Universal Number Posit Arithmetic Generator on FPGA

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*—Posit number system format includes a run-time varying exponent component, defined by a combination of regime-bit (with run-time varying length) and exponent-bit (with size of up to ES bits, the exponent size). This also leads to a run-time variation in its mantissa field size and position. This run-time variation in posit format poses a hardware design challenge. Being a recent development, posit lacks for its adequate hardware arithmetic architectures. Thus, this paper is aimed towards the posit arithmetic algorithmic development and their generic hardware generator. It is focused on basic posit arithmetic (floating-point to posit conversion, posit to floating point conversion, addition/subtraction and multiplication). These are also demonstrated on a FPGA platform. Target is to develop an open-source solution for generating basic posit arithmetic architectures with parameterized choices. This contribution would enable further exploration and evaluation of posit system.**

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

I. Introduction

Posit is the latest development in Universal Number (unum) system under type-3 unum [1], [2]. Posit is claimed as a possible substitute for floating point (FP) number system [3]. Posit provides many benefits over FP standard, including better dynamic range and accuracy over same bit field, more accurate and exact arithmetic computations [1].

A range of software tools are available for posit using Julia, C, C++ etc. However, to the best of authors knowledge, any hardware architecture generator is yet to available for posit arithmetic. This paper is aimed towards an open-source arithmetic hardware generator for posit number system. Currently, it is focused on basic arithmetic of FP to posit converter, posit to FP converter, addition/subtraction and multiplier arithmetic.

Posit format is shown in (1) which is define by its exponent size (ES). Sign bit function as floating point (FP) standard. Regime numerical value determined by the run length of the regime bits. A sequence of m-bit 0 terminated by (m+1)th true bit gives a value of -m and for a sequence of m-bit 1 terminated by (m+1)th false bit gives a value of m-1 for regime. The exponent bits are just an unsigned integer, and it can be up to ES bits based on the availability at the right side of regime bits. Mantissa bits function similar to the normalized floating point standard, and remaining bits (if available) after regime and exponent are occupied by it. 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 $s * (2^{(2^{ES})})^k * 2^e * f$. Here, zero (00...0) and infinity (100...0) are reserved representation. A detailed information on posit format is recommended to sought from [1].

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

Handling components extraction and components packing due to run-time variation in Posit are primary challenging part in its hardware implementation. This requires a significant dynamicity in related hardware components, and further including parameterization in these including other sub-components of each arithmetic unit, poses an additional design challenge.

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

- Proposed algorithmic flows for hardware architecture of posit arithmetic of FP to posit converter, posit to FP converter, adder and multiplier units.
- Proposed open-source parameterized Verilog HDL generators for each arithmetic architectural units.
- Demonstrated the implementation details on a FPGA platform with 8, 16, 24, 32, 50 and 64 bits implementations with varying exponent size (ES).

II. Proposed Posit Arithmetic Algorithmic Flow and Architectures Generator

This section discusses the all proposed posit arithmetic architectures details. In the discussion below, readers are assumed to have basic understanding of FP standard. As posit is currently not a defined standard, in a practical scenario an application would provides FP inputs as operands, which necessitates a FP to posit converter unit. Similarly, in the last phase of computation a posit to FP converter would also find a trivial place in system. Thus, the discussion of proposed modules are presented in a sequence of FP to posit converter, posit to FP converter, adder/subtractor and multiplier architectures.

A parameterized Verilog HDL is constructed for each unit which takes posit word size (N) and posit exponent size (ES), FP exponent size (E), where required, as its parameter and produces corresponding hardware. As, regime bits can reach up to last bit, RS bits ($= Log_2N$) can accommodate its maximum absolute numerical value.

A. Proposed Floating Point to Posit Converter

The algorithmic flow for FP to posit converter is shown in Algorithm-1 and is presented in a parameterized format. It consists of two major parts, FP data extraction and posit

Algorithm 1 Proposed FP to Posit Converter Flow

```
1:  **GIVEN:**
2:     N: FP / Posit Word Size
3:     E: FP Exponent Field Size
4:     BIAS = (2**(E-1))-1 : FP Exponent Bias
5:     ES: Posit Exponent Field Size
6:  **Input Operand:** $IN$
7:  **FP Data Extraction:Sign** ($S_{FP}$)**, Exponent** ($E_{FP}$)**, Mantissa** ($M_{FP}$)**, Exceptions**
    **(Infinity** ($INF_{FP}$)**, Zero** ($Z_{FP}$)**)**
8:     $S_{FP} \leftarrow IN[N-1]$
9:     $E_{FP} \leftarrow IN[N-2:N-1-E]$
10:    $M_{FP} \leftarrow \{|E_{FP}, IN[N-2:N-1-E]\}$
11:    $Z_{FP} \leftarrow !(|IN[N-2:0])$
12:    $INF_{FP} \leftarrow \&E_{FP}$
13:    Pre-Normalization of FP:
14:       Lshift $\leftarrow$ LOD of $M_{FP}$
15:       $M_{FP}[N-1:0] \leftarrow$ Dynamic Left Shift of $\{M_{FP}, E'b0\}$ by Lshift
16:    $Exp[E:0] \leftarrow \{E_{FP}[E-1:1], E_{FP}[0]|(!(|E_{FP}))\}$ - BIAS - Lshift
17: **Posit Component Construction: Exponent** ($E_O$)**, Regime Value** ($R_O$)**, Mantissa**
    **and their Packing** ($REM$)
18:    $Exp_N[E-1:0] \leftarrow Exp[E]\ ?\ -Exp[E-1:0]\ :\ Exp[E-1:0]$
19:       **IF** $(Exp[E]\&(|Exp_N[ES-1:0]))$
20:          $E_O[ES-1:0] \leftarrow$ 2's complement of $Exp_N[ES-1:0]$
21:       **ELSE**
22:          $E_O[ES-1:0] \leftarrow Exp_N[ES-1:0]$
23:       **IF** $(!Exp[E]||(Exp[E]\&(|Exp_N[ES-1:0])))$
24:          $R_O[E-ES-1:0] \leftarrow Exp_N[E-1:ES]+1$
25:       **ELSE**
26:          $R_O[E-ES-1:0] \leftarrow Exp_N[E-1:ES]$
27:    $REM[2*N-1:0] \leftarrow \{N\{!Exp[E]\}, Exp[E], E_O, M_{FP}[N-2:ES]\}$
28:    $REM \leftarrow$ Dynamic Right Shifting by $R_O$ amount
29:    **If** $(S_{FP} == 1)$: $REM \leftarrow$ (2's complement of $REM$)
30: **Final Output**:
31:    Combine $S_{FP}$ with LSB (N-1) bit of $REM$
32:    Discharge Output while considering $INF_{FP}$ and $Z_{FP}$
```

construction. FP data extraction part works like in any FP arithmetic unit which extracts sign, exponent and mantissa part (line 8-10). It also checks for exceptional cases of ZERO and Infinity in FP operand (line 11-12). NaN (not-a-number) checking is excluded as it not a part of posit system. The FP operand also undergoes pre-normalization (to normalize the sub-normal operand, if any) (line 13-15), which requires detection of leading true bit in mantissa using a leading-one-detector (LOD) and left shifting of mantissa by this position value using a dynamic left shifter (this position value later adjusted in exponent computation). The actual value of FP exponent is computed by incorporating sub-normal status, BIAS value and mantissa left shift amount in line-16.

The posit construction part includes *component packing design challenge of Posit arithmetic*, in which position of exponent bit and mantissa varies at run-time due to run-time variation in the length of regime-bit. With a given signed exponent value (Exp) first step is to determine the regime value R_O and unsigned exponent E_O (of size ES bits) for posit (line 18-26). Basically, LSB (least significant bits) ES bits of absolute exponent Exp_N determines the E_O and remaining MSBs (most significant bits) determines R_O. If Exp is negative and $Exp_N[ES-1:0]$ is non-zero, then posit E_O would be 2's complement of $Exp_N[ES-1:0]$ (line 19-20). Since, E_O represents only unsigned integer, this procedure with an increment in corresponding negative valued R_O will take care of negative exponent representation in posit. In otherwise case, E_O would be directly taken as $Exp_N[ES-1:0]$ (line 21-22). Remaining absolute exponent MSBs $Exp_N[E-1:ES]$ would become R_O if Exp is negative and $Exp_N[ES-1:0]$ is zero. Otherwise, R_O would be an incremented value of $Exp_N[E-1:ES]$ (line 23-26).

Now from the available S_{FP}, R_O, E_O, and M_{FP}, the construction of posit word proceeds as follows:

1) S_{FP} would act as posit sign-bit.
2) A repetitive sequence of $!Exp[E]$ would act as the desired regime-bit sequence (sequence of zeros for negative exponent or sequence of ones for positive exponent). Construct a N-bit sequence of it, $N\{!Exp[E]\}$.
3) $Exp[E]$ acts as the regime sequence terminating bit.
4) Construct a N-bit word $\{!Exp[E], E_O, M_{FP}\}$. Pad necessary zero bits at the LSB side in order to complete N-bit word.
5) Combine both N-bit words (regime sequence at MSB side) to construct a 2N-bit word REM (line 27).
6) Dynamically right shifting REM by regime value amount (R_O) would insert required regime sequence in the LSB N-bit of REM (line 28). Take 2's complement of REM for negative posit sign-bit (line 29). $REM[N-1:1]$ is the desired regime, exponent and mantissa packing.

Finally, by combining posit sign-bit with REM[N-1:1], while taking care of zero and infinity cases, the equivalent posit number is obtained.

The components/statements in Algorithm-1 are presented with parameters N, E, ES and BIAS, and thus implemented, except LOD and dynamic shifters. The parameterized generation of LOD and LZD (leading zero detector, to be used in later units) are constructed in a similar hierarchical manner, except with a different respective basic 2:1 (LZD/LOD) building block (line 4-6), using parameterized Algorithm-2. For dynamic left/right shifting a parameterized barrel shifter is constructed with word width (N) and shifting amount (S) as parameter. A barrel shifter requires one N-bit 2:1 MUX for each bit of S. So, here it requires S numbers of 2:1 MUXs each of N-bit size. The parameterized generation for dynamic left shifter (DLS) is also shown in Algorithm-2, and similarly done for dynamic right shifter (DRS).

Algorithm 2 Parameterized Generation of LOD/LZD and DLS

```
1:  **LOD/LZD** #(N) (in[N-1:0], K[S-1:0], vld):
2:     N: Word Size,    S: $Log_2(N)$
3:        **GENERATE**
4:           **IF** (N == 2)
5:              For LOD:   vld = |in,    K = (!in[1]) & in[0]
6:              For LZD:   vld = !(&in),    K = in[1] & (!in[0])
7:           **ELSIF** (N & (N-1))
8:              LOD/LZD #(1«S) (1«S 1'b0 | in, K, vld)
9:           **ELSE**
10:             K_L[S-2:0], K_H[S-2:0], vld_L, vld_H
11:             LOD/LZD #(N»1) (in[(N»1)-1:0], K_L, vld_L)
12:             LOD/LZD #(N»1) (in[N-1:N»1], K_H, vld_H)
13:             vld = vld_L | vld_L
14:             K = vld_H ? {1'b0,K_H} : {vld_L,K_L}
15:       **ENDGENERATE**
16:
17: **DLS** #(N) (in[N-1:0], b[S-1:0], OUT):
18:    N: Word Size,    S: $Log_2(N)$,    TMP[S-1:0][N-1:0]
19:       TMP[0] = b[0] ? in « 1 : in;
20:       **GENVAR i**
21:       **GENERATE**
22:          **for** (i=1; i<S; i=i+1)
23:             TMP[i] = b[i] ? (TMP[i-1] « 2**i) : TMP[i-1]
24:          **end**
25:       **ENDGENERATE**
26:    OUT = TMP[S-1]
```

B. Proposed Posit to Floating Point converter

The parameterized algorithmic flow for this unit is shown in Algorithm-3. In this arithmetic unit, the posit input is first checked for zero and infinity (line 4-5). MSB of input acts as

Algorithm 3 Proposed Posit to FP Converter Flow

```
1: GIVEN: N, ES, E, BIAS (Similar to the Algorithm-1 definition)
2: Input Operand: $IN$
3: Posit Data Extraction:Sign ($S_P$), Regime ($R$), Exponent ($E_P$), Mantissa ($M_P$),
   Exceptions (Infinity ($INF_P$), Zero ($Z_P$))
4:   $Z_P \leftarrow !IN$, (All bits of IN are 0)
5:   $INF_P \leftarrow IN[N-1]\&(!IN[N-2:0])$, (Except MSB, all bits are 0)
6:   $S_P \leftarrow IN[N-1]$
7:     $XIN \leftarrow S_P\ ?\ -IN\ :\ IN$, (2's complement for -ve posit)
8:     Regime Check (RC): $RC \leftarrow XIN[N-2]$, (0 for -ve regime, 1 for +ve regime)
9:     $K0 \leftarrow$ LOD of XIN[N-2:0], (For -ve regime sequence)
10:    $K1 \leftarrow$ LZD of XIN[N-3:0], (For +ve regime sequence)
11:    Absolute Regime Value: $R \leftarrow RC\ ?\ K1\ :\ K0$
12:    Regime Left Shift Amount: $Lshift \leftarrow RC\ ?\ K1+1\ :\ K0$
13:    $XIN\_tmp[N-1:2] \leftarrow XIN[N-3:0] << Lshift$, (Dynamic left shifting)
14:      $E_P[E-1:0] \leftarrow$ XIN_tmp[N-1:N-ES]
15:      $M_P[N-1:ES-1] \leftarrow$ {|IN[N-2:0], XIN_tmp[N-ES-1:0]}
16: FP Construction:
17:   $E_0[E:0] \leftarrow RC\ ?\ \{R,E_P\}+BIAS : \{-R,E_P\}+BIAS$
18:   IF $(INF_P \mid E_O[E] \mid \&E_O[E-1:0])$:   $FP_O \leftarrow$ Infinity
19:   ELSIF $(Z_P \mid (\ M_P[N-1])$: $FP_O \leftarrow \{S_P, E-1\{1'b0\}, M_P[N-2:E]\}$
20:   ELSE  $FP_O \leftarrow \{S_P, E_O[E-1:0], M_P[N-2:E]\}$
```

posit sign bit (line 6). For -ve sign bit, a 2's complement of input posit is taken (XIN) (line 7).

The second design challenge of posit arithmetic, the posit component extraction under run-time variation in its format, is available here. It is handled by incorporating LOD, LZD and DLS components as follows.

1) XIN[N-2] acts as regime check bit (RC), which determines if it contains sequence of 0 or 1 (line 8).
2) To count a sequence of 0 with a terminating 1 (-ve regime) a LOD is applied for XIN[N-2:0]. Similarly, to count sequence on 1 with a terminating 0 (+ve regime) a LZD is applied for X[N-3:0] (1-bit less, as regime value would be one less than actual count of 1's).
3) Based on RC, effective absolute regime value R (K0 or K1) (line 11), and regime left shift amount Lshift (line 12) are determined.
4) In order to extract the exponent and mantissa, the XIN is require to dynamically left shift so as to through-out the entire regime sequence and align exponent and mantissa at MSB. XIN_tmp is obtained by dynamic left shifting of XIN[N-3:0] (line 13).
5) Here, MSB ES bits of XIN_tmp are the exponent-bit and remaining bits are the mantissa (appending hidden bit at its MSB) (line 14-15).

For the final FP construction, the actual exponent BIASed exponent (E_O) is computed based on the RC, R, E_P, and BIAS values (line 17). Finally, equivalent FP output is generated in after inclusion of zero and infinity signals. The LSB E-bit of E_O defines exponent of FP and M_P defines mantissa of FP equivalent output (line 18-20).

C. Proposed Posit Adder/Subtractor

The proposed parameterized algorithmic flow for posit addition is shown in algorithm-4. The major blocks in this include Posit Data Extraction, Core Adder Arithmetic Processing, Data Composition and Post-Processing. This same computation flow can be used for posit subtraction also, after negating the second operand.

Similar to posit data extraction in Algorithm-3, data extraction is performed on both operands IN1, IN2 (line 6-8). The core arithmetic stage involves the mantissa addition, and final exponent and regime numerical value computation. This follows from line 10 to 28. These processing are mostly analogous to FP standard except regime inclusion. First effective operation is evaluated. A comparison of XIN1 and XIN2 gives the information of large and small operand, and large/small components are thus evaluated. A decimal point alignment of both mantissa is achieved by dynamic right shifting of smaller mantissa by Ediff (the difference of effective large exponent and small exponent, by combining R and E). Based on the effective operation OP, the small shifted mantissa is added/subtracted from large mantissa LM by using a N-bit add/sub unit, and result is checked for mantissa overflow and underflow (which requires left/right shifting).

The total effective large exponent (Exp) is computed (line 27) by combining LRC, LR, Movf, and Nshift. Further, similar to line 18-26 of FP to Posit converter Algorithm-1, the E_O and R_O are computed. The regime, exponent and mantissa packing is done similar to the (line 27-29 of Algorithm-1) FP to posit converter procedure. At this stage, rounding operation is performed. The final REM is then combined by the large sign bit (LS) to produce the posit addition result, while considering ZERO and Infinity check of the input operands.

Algorithm 4 Proposed Posit Adder Computational Flow

```
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: Posit Data Extraction → Effective Operand (XIN), Sign (S), Regime Check
   (RC), Regime (R), Exponent (E), Mantissa (M), Infinity (Inf), Zero (Z):Performed
   similar to data extraction of posit in Algorithm-3
7:   Extraction from IN1 → XIN1, S1, R1, E1, M1, Inf1, Z1
8:   Extraction from IN2 → XIN2, S2, R2, E2, M2, Inf2, Z2
9:   $Z \leftarrow Z1\&Z2$          $Inf \leftarrow Inf1|Inf2$
10: Core Adder Arithmetic Processing:
11:   Effective Operation: OP ← S1 xor S2
12:   Large Operand: IN1_gt_IN2 ←XIN1[N-2:0]>XIN2[N-2:0] ? 1 : 0
13:     Large (L) Component: LS, LRC, LR, LE, and LM
14:     Small (S) Component: SS, SRC, SR, SE, and SM
15: MANTISSA ADDITION:
16: Effective Exponent Difference (Ediff):
17: $Ediff \leftarrow ((LRC?LR-(SRC?SR:-SR):SR-LR) << ES)+LE-SE$
18: $SM_T \leftarrow$ Dynamic Right Shift SM by Ediff
19: Add LM and $SM_T$:
20:   $Add_M \leftarrow OP\ ?\ LM+SM_T\ :\ LM-SM_T$
21:   Mantissa Overflowed: $Movf \leftarrow Add_M[MSB]$
22:   $Add_M \leftarrow Movf\ ?\ Add_M\ :\ Add_M << 1$
23: Normalization of $Add_M$:
24:   Nshift ← LOD of $Add_M$
25:   $Add_M \leftarrow Add_M << Nshift$, (Dynamic Left Shifting by Nshift)
26: Final EXPONENT ($E_O$) and REGIME ($R_O$) Computation:
27:   $Exp \leftarrow \{(LRC\ ?\ LR\ :\ -LR), LE\}+Movf-Nshift$
28:   $E_O$ and $R_O$: Computed as Line 18-26 of Algorithm-1
29: Data Composition and Post-Processing:
30:   Regime, Exp & Mantissa Packing (REM): Computed similar to Line 27-29 of
   Algorithm-1 while using $Exp$, $R_O$, $Add_M$.
31:   Rounding: Round-to-zero (Truncation)
32: Final Output: Combine LS with LSB (N-1) bit of rounded REM, while considering
   Exceptions
```

D. Proposed Posit Multiplier

Basically, in the core of posit multiplier unit it requires to multiply both mantissas and sum their effective exponents. The proposed algorithmic flow for posit multiplier is shown in algorithm-5. Most components and their descriptions in this unit are similar to the Adder arithmetic flow. Line 3-6 extract data same as adder unit. Line 7 determined the actual (+ve / -ve) of regime (RG1 and RG2) using respective regime check bits (RC1 and RC2), to be used for operands exponent

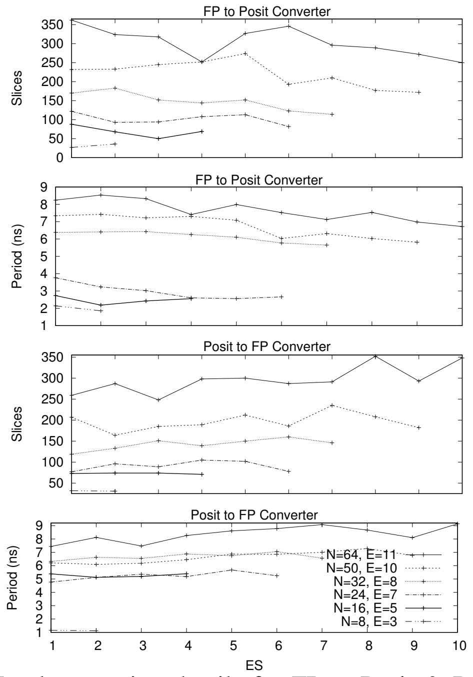

Fig. 1: Implementation details for FP to Posit & Posit to FP.

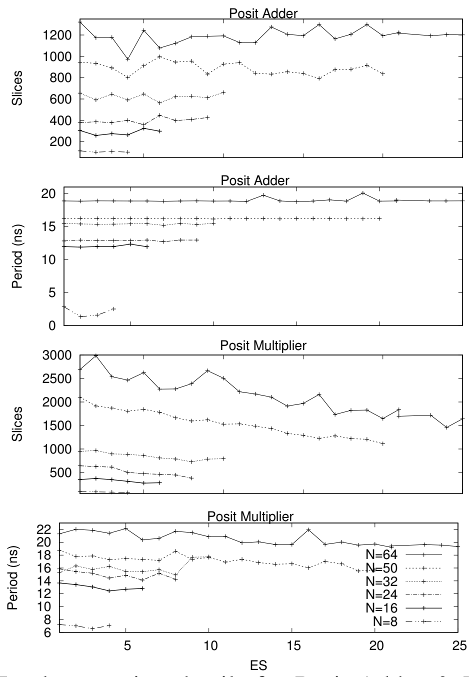

Fig. 2: Implementation details for Posit Adder & Multiplier.

sum processing. The mantissa multiplication processing is performed on lines 8-11, where both mantissas are multiplied using integer multiplier, which result later checked for overflow and then shifted by 1-bit accordingly. The exponent sum is performed at line 13 which sums up effective exponents (by combing regime and exponent-bits) of both operands. All the remaining processing goes similar to the posit adder unit.

Algorithm 5 Proposed Posit Multiplier Computational Flow

```
1: GIVEN: N, ES, RS (Similar to the Adder Algorithm-4 definition)
2: Input Operands: $IN1$, $IN2$
3: Posit Data Extraction: $\rightarrow$ Similar to data extraction in Algorithm-3,4
4:    IN1 $\rightarrow$ XIN1, S1, R1, E1, M1, Inf1, Z1
5:    IN2 $\rightarrow$ XIN2, S2, R2, E2, M2, Inf2, Z2
6:    $Z \leftarrow Z1 \& Z2$          $Inf \leftarrow Inf1 | Inf2$
7:    $RG1 \leftarrow RC1 \ ? \ R1 \ : \ -R1$,      $RG2 \leftarrow RC2 \ ? \ R2 \ : \ -R2$
8: Mantissa Multiplier Arithmetic Processing:
9:    $M \leftarrow M1 * M2$
10:   $Movf \leftarrow M[MSB]$
11:   $M \leftarrow Movf \ ? \ M \ : \ M << 1$
12: Final EXPONENT ($E_O$) and REGIME ($R_O$) Computation:
13:   $Exp \leftarrow \{RG1, E1\} + \{RG2, E2\} + Movf$
14:   $E_O$ and $R_O$: Similar to Algorithm-1,4
15: Data Composition, Rounding and Final Output: Similar to Algorithm-1,4
```

III. Implementation Results

A single cycle implementation of proposed algorithmic flow of all posit arithmetic is implemented on a Xilinx Virtex-6 FPGA device. All the modules are parameterized with N, ES, E, and RS. The functional verification are done against the Julia package for posit [4] provided by the posit developers. The implementation details of posit arithmetic architectures are shown in graphical format with variations of N (8, 16, 24, 32, 50 and 64) and ES in Figs. 1 and 2. These units are implemented with two levels of registers at the input and output port to avoid any IO port delay, which added some unaccounted resources to these units. Here, to get a clear idea on variations in resource utilization, only logics are used for multiplier implementation, instead of DSP48 IPs.

All the proposed posit arithmetic units are parameterized for N, ES, and E, which can generate hardware for any desired value. The source code of these modules will be provided as an open-source material which can be find at [5].

IV. Conclusions

This paper addressed the architectural algorithmic development of posit arithmetic units and their implementation on FPGA platform. Posit is an exciting inclusion in the recent development of Unum theory. The arithmetic unit of FP to posit converter, posit to FP converter, adder and multiplier units are explored here. All units are developed and implemented with parameterized components. Also, this work is being made open-source for an easy access to community. This would facilitate researchers to further investigate and explore on posit arithmetic in different applications.

V. Acknowledgments

This work was supported in part by the Research Grants Council of Hong Kong (Project GRF 17245716), and the Croucher Foundation (Croucher Innovation Award 2013).

References

[1] Gustafson, John L. and Yonemoto, Isaac, "Beating Floating Point at its Own Game: Posit Arithmetic," pp. 1–16. [Online]. Available: http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

[2] John L. Gustafson. 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

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

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

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