the proposed posit adder with RZ has a delay similar to [8]. As the posit adder with RE has an extra module for unbiased rounding, it has more datapath delay than [8].

Fig. 3(c) shows the resource utilization on an FPGA for a posit multiplier with variable *es* size. It can be observed that the proposed posit multiplier with RZ uses fewer resources than [8]. Also, the resource utilization of a posit multiplier with RE is similar to [8].

Fig. 3(d) shows the datapath delay on FPGA for a posit multiplier with variable *es* size. The proposed posit multiplier with RZ has less datapath delay as compared to [8]. The datapath delay of posit multiplier with RE is similar to [8].

C. ASIC Synthesis Results

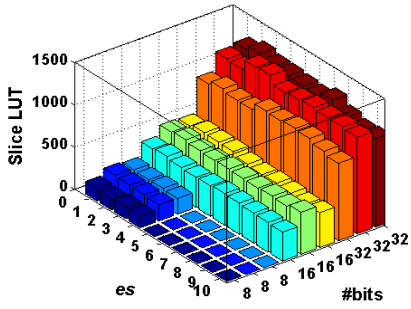
ASIC synthesis is performed using Synopsys design compiler with the 90 nm-CMOS Faraday library for proposed PAU generated units and presented in [8]. It can be observed from the table III that for small bit sizes, the proposed posit adders and multipliers outperform the adders and multipliers presented in [8]. For larger sized adders and multipliers, similar performance can be achieved by pipelining the units.

TABLE III
ASIC SYNTHESIS AT 200 MHZ

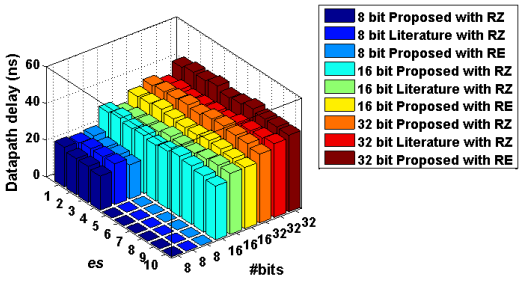
Bit Configuration	Adder				Multiplier			
	Area (μm^2)		Power (mW)		Area (μm^2)		Power (mW)	
	Proposed Work	Literature	Proposed Work	Literature	Proposed Work	Literature	Proposed Work	Literature
(8,1)	2157.56	2472.43	0.15	0.16	1635.42	1917.66	0.12	0.13
(8,2)	2079.95	2436.67	0.14	0.15	1463.72	1923.15	0.11	0.13
(16,1)	6870.18	6366.86	0.47	0.43	7252.78	7767.08	0.57	0.58
(16,2)	6795.71	6210	0.44	0.40	6790.22	7218.28	0.53	0.54
(32,1)	19122.54	15360.12	1.43	1.08	28146.38	26054	2.47	2.27
(32,2)	18517	15471.45	1.33	1.08	27268.30	25338.09	2.39	–

D. Comparison with IEEE-754 32-Bit & 16-Bit Floats

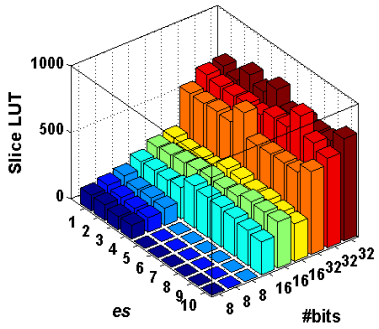
A comparison of the dynamic range and the maximum number of fraction bits for IEEE Floating Point versus posits of various (N, es) configurations is shown in table IV. The maximum fraction bit accuracy holds for posits in the range



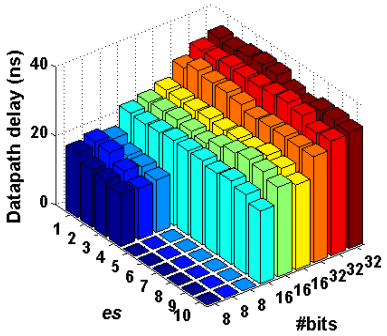
(a) Posit Adder Resource Utilization (Proposed Vs. Literature)



(b) Posit Adder Datapath Delay (Proposed Vs. Literature)



(c) Posit Multiplier Resource Utilization (Proposed Vs. Literature)



(d) Posit Multiplier Datapath Delay (Proposed Vs. Literature)

Fig. 3. Comparision of Posit Adder and Multiplier Resource Utilization (Proposed Vs. Literature) and Data Path Delay (Proposed Vs. Literature) in FPGA Synthesis

$1/useed$ to $useed$, which is $1/256$ to 256 for $es = 3$. The accuracy decreases by one bit for every additional increase or decrease of a factor of $useed$ in the magnitude of values represented as posits. The resource utilization of floating point

TABLE IV
COMPARISON OF POSITS WITH IEEE-754 16 & 32-BIT FLOATS

Bit Configuration	Adder		Multiplier			Fraction Bits	Dynamic Range (decades)
	Slice LUT	Datapath Delay (ns)	Slice LUT	DSP Utilized	Datapath Delay (ns)		
32 : FP	**1049**	**41.576**	**533**	**4**	**29.051**	**23**	**83**
(32,1): Posit	934	38.041	576	4	31.013	28	36
(32,2): Posit	981	40.032	572	4	33.021	27	72
(32,3): Posit	951	39.254	582	4	32.263	26	144
(32,4): Posit	955	39.960	576	4	32.281	25	289
(31,3): Posit	894	39.964	560	4	31.927	25	140
(30,3): Posit	873	39.542	655	3	32.199	24	135
(29,3): Posit	837	39.748	464	2	29.496	23	130
(28,3): Posit	821	39.369	459	2	28.966	22	125
16 : FP	**356**	**27.027**	**212**	**1**	**21.379**	**10**	**12**
(16,1): Posit	391	32.374	218	1	24.041	12	17
(16,2): Posit	404	33.974	223	1	23.680	11	34
(16,3): Posit	386	32.466	219	1	24.078	10	67
(16,4): Posit	371	33.079	233	1	24.581	9	135
(15,1): Posit	382	30.416	207	1	23.848	11	16
(14,1): Posit	353	29.835	184	1	23.282	10	14
(13,1): Posit	290	28.420	181	1	23.445	9	13
(12,1): Posit	254	28.549	167	0	21.142	8	12

adder compared to posit for $m = n$, and $m < n$ bits is shown in table IV for various values of es. It can be observed in the table IV that the resource utilization of posit for $m = n$ is comparable to IEEE floats while it is less than the resource utilization of IEEE floats for $m < n$. The dynamic range approximately doubles for every increment by 1 in es; a $(32, 2)$ posit format has a dynamic range of 72 decades, slightly less than that of a 32-bit IEEE float, yet it has four more bits of significand accuracy for values between $1/16$ and 16.

E. FIR Filter Study

In order to compare the total area utilized on FPGA as well as the accuracy of computation by the floating point arithmetic unit [8] with the proposed posit arithmetic unit implementations, we considered a simple 4-tap Finite-Impulse-Response (FIR) filter.

1) FPGA-Synthesis Comparison of Float and Posit Arithmetic Based FIR Filter: Table V shows the comparison on total slice LUT utilization on FPGA, it can be observed that the proposed work has comparable resource utilization against floating point for $m = n$ bits and less resource utilization for $m < n$ bits. In addition, it shows that our work is more efficient than the reported in [8] in-terms of resource utilization.

2) Accuracy comparison of Float and Posit Arithmetic based FIR Filter: Input sets X[n] with 8-bit representation for floating point and (8,1) representation for posits was applied to a 4-tap FIR filter to obtain the response Y[n]. The filter coefficients are fixed to 1 for simplicity. Y[n] has been converted to its decimal equivalent from its floating point and posit outputs. Table VI and table VII show the obtained output Y[n] and relative error for Y[n] with floating point arithmetic and the

TABLE V
COMPARISON OF RESOURCE UTILIZATION FOR 4 TAP FIR FILTER WITH FLOAT AND POSIT (PROPOSED & LITERATURE)

Bit Configuration	Slice LUT		DSP Utilized		Slice without DSP	
	Proposed Work	Literature	Proposed Work	Literature	Proposed Work	Literature
16: FP	**1956**		**4**		**2579**	
(16,1): Posit	2067	2601	4	4	2810	3746
(15,1): Posit	1963	2237	4	4	2702	3033
(14,1): Posit	1790	1965	4	4	2344	2617
(13,1): Posit	1584	1791	4	4	2032	2331
(12,1): Posit	1831	1268	0	4	1831	1646
32: FP	**4794**		**16**		**7798**	
(32,3): Posit	4981	6220	16	16	9344	10398
(31,3): Posit	4785	5487	16	16	9125	9487
(30,3): Posit	5260	5267	12	16	8301	8751
(29,3): Posit	4576	5225	8	16	7755	8641
(28,3): Posit	4474	5103	8	16	7223	8104

TABLE VI
COMPARISON OF FIR OUTPUT Y[N] WITH FP, LITERATURE[8] AND PROPOSED WORK

X[n]	Expected Y[n]	Y[n] with FP	Y[n] with Literature	Y[n] with Proposed Work
0.000244140625	0.0009765625	0.03125	0.0009765625	0.0009765625
0.005859375	0.0234375	0.03125	0.0234375	0.0234375
0.0390625	0.15625	0.1875	0.15625	0.15625
0.078125	0.3125	0.40625	0.3125	0.3125
0.03125	0.125	0.125	0.125	0.125
0.203125	0.8125	0.90625	0.8125	0.8125
0.390625	1.5625	1.5625	1.5625	1.5625
0.71875	2.875	2.875	2.875	2.875
0.96875	3.875	3.875	3.875	3.875
1.4375	5.75	5.75	5.5	6
0.02734375	0.109375	0.125	0.109375	0.109375
4	16	Inf	16	16
5	20	Inf	20	20
6	24	Inf	24	24
7	28	Inf	28	28
256	1024	NaN	1024	1024
59	236	NaN	192	256

proposed (and literature) [8] posit arithmetic unit; posits have higher accuracy and a larger dynamic range. The largest value that can be represented by an 8-bit floating point with 3 bits of exponent is 15.5 in decimal, and the smallest normalized value is 0.25 in decimal (with denormalization, more values can be packed between zero and the smallest normalized value); it has 221 finite values, 2 zeros, 2 infinities, and 30 NaNs, whereas (8,1) posit representation has a maximum value of 4096 in decimal and a minimum value of 1/4096 in decimal, with unique zero and NaR representations.

VI. CONCLUSION

This paper presented the first complete posit-arithmetic-unit generator that is parameterized to generate adders and multipliers of different sizes. It is shown that in FPGA and ASIC synthesis, the area and power of adders and multipliers generated are comparable with their IEEE 754-2008 technical standard counterparts. Furthermore, it is experimentally shown that n-bit IEEE 754-2008 compliant adders and multipliers can be replaced by m-bit posit adders and multipliers where $m < n$. With 4-tap FIR filters, such a replacement is feasible. Experimental results reflect great potential in posit arithmetic for prospective exploration of high performance and embedded computing systems.

TABLE VII
COMPARISON OF RELATIVE ERROR FOR OUTPUT Y[N] WITH FP, LITERATURE[8] AND PROPOSED WORK

Relative Error in FP	Relative Error in Literature	Relative Error in Proposed Work
31	0	0
0.3333333333333333	0	0
0.2	0	0
0.3	0	0
0	0	0
0.115384615384615	0	0
0	0	0
0	0	0
0	0	0
0	0.0434782608695652	0.0434782608695652
0.142857142857143	0	0
Inf	0	0
Inf	0	0
Inf	0	0
Inf	0	0
NaN	0	0
NaN	0.186440677966102	0.0847457627118644

REFERENCES

[1] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, 2018, pp. 1051–1056. [Online]. Available: https://doi.org/10.23919/DATE.2018.8342167

[2] M. Langhammer and B. Pasca, "High-performance qr decomposition for fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 183–188. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174273

[3] X. Fang and M. Leeser, "Open-source variable-precision floating-point library for major commercial fpgas," *TRETS*, vol. 9, no. 3, pp. 20:1–20:17, 2016. [Online]. Available: http://doi.acm.org/10.1145/2851507

[4] W. E. Wong, X. Li, and P. A. Laplante, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *Journal of Systems and Software*, vol. 133, pp. 68 – 94, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217301334

[5] N. Whitehead and A. Fit-florea, "Precision & performance: Floating point and ieee 754 compliance for nvidia gpus," 2011.

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

[7] J. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, pp. 71–86, Jun. 2017. [Online]. Available: https://doi.org/10.14529/jsfi170206

[8] M. K. Jaiswal and H. K. So, "Universal number posit arithmetic generator on FPGA," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, 2018, pp. 1159–1162. [Online]. Available: https://doi.org/10.23919/DATE.2018.8342187

[9] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Grkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct 2017.

[10] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991. [Online]. Available: http://doi.acm.org/10.1145/103162.103163

[11] M. K. Jaiswal and H. K. H. So, "Architecture generator for type-3 unum posit adder/subtractor," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[12] X. Fang and M. Leeser, "Open-source variable-precision floating-point library for major commercial fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, pp. 20:1–20:17, Jul. 2016. [Online]. Available: http://doi.acm.org/10.1145/2851507

[13] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn, "The transprecision computing paradigm: Concept, design, and applications," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, 2018, pp. 1105–1110. [Online]. Available: https://doi.org/10.23919/DATE.2018.8342176

[14] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini, "A transprecision floating-point architecture for energy-efficient embedded computing," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[15] H. Giefers and D. Diamantopoulos, “Extending the power architecture with transprecision co-processors,” in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.
[16] M. K. Jaiswal, “(2017) Posit HDL Arithmetic. [Online],” https://github.com/manish-kj/Posit-HDL-Arithmetic, 2018, [Online; accessed 19-May-2018].
[17] I. Yonemoto, “2017) Sigmoid Numbers for Julia. Available,” https://github.com/interplanetary-robot/SigmoidNumbers, 2018.