IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 10, 2019, accepted May 28, 2019, date of publication June 5, 2019, date of current version June 20, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2920936

PACoGen: A Hardware Posit Arithmetic Core Generator

MANISH KUMAR JAISWAL, (Member, IEEE), AND HAYDEN K.-H. SO, (Senior Member, IEEE)
Department of Electrical and Electronics Engineering, The University of Hong Kong, Hong Kong

Corresponding author: Manish Kumar Jaiswal (manishkj@hku.hk)

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

ABSTRACT This paper proposes open-source hardware **P**osit **A**rithmetic **Co**re **Gen**erator (PACoGen) for the recently developed universal number posit number system, along with a set of pipelined architectures. The posit number system composed of a run-time varying exponent component, which is defined by a composition of varying length "regime-bit" and "exponent-bit" (with a maximum size of ES bits, the exponent size). This in effect also makes the fraction part to vary at run-time in size and position. These run-time variations inherit an interesting hardware design challenge for posit arithmetic architectures. The posit number system, being at an infant stage of its development, possess very limited hardware solutions for its arithmetic architectures. In this view, this paper targets the algorithmic development and generic HDL generators (PACoGen) for basic posit arithmetic. The proposed open source PACoGen currently includes the adder/subtractor, multiplier, and division arithmetic. The PACoGen can provide the Verilog HDL code respective posit arithmetic for any given posit word width (N) and exponent size (ES), as defined under the posit number system. Further, pipelined architectures of 32-bit posit with 6-bit exponent size are also proposed and discussed for addition/subtraction, multiplication, and division arithmetic. The proposed posit arithmetic architectures are demonstrated on the Virtex-7 (xc7vx330t-3ffg1157) FPGA device as well as Nangate 15 nm ASIC platform. The PACoGen would open a gateway for further posit arithmetic hardware exploration and evaluation.

INDEX TERMS Adder, ASIC, digital arithmetic, division, floating point arithmetic, FPGA, multiplier, posit arithmetic, subtractor, universal number system.

I. INTRODUCTION

The universal number (unum) system was first proposed in 2015 and has undergone a number of progressive evolution since then as type-1 [1]–[3], type-2 [4], [5], and type-3 [6], [7]. Posit is the part of latest revision, the type-3 unum. Unum is claimed as a possible substitute for floating point (FP) number system [8]. Interesting, type-3 unum, the posit, format is more closer to FP representation than type-1 and type-2 unum. As claimed and shown by its developers', posit provides many benefits over FP standard, including better dynamic range and accuracy over same bit field, more accurate and exact arithmetic computations [6].

These proposals have created a significant amount of interest in the community. As a result a range of software tools are developed for unum using Julia, C, C++ etc. However, very limited work has been aimed for hardware solutions. This paper is aimed towards an open-source hardware **P**osit **A**rithmetic **Co**re **Gen**erator (PACoGen) for recently developed posit number system under the unum umbrella. Currently, it is focused on basic arithmetic of posit addition/subtraction, multiplication and division arithmetic.

$$\overbrace{s}^{Sign}\ \underbrace{\overbrace{r\,r\cdots r\,\bar{r}}^{Regime-bit}}_{Run-Length}\ \underbrace{\overbrace{e_1\,e_2\cdots e_{es}}^{Exponent-bit,\ if\ any}}_{Max.\ ES-Bit}\ \underbrace{\overbrace{f_1\,f_2\ \cdots\cdots.}^{Mantissa-bit,\ if\ any}}_{Remaining-Bit,\ if\ any} \tag{1}$$

A posit format is defined as a combination of its word size (N) and exponent size (ES), as (N,ES). As shown in (1), the components in posit format (regime, exponent and mantissa) varied at run-time which provides various options of dynamic range and accuracy/precision. As claimed and shown (with detailed demonstration on several example cases) by the posit developers [6], [9], these available choices in posit would be beneficial for a variety of applications having different set of requirements on dynamic range and

The associate editor coordinating the review of this manuscript and approving it for publication was Gian Domenico Licciardo.

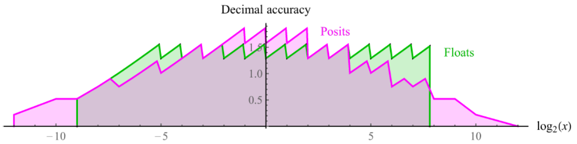

FIGURE 1. Decimal accuracy of 8-bit posit vs 8-bit FP.

accuracy, while providing significant benefits over floating point standard. Posit provide several key benefits over FP standard, including

- Better dynamic range: for a similar word size, posit provide more wider dynamic range than FP.
- Posit provide more accurate and exact arithmetic computations than FP.
- Posit does not Overflow/Underflow, and thus does not lose all the information.
- Posit defines only one `ZERO`, one `Infinity` and no `Not-a-Number (NaN)`. The `Infinity` representation can also be used for NaN outcome. By not including NaN, with a M-bit FP mantissa, posit saves on $2^{M+1} - 2$ invalid (NaN) combinations.

A brief analysis of 8-bit floating point and posit format is shown below along with their accuracy variation in Fig. 1 (taken from [9])

Floating Point (1,4,3)	Posit (with ES=1)
Range: 1/1024 to 240	1/4096 to 4096
Dynamic range of 5.1 decades	Dynamic range of 7.2 decades
14 NaN Combinations	No NaN Combinations
Tapered Accuracy	Symmetric Accuracy
	(More Accuracy around 1.0)

While posit provides a range of qualitative numerical benefits over FP standard, the run-time variation in its component provide an interesting hardware design challenge. 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. It is discussed in all related sections how these are handled during architectural development.

Earlier related hardware solutions on posit includes [10]–[13]. The initial work on posit hardware research [10], [13] are proposed by the authors which briefly includes the primary on posit arithmetic generators. Following these initial work, [11] has proposed a fixed width (32-bit) posit adder and multiplier architectures, and [12] has briefly presented a (proprietary) generator for posit adder and multiplier. These literature has briefly laid down the initial findings on the posit hardware work. Following these literature, the current manuscript is aimed as a detailed, improved as well as augmented proposal on posit arithmetic core generator, named as PACoGen.

This manuscript is build upon the work presented in [10] which has briefly discussed the HDL generators for posit adder and multiplier on a FPGA platform. The current manuscript includes a much wider and minute explanation of the PACoGen for adder and multiplier along with several improvements over earlier work. The current manuscript also includes the posit HDL generator for division arithmetic based on Newton-Raphson method [14]–[16]. Our future work on posit division generator will also be focused on including other division methods (digit recurrence, digit convergence, SRT methods). The current manuscript also presents a pipelined version of posit adder, multiplier and divider arithmetic architecture for a specific posit configuration for performance improvement. It also demonstrated the implementation results on Nangate 15nm ASIC [17] as well as Virtex-7 FPGA device platform.

The prime contributions of current manuscript can be summarized as follows:

- Algorithmic flows for posit arithmetic architectures are proposed for addition/subtraction, multiplication and division (with Newton-Raphson method).
- An open source parameterized Verilog HDL generator for posit adder/subtractor, multiplier and division (with Newton-Raphson method) arithmetic is proposed.
- Implementations are demonstrated on a Virtex-7 FPGA device as well as 15nm ASIC platform over a range of word-size with varying exponent size (ES).
- A pipelined architecture for these arithmetic are also proposed and discussed for 32-bit posit with 6-bit exponent size.

II. BACKGROUND: THE POSIT NUMBER SYSTEM

Here a brief introduction of posit number system is presented. A detailed information of posit number system can be found at [6]. A posit number is defined by its exponent size (ES) for a given word size (N). Its generic format is shown in (1). Posit format has following features.

- Posit format has only two reserved/special representation, one for ZERO and another for infinity.
- The ZERO is represented by all bits with zero ‘0’ value (000 . . . .000).
- The infinity is representation by all bits, but sign-bit, as ‘0’ value (100 . . . 000).
- The posit format does not provide any separate/special representation for Not-a-Number (NaN).
- It also does not consider sub-normal or denormal representation. Thus, posits are always a normalized number.
- The posit format consists of four field: Sign-bit, Regime-bits, Exponent-Bits and Mantissa-Bits.
- Sign-Bit:
    - is 0 for positive posit numbers.
    - is 1 for negative posit numbers. Further, for negative number, it requires to first take a two’s complement of remaining field before decoding the regime, exponent and mantissa bits.
- The total exponent value in posit is defined by the combination of regime-bits and exponent-bits.
- Regime Bits:

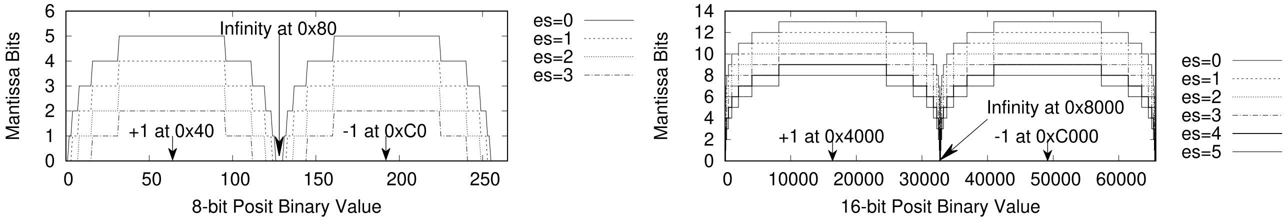

FIGURE 2. Variation of mantissa size with respect to exponent size (es) in 8 and 16 bits posit format.

  - Regime bits are a sequence of either all 0 or 1, terminated by an opposite bit. Its run-length determines the value of regime-bit.
  - A string of m-bit 0 terminated by (m+1)th bit as 1 gives a value of -m (01 → -1), (0001 → -3)
  - A string of m-bit 1 terminated by (m+1)th bit as 0 gives a value of m-1 (10 → 0), (111110 → 4)
  - With regime value of k, its contribution in total exponent value is $(2^{(2^{ES})})^k$
- Exponent Bits:
  - Exponents bits are unsigned integer which can be maximum of ES (exponent size) bits in length. The value of ES defined the format of posit number.
  - With value of e, its contribution in total exponent is 2^e.
- Mantissa Bits:
  - It function similar to the normalized floating point standard and any remaining bits (if available) after regime and exponent bits are occupied by it.
  - For a given ES value, the maximum mantissa width can be N-ES-3.
- With a regime value of k, exponent value of e and mantissa value of f (including hidden bit 1), the equivalent decimal value of a posit would be

$$s * (2^{(2^{ES})})^k * 2^e * f \tag{2}$$

- In case, width of exponent part is less than ES, the available portion will be treated as MSB of exponent portion. For example, with N = 8 and ES = 4, for posit number 00000111 → $0_{0}0001_{1}1$, sign is 0, regime sequence is 00001 and only 2-bit of exponent 11 (for ES = 4) is available. Here, the actual exponent will be 1100, after appending remaining LSBs with 0.

Let us go through few examples for decimal equivalent of 8-bit posit (P). The sign, regime, exponent and mantissa field are separated (by underscore) for easy understanding, and negative values are first converted into 2's complement before decoding.

$$
\begin{array}{llll}
WithES=2: & \text{Sign Regime} & \text{Exp} & \text{Mantissa} \\
0\_0001\_11\_1 & =+(2^{(2^2)})^{-3} & *2^3 & *(1+\frac{1.0}{2}) \\
0\_110\_10\_11 & =+(2^{(2^2)})^{1} & *2^2 & *(1+\frac{3.0}{4}) \\
10001111 & & & \\
\underset{\rightarrow}{2'c}\ 0\_1110\_00\_1 & =-(2^{(2^2)})^{2} & *2^0 & *(1+\frac{1.0}{2}) \\
11101011 & & & \\
\underset{\rightarrow}{2'c}\ 0\_001\_01\_01 & =-(2^{(2^2)})^{-2} & *2^1 & *(1+\frac{1.0}{4}) \\
WithES=3: & & & \\
0\_00001\_11 & =+(2^{(2^3)})^{-4} & *2^6 & *(1+0.0) \\
0\_110\_101\_1 & =+(2^{(2^3)})^{1} & *2^5 & *(1+\frac{1.0}{2}) \\
10001111 & & & \\
\underset{\rightarrow}{2'c}\ 0\_1110\_001 & =-(2^{(2^3)})^{2} & *2^1 & *(1+0.0) \\
11101011 & & & \\
\underset{\rightarrow}{2'c}\ 0\_001\_010\_1 & =-(2^{(2^3)})^{-2} & *2^2 & *(1+\frac{1.0}{2})
\end{array}
\tag{3}
$$

As mentioned above that for a given value of ES, the mantissa width also varies at run-time. A graphical representation of this variation is shown in Figs 2 for N = 8 and N = 16 posit. It can be seen that posit representation founds more fraction bits around ±1.For a given ES, the maximum mantissa width could be N-ES-3 (where, value 3 accounts for 1 sign bit and minimum 2 bits for regime 01 or 10).

III. PACOGEN: POSIT ARITHMETIC CORE GENERATOR

In this section a detailed description of posit arithmetic HDL generators are presented. HDL generators for adder/subtractor, multiplier and division arithmetic is proposed in this paper. These are parameterized for any posit word size (N) and exponent size (ES). The basic computation flow for these arithmetic is presented in Fig. 3. It starts with the posit data extraction, then performs the core arithmetic processing related to given arithmetic, followed by the posit construction, rounding and final processing. The details regarding each of these steps are discussed in each of arithmetic section description.

A. ADDER/SUBTRACTOR HDL GENERATOR

The proposed parameterized algorithmic computational flow for posit addition/subtraction is partitioned into 3 segments, as shown in Fig. 3. These include posit data extraction segment which extracts the sign, regime-bit, exponent-bit, and

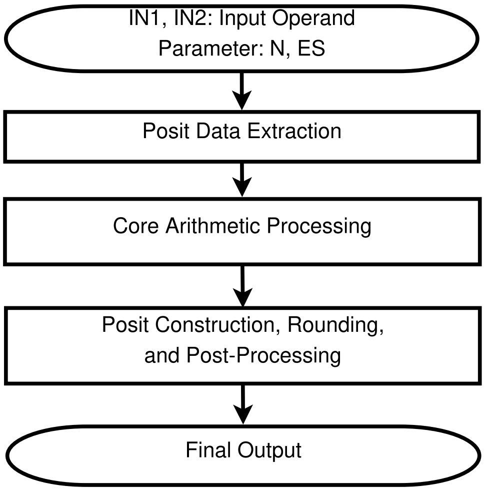
IN1, IN2: Input Operand
Parameter: N, ES
Posit Data Extraction
Core Arithmetic Processing
Posit Construction, Rounding, and Post-Processing
Final Output

FIGURE 3. Basic posit arithmetic flow.

mantissa information from the input operands, core adder arithmetic processing for mantissa addition and effective exponent value computation and finally followed by the posit data composition, rounding and post-processing which combine sign, regime, exponent, mantissa. This same computation flow is used for posit subtraction also, after taking a two's complement of second operand. So, for further discussion, we will just refer to addition/adder only.

1) POSIT DATA EXTRACTION

The posit data extraction flow is presented in Algorithm-1. 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 maximum width of regime sequence can be of (N-1) bits, RS bits (= $Log_2\ N$) can store its maximum absolute numerical value.

The Posit extraction algorithm first check the input operands for exceptional cases (ZERO and INFINITY) (Line 7 to 13 in Algorithm-1). All bits with 0 leads to a zero posit representation, however, all bits with 0 except a true sign bit leads to infinity posit representation.

The MSB of posit operands provide the respective sign-bits (Line 15). For negative posit operands (if true sign bit), it would undergo a 2's complement conversion, except the sign bit, and produces transformed XIN1 and XIN2 operands.

Algorithm 1 Proposed Posit Data Extraction Flow

```
1: GIVEN:
2:   N: Posit Word Size
3:   ES: Posit Exponent Size
4:   RS: $log_2\ (N)$ (Posit Regime Value Storage Size)
5: Input Operands: $IN1,\ IN2$
6: Data Extraction:Sign (S), Regime Value (R), Exponent (E), Mantissa (M), Exceptions (Infinity (Inf), Zero (Z))
7:   Check for ZEROs
8:     $Z1 = |IN1,\ \ Z2 = |IN2,$     (if all bits of IN1, IN2 are 0)
9:     $Z \leftarrow Z1\&Z2$
10:  Check for Infinity's
11:    $Inf1 = IN1[N-1]\&|IN1[N-2:0],$     (if all bits, except Sign-bit, are 0)
12:    $Inf2 = IN2[N-1]\&|IN2[N-2:0]$
13:    $Inf \leftarrow Inf1|Inf2$
14:  Extraction from IN1:
15:    S1← IN1[N-1]     (Sign-Bit)
16:    XIN1[N-2:0] ← S1 ? −IN1[N-2:0] : IN1[N-2:0]     (2's Complement if −ve)
17:    RC1 ← XIN1[N-2]     (Regime Check Bit)
18:    XIN1_tmp ← RC1 ? !(XIN1) : XIN1     (1's complement if RC1 = 1)
19:    R ← Leading One Detection (XIN1_tmp[N-2:0])
20:    R1 ← RC1 ? R-1 : R     (Effective Regime Value)
21:    XIN1_tmp ← XIN1_tmp << R     (Flush out regime sequence)
22:    E1 ← MSB ES-bits of XIN1_tmp     (Exponent)
23:    M1 ← Remaining bits of XIN1_tmp     (Mantissa, Append Hidden Bit)
24:  Extraction from IN2: → S2, R2, E2, M2
```

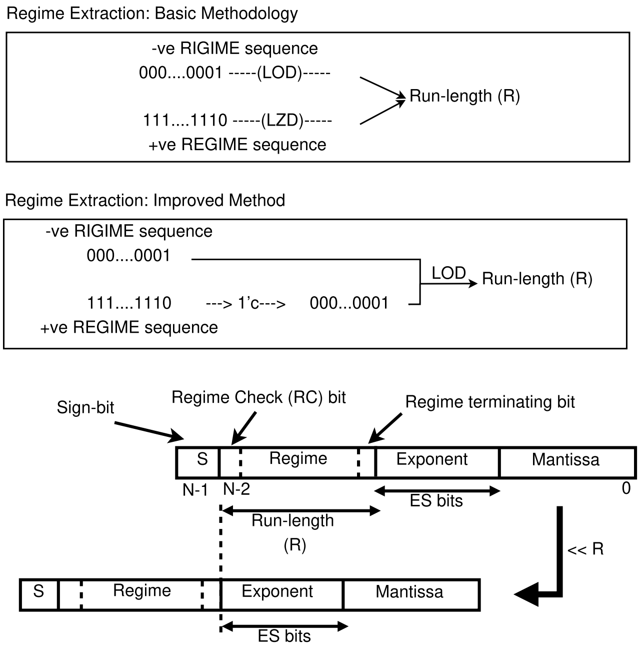

FIGURE 4. Posit data extraction.

The posit data extraction idea can be sought from Fig. 4. The MSB of XIN operand act as the regime check (RC) bit: 0 for negative regime sequence and, 1 for positive regime sequence. Now, for regime sequence detection it requires to find out the position of terminating regime bit. As shown in Fig. 4, primary idea is to use a leading one detector (LOD) to detect the terminating 1 in a sequence of 0s (000...0001) and use a leading zero detector (LZD) to detect the terminating 0 in a sequence of 1s (111...1110). However, if the presence of a sequence of 111...1110 complemented (1's complement), then the use of a single LOD would suffice the purpose of finding regime sequence terminating bit. This is shown in Fig. 4 as regime extraction improved method, and this process provides the run-length of regime sequence (R). The effective absolute regime value would be R for a sequence of 0 (−ve regime sequence), and R-1 for a

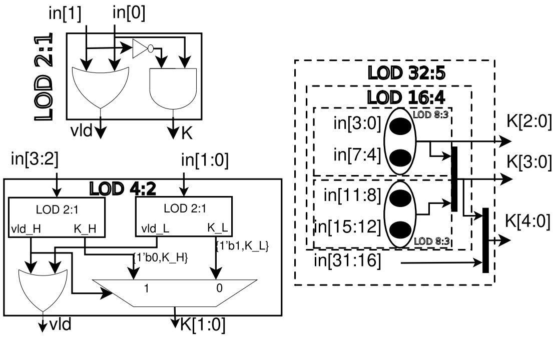

FIGURE 5. LOD architectural construction.

sequence of 1 (+ve regime sequence). These are shown in Lines 19-20 of Algorithm-1.

The parameterized Verilog based generation of leading one detector (LOD) is shown in Algorithm-2. As shown in Fig. 5, it is generated and constructed in a hierarchical manner. It is based on the building up larger size LOD using 2:1 LOD.

Algorithm 2 Parameterized Generation of Leading One Detector (LOD)

```
LOD #(N) (in[N-1:0], K[S-1:0], vld):
  N: Word Size,   S: $Log_2(N)$
    GENERATE
      IF (N == 2)
        vld = |(in),    K = !in[1] & in[0]
      ELSIF (N & (N-1))
        LOD #(1<<S) ({1<<S {1'b0}} | in, K, vld)
      ELSE
        K_L[S-2:0], K_H[S-2:0], vld_L, vld_H
        LOD #(N>>1) (in[(N>>1)-1:0], K_L, vld_L)
        LOD #(N>>1) (in[N-1:N>>1], K_H, vld_H)
        vld = vld_L | vld_L
        K = vld_H ? {1'b0,K_H} : {vld_L,K_L}
    ENDGENERATE
```

After finding the regime run-length value, the regime sequence is flushed out of XIN_tmp operand by left shifting it by R amount with the help of a dynamic left shifter. This process aligned the exponent and mantissa combination at the MSB of XIN_tmp. As the length of exponent is defined (ES bits) and its position is know at this stage of computation, the exponent and mantissa (the remaining portion) are easily extracted as in Lines 22-23 of Algorithm-1. The maximum bit-width of mantissa can be N-ES-2.

The parameterized Verilog based generation and construction of dynamic left shifter (DLS) is presented in Algorithm-3. It is based on the barrel shifter and parameterized with word size N and shifting amount S. It requires one N-bit 2:1 MUX for each bit of S. Thus, here it would need RS numbers of 2:1 MUXs each of (N-1) bit size.

Thus, posit extraction unit extract the sign bits, effective absolute regime values, exponents value and mantissa from respective posit operands and passes (XIN1,S1,RC1,R1,E1,M1 and XIN2,S2,RC2,R2,E2,M2) to next unit for further processing.

Algorithm 3 Parameterized Generation of Dynamic Left Shifter (DLS)

```
DLS #(N) (in[N-1:0], b[S-1:0], OUT):
  N: Word Size,   S: $Log_2(N)$,   TMP[S-1:0][N-1:0]
    TMP[0] = b[0] ? in << 1 : in;
    GENVAR i
    GENERATE
      for (i=1; i<S; i=i+1)
        TMP[i] = b[i] ? (TMP[i-1] << 2**i) : TMP[i-1]
      end
    ENDGENERATE
  OUT = TMP[S-1]
```

Algorithm 4 Posit Core Adder Arithmetic 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:   S1,RC1,R1,E1,M1,and S2,RC2,R2,E2,M2: Operands Extracted Components
6: Core Adder Arithmetic Processing:
7:   OP ← S1 xor S2                    (Effective Operation)
8:   Check for Large Operand and Small Operand
9:     IN1_gt_IN2 ← XIN1 >> XIN2 ? 1 : 0    (Operands Comparison)
10:    Large (L) Component: LS, LRC, LR, LE, and LM
11:    Small (S) Component: SS, SRC, SR, SE, and SM
12: MANTISSA ADDITION:
13:   Effective Exponent Difference (Ediff):
14:     Ediff ← {LRC ? LR : -LR, LE}-{SRC ? SR : -SR, SE}
15:   SM_tmp ← SM >> Ediff (Right Shift SM by Ediff amount)
16:   Add LM and SM_tmp:          (Mantissa Addition/Subtraction)
17:     Add_M → OP ? LM + SM_tmp : LM - SM_tmp
18:     Movf ← Add_M[MSB]              (Mantissa Overflow)
19:     Add_M ← Movf ? Add_M : Add_M << 1
20:   Normalization of Add_M:
21:     Nshift ← LOD of Add_M          (Check for Leading-One)
22:     Add_M ← Add_M << Nshift, (Dynamic Left Shifting by Nshift)
23: Final EXPONENT (E_O) and REGIME (R_O) Computation:
24:   LE_O ← Combine LR, LE, Movf and Nshift
25:   E_O: ← Based on +/− of LE_O, Compute LSB ES bits from LE_O
26:   R_O: ← Based on +/− of LE_O, Compute MSB RS bits from LE_O
```

2) CORE ADDITION ARITHMETIC

This unit mostly work like a analogous floating point adder arithmetic unit. In this stage the core arithmetic of mantissa addition/subtraction, and resultant exponent and regime value computation are processed. The effective arithmetic operation (addition/subtraction) between both mantissa operands are find by a XOR logic operation between both sign bits (Line 7 algorithm'4). To perform the mantissa addition/subtraction, large and small operands need to be find out out (Line 8-11). A direct comparison of XIN1 and XIN2 gives the information of large and small operand. It requires a (N-1) bits greater-than-equal-to comparison component. Based on this comparison the large and small components of sign-bit, regime-check bit, regime, exponent and mantissa are computed (LS, SS for sign; LRC, SRC for regime-check bit; LR, SR for regime; LE, SE for exponent; and LM, SM for mantissa). All of these require 2:1 MUXs of respective component bit-width.

In order to perform mantissa arithmetic a decimal alignment of mantissa operands are required. To achieve this, smaller mantissa is required to be dynamically right shifted by the difference of effective total value of large exponent and small exponent (Ediff). Ediff[BS:0] is computed by combining the difference of effective regime values (by considering their signs, which then shifted left by ES bits), and exponent differences (Line 14). The small mantissa is shifted by Ediff amount by a dynamic right shifter, and produces SM_tmp (Line 15). The dynamic right shifter is generated using the Algorithm-5. It is a parameterized dynamic right shifter designed using barrel right shifter. Its functioning is very similar to the dynamic left shifter except for the shifting direction.

Algorithm 5 Parameterized Generation of Dynamic Right Shifter (DRS)

```
DLS #(N) (in[N-1:0], b[S-1:0], OUT):
  N: Word Size,    S: $Log_2(N)$,    TMP[S-1:0][N-1:0]
    TMP[0] = b[0] ? in >> 1 : in;
    GENVAR i
    GENERATE
      for (i = 1; i < S; i = i + 1)
        TMP[i] = b[i] ? (TMP[i-1] >> 2**i) : TMP[i-1]
      end
    ENDGENERATE
  OUT = TMP[S-1]
```

The addition/subtraction of shifted small mantissa and large mantissa is carried out using a (N-ES-2) bits add/sub unit and produces Add_M (Line 17). For the case of effective addition operation, any mantissa overflow (Movf) is detected by its MSB-bit and mantissa is accordingly shifted by 1-bit to the left, which requires an N-ES-2 bits 2:1 MUX (Line 18-19). The final total exponent value later incremented by one for true mantissa overflow. For an effective mantissa subtraction operation, if LM and SM_tmp consists of very close values, then Add_M may lose some MSB bits. In this case a normalization of mantissa is required. This requires a leading one detection (LOD) operation on Add_M and further dynamic left shifting. It requires a N-ES-1 bit LOD to get normalization shift (Nshift[RS-1:0]) amount (Line 21), and then process the dynamic left shifting (Line 22). Nshift amount is later adjusted in final exponent computation. The parameterized LOD and Dynamic left shift architecture generation is already discussed above.

After above processing the computation of final regime numerical value (R_O) and exponent (E_O) performed.

The concept of the regime value (R_O) and ES-bit exponent (E_O) computation from a total effective exponent (EXP_O) is presented in Algorithm-6. The computed resume value is later converted appropriate sequence of either 0 or 1, based on the sign of total effective exponent.

Algorithm 6 Concept of Regime Value (R_O) and ES-bit Exponent Value (E_O) Computation From Total Effective Exponent (EXP_O)

```
GIVEN
  EXP_O[E:0]: Total effective exponent value, to be converted
  in posit format, the regime value and ES-bit exponent value
  combination
IF (EXP_O >= 0)
    E_O = EXP_O[ES-1:0]
    R_O = EXP_O[E:ES]
ELSE
  EXP_O ← 2's complement of EXP_O
  IF (EXP_O[ES-1:0] == 0)
    E_O = EXP_O[ES-1:0]
    R_O = EXP_O[E:ES]
  ELSE
    E_O = 2's complement of EXP_O[ES-1:0]
    R_O = EXP_O[E:ES] + 1
```

For the posit adder these computations are shown below in Algorithm-7, in a hardware friendly format. First, an effective exponent output (EXP_O) is computed by combining absolute large regime value (LR) and large exponent (LE), and then adjusting it for mantissa addition overflow (Movf) and mantissa subtraction underflow amount (Nshift). The LE_O would be of (ES + RS + 1) bit-width. If EXP_O is less than 0, it is negated as EXP_ON. Based on the EXP_O and EXP_ON, the final regime R_O and final exponent E_O are computed in lines 3-5.

Algorithm 7 Computation of Final Regime and Exponent

```
1: EXP_O = {(LRC ? LR : -LR),LE} + Movf - Nshift
2: EXP_ON = EXP_O[ES + RS] ? -EXP_O : EXP_O
3: E_O = (EXP_O[ES + RS] & (|EXP_ON[ES-
   1:0])) ? EXP_O[ES-1:0] : EXP_ON[ES − 1:0]
4: R_O = !(EXP_O[ES + RS]) | (EXP_O[ES + RS] &
   (|EXP_ON[ES-1:0]))
5:          ? EXP_ON[ES + RS − 1:ES] + 1'b1 : EXP_ON[ES
   + RS − 1:ES]
```

In this unit final sign-bit (LS), regime (R_O), exponent (E_O) and mantissa (ADD_M) is processed, which are combined in next unit for further processing.

3) POSIT DATA COMPOSITION AND ROUNDING

This unit deals with the posit composition, rounding and final processing. The strategy for posit packing is shown in Fig .5 and in Algorithm-8. It is carried out using a 2*N + 3 bit REM data construction. In this the MSB N-bit are a sequence of the complement of the sign-bit of total exponent output (LE_O). It meant for creating the desired regime sequence. Since regime consists of a sequence of 1 for positive exponent and a sequence of 0 for negative sequence, a sequence of complemented sign-bit of actual exponent would act as the desired regime sequence. Also, as the regime termination bit is the opposite of regime sequence bits, thus, the same sign-bit would act as the regime sequence termination bit, which is kept just after the N-bit regime sequence.

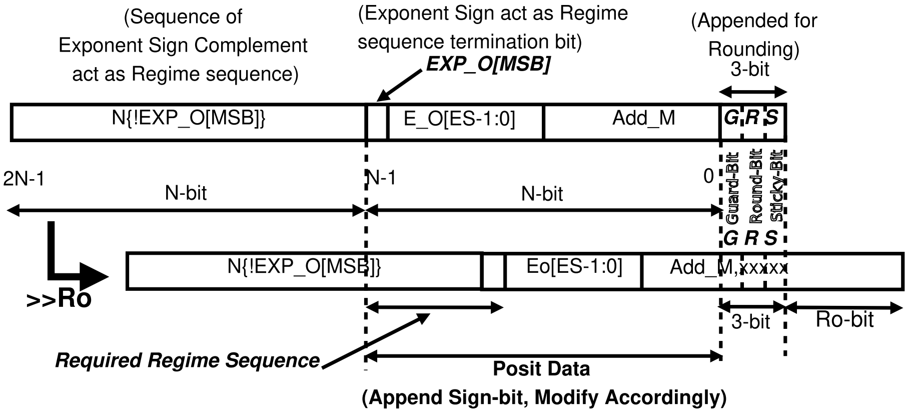

FIGURE 6. Posit data construction.

Algorithm 8 Posit Construction, Rounding and Final Processing 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:    LS, R_O, LE_O, E_O, Add_M
 6: Input Operands: $IN1,\ IN2$
 7: Posit Data Composition:
 8: REGIME, EXPONENT and MANTISSA Packing:
 9:    REM ← $\overbrace{N\{!EXP\_O[MSB]\}, EXP\_O[MSB]}^{N+1\ Bits\ Regime\ Sequence},$
       $\overbrace{E\_O, Add\_M[N-ES-2:0]}^{Exponent\ -\ Mantissa},\ \ \overbrace{3'b0}^{GRS-Bit}$
10:    REM ← REM >> R_O
11: Rounding: Round to nearest even
12:    IF (R_O < N-ES-2)
13:       ULP_add = G.(R + S) + L.G.(!(R + S))
14:       REM ← REM + {(N-1)'b0,ULP_add}
15: REM ← (LS == 0) ?  REM :  2's Complement of REM
16: Final Output:
17:    Combine LS with MSB (N − 1) bit of rounded REM
18:    Discharge Output while considering Exceptions
```

Next N-1 bits are consists of ES bits of exponent output (E_O) followed by the mantissa (ADD_M, after leaving the hidden 1 bit). Finally, a 3 bits of 0 is appended at LSB side for Guard (G), Round (R) and Sticky (S) bits for rounding purpose (Line 9).

Now, the desired composition of posit is obtained by dynamic right shifting of above constructed REM by absolute regime output value (R_O) (Line 10). The shifted R_O bits at the LSB will contribute to the sticky-bit for rounding purpose (shown in Fig. 6).

After above processing rounding is performed (Lines 11-14). The posit format talks only of one rounding method, round-to-nearest-even method. Rounding is performed by generating the ULP-Add bit (Unit at last place addition bit) by the combination of mantissa precision bit position (L), the next-bit as Guard-bit (G), the next-bit as round-bit (R) and OR of remaining all bits as sticky-bit (S). The ULP-add bit is computed using few gates by following relation: $ULP\_add\ =\ G.(R\ +\ S)\ +\ L.G.(!(R\ +\ S))$. The ULP_add bit is added at precision bit of the posit mantissa. This will provide the required combination of regime sequence, exponent and rounded mantissa. It is to be noted that rounding addition is required only when R_O is less than equal to the maximum mantissa fraction width of posit format (N-ES-3) (Line 12-14). Further, for true output sign-bit (LS) a two's complemented is taken as per posit format requirement (Line 15), which is then appended by sign-bit at the MSB for producing N-bit posit output (Line 17). The final posit output is produced after exceptional check for zero and infinity. For zero case all bit will be 0, and for infinity all but sign bit will be, otherwise, computed posit value will provided as final output.

All the above processing are parameterized for N and ES. The above processing discussion related to the posit data extraction, posit data construction, rounding and final processing acts in similar way for posit multiplier and division, except for core arithmetic computation. The discussion related to posit multiplier and division arithmetic generation is presented in the respective unit discussed below.

B. MULTIPLIER HDL GENERATOR

Similar to the posit adder arithmetic, the posit multiplier arithmetic also consists of three main processing section: posit extraction, core arithmetic and posit construction. The algorithmic computational flow for posit multiplier generator is shown in algorithms-9. The posit data extraction is done exactly as in Algorithm-1. It supplies the operands complemented form (XIN1, XIN2, for negative posit), sign bits (S1, S2), regime check bits (RC1, RC2), absolute regime sequence values (R1, R2), exponent values (E1, E2), mantissas (M1, M2), and infinity & zero check bits (Inf1, Inf2, Z1, Z2).

Algorithm 9 Proposed Posit Multiplier 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: → Algorithm-1
 7:    IN1 → XIN1, S1, RC1, R1, E1, M1, Inf1, Z1
 8:    IN2 → XIN2, S2, RC2, R2, E2, M2, Inf2, Z2
 9:    $Z \leftarrow Z1\&Z2$        $Inf \leftarrow Inf1|Inf2$
10: Posit Core Multiplier Arithmetic Processing:
11: Sign Processing:
12:    S ← S1 xor S2
13: Mantissa Multiplication Processing:
14:    M ← M1×M2                  (Mantissa Multiplication
15:    Movf ← M[MSB]              (Check Mantissa Overflow)
16:    M ← Movf ? M : M << 1      (1-bit Mantissa Shifting for
       overflow)
17: Final EXPONENT (E_O) and REGIME (R_O) Processing:
18:    RG1 ← RC1 ? R1 : -R1       (Effective regime-1 value)
19:    RG2 ← RC2 ? R2 : -R2       (Effective regime-2 value)
20:    Exp_O[ES + RS + 1:0] ← {RG1, E1} + {RG2, E2} + Movf
       (Total Exponent value)
21:    Exp_ON[ES + RS:0] = Exp_O[ES + RS + 1] ? -
       Exp_O : Exp_O              (Absolute Total Exponent
       Value)
22:    E_O[ES-1:0] = (Exp_O[ES + RS + 1] & (|Exp_ON[ES-
       1:0])) ? Exp_O[ES-1:0] : Exp_ON[ES-1:0]     (Exponent
       Output)
23:    R_O[RS:0] = !(Exp_O[ES + RS + 1]) | (Exp_O[ES +
       RS + 1] & (|Exp_ON[ES-1:0])) ? Exp_ON[ES + RS:ES] +
       1'b1 : Exp_ON[ES + RS:ES]   (Absolute Regime Value)
24: Posit Construction, Rounding and Final Processing:
25:    Algorithm-8 using S, R_O, Exp_O, E_O, M
26: Posit Data Composition:
27: REGIME, EXPONENT and MANTISSA Packing:
28: REM ←
       $\overbrace{\text{N\{!Exp\_O[MSB]\}, Exp\_O[MSB]}}^{N+1\ Bits\ Regime\ Sequence},$
    $\overbrace{\text{E\_O, M[(N-ES-2) bits MSBs]}}^{Exponent\ -\ Mantissa},$
     $\overbrace{\text{M[Next 2 bits],|(M[:0])}}^{GRS-Bits}$
29: REM ← REM >> R_O
30: Rounding: Round to nearest even
31:    IF (R_O < N-ES-2)
32:       ULP_add = G.(R + S) + L.G.(!(R + S))
33:       REM ← REM + {(N-1)'b0,ULP_add}
34: REM ← (LS == 0) ? REM : 2's Complement of REM
35: Final Output:
36:    Combine LS with MSB (N − 1) bit of rounded REM
37:    Discharge Output while considering Exceptions
```

The extracted components are processed for core multiplication arithmetic. The sign-bit computation requires a 1-bit xor operation among input sign bits (Line 12). For the mantissa multiplication it requires a (N-ES-2)X(N-ES-2) integer multiplier (Line 14), which output MSB bit is checked for any mantissa multiplication overflow (Line 15). For mantissa multiplication overflow, the mantissa is left shifted by 1-bit for proper normalization (Line 16), the final exponent computation is incremented by 1 (Line 20). For the exponent computation, initially the actual regime values (RG1 and RG2) are computed using respective regime check bits and absolute regime value (Lines 18-19). These regime values are combined with respective exponent (E1, E2) to provide respective effective exponent values of each operand, which are then added along with the Movf bit (mantissa overflow) to provide total output exponent value, Exp_O (Line 20). Based on Algorithm-6, using the total output exponent value Exp_O, the ES-bit exponent output E_O and absolute regime output value R_O is computed in Lines 21-23.

After core arithmetic processing, the posit construction, rounding and final processing is processed similar to the Algorithm-8 and Fig. 6.

C. DIVISION HDL GENERATOR

This section discusses the posit division arithmetic HDL generator. Here, except the core division arithmetic processing, all other processings are similar to the posit multiplier arithmetic. Therefore, only discussion related to the core division processing is presented here. The computational flow for posit division arithmetic is presented in Algorithm-10.

After the posit data extraction, output sign-bit is computed as the xor operation among operands sign bits. The exponent processing (lines 17-23) consists of first combining the regimes with respective exponents, then perform subtraction of divisor total exponent value from dividend value. The outcome is then adjusted for mantissa division underflow and for whether if divisor fraction is zero. After, it is processed similar to the posit multiplier for computation of exponent output (E_O) and absolute regime output value (R_O).

The mantissa division generator is designed using Newton-Raphson (NR) method [14]–[16]. Newton-Raphson method uses the Newton's method to find the inverse of the denominator mantissa and which is then multiply by the numerator mantissa to compute the mantissa quotient. Let N is numerator and D is denominator, then it proceeds as follows:

1) Get an approximated inverse of D, as X_0
2) Compute the successive more accurate approximations $X_1, X_2, X_3, \ldots X_N$, as
   - $X_{i+1} = X_i \times (2 - X_i * D)$
3) Compute quotient as $Q = N \times X_N$

Here, in step-2 the number of correct bits in inverse approximation doubles after each iteration of Newton-Raphson method. Thus, based on the correct size of initial approximation number of iteration is determined for a give output precision.

Based on above NR method, the mantissa division generator is designed and its computational flow is presented in Algorithm-11. It is parameterized for the number of NR iteration. The initial approximation address and word size is taken as ceil(mantissa width / 2^{NR_Iter}). However, for very small mantissa width ($<= 8$), only look-table approximation is used and no NR iteration is taken. With this settings it requires two NR iterations for up-to 32-bit posits and one NR iteration for up-to 16-bit posits, in order to achieve the required accuracy.

In the presented generator, a look-up table of size $2^8 \times 9$ is used, which is suitable for up to 32-bit posit. For higher precision requirement, i.e. for more than 32-bit posit a larger look-up table would be needed based on the effective mantissa size and NR iterations. However, with more NR iterations the effective look-up table size would be smaller. As shown in Algorithm-11 lines 17-26, the required size of look-up table is generated for a given parameter sets. Using the initial approximation, a `for` loop generator is used for successive NR iterations generation (lines 34-37). In each NR iteration it uses two (i*IW + 1)x(MW + 1) integer multipliers and one subtractor, where i is i_{th} NR iteration. After completing NR iteration final approximation is then multiplied by the numerator mantissa (line 38) with a (MW + 1)×(MW + 1) integer multiplier to produce the division quotient.

After the core division computation, the posit construction, rounding, and final processing is carried out similar to the posit multiplier and presented in lines (24-37) of Algorithm-10.

IV. PIPELINED POSIT ARITHMETIC ARCHITECTURES

This section discusses the pipelined architectures for 32-bit posit with ES = 6. It is constructed for each of the adder/subtractor, multiplier and division arithmetic unit. Here, ES = 6 is selected as it requires a maximum mantissa width of 23-bit (N-ES-3), which is same as that for single precision floating point format. However, the similar architecture can be used for different ES values without any larger modifications.

A. ADDER/SUBTRACTOR ARCHITECTURE WITH N = 32 AND ES = 6

The architecture of posit adder with N = 32 and ES = 6 is presented in Fig. 7. It is constructed with 5-stage pipeline registers.

The first stage deals with the posit data extraction, which is carried out as shown in Algorithm-1 and Fig. 4. It requires some logic gate operations for zero, infinity checks. It also needs a 32-bit 2's complement converter (invert and add-1) if posit is negative, a 32-bit LOD for detecting regime termination bit, a 32-bit dynamic left shifter for regime sequence flushing out. Along with posit data extraction, this stage also find out the large operand using a 31-bit greater-than-equal-to unit.

The second, third and part of fourth stages perform core arithmetic processing as discussed in Algorithm-4. The second stage first computes the large and small operand components (sign LS,SS; regime-check bit LRC, SRC; regime value LR, SR; exponent value LE,SE; and mantissa LM,SM) using 2:1 MUXs. Then it computes the total effective exponent value difference by first combining the regime values to the respective exponent value and then perform a ES + RS bits subtraction. The small mantissa is right shifted by this exponent difference value using a 23-bit dynamic right shifter which consists of 5 levels of 23-bit barrel shifters.

Algorithm 10 Proposed Posit Division 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: → Algorithm-1
7:    IN1 → XIN1, S1, RC1, R1, E1, M1, Inf1, Z1
8:    IN2 → XIN2, S2, RC2, R2, E2, M2, Inf2, Z2
9:    $Z \leftarrow Z1\&Z2$          $Inf \leftarrow Inf1|Inf2$
10: Posit Core Division Arithmetic Processing:
11: Sign Processing:
12:    S ← S1 xor S2
13: Mantissa Division Processing using Newton-Raphson Method:
14:    M ← Algorithm-11 (M1, M2)
15:    Mudf ← M[MSB]          (Check Mantissa Underflow)
16:    M ← Mudf ? M : M << 1          (1-bit Mantissa Shifting for overflow)
17: Final EXPONENT (E_O) and REGIME (R_O) Processing:
18:    RG1 ← RC1 ? R1 : −R1          (Effective regime-1 value)
19:    RG2 ← RC2 ? R2 : −R2          (Effective regime-2 value)
20:    Exp_O[ES + RS + 1:0] ← {RG1, E1} - {RG2, E2} − 1 + Mudf + |(M2)          (Total Exponent value)
21:    Exp_ON[ES + RS:0] = Exp_O[ES + RS + 1] ? −Exp_O : Exp_O          (Absolute Total Exponent Value)
22:    E_O[ES-1:0] = (Exp_O[ES + RS + 1] & (|Exp_ON[ES-1:0])) ? Exp_O[ES-1:0] : Exp_ON[ES-1:0]          (Exponent Output)
23:    R_O[RS:0] = !(Exp_O[ES + RS + 1]) | (Exp_O[ES + RS + 1] & (|Exp_ON[ES-1:0])) ? Exp_ON[ES + RS:ES] + 1'b1 : Exp_ON[ES + RS:ES]          (Absolute Regime Value)
24: Posit Construction, Rounding and Final Processing:
25:    Algorithm-8 using S, R_O, Exp_O, E_O, M
26: Posit Data Composition:
27: REGIME, EXPONENT and MANTISSA Packing:
28: REM ← $\overbrace{\text{N\{!Exp\_O[MSB]\}, Exp\_O[MSB],}}^{N+1\ Bits\ Regime\ Sequence}$
        $\overbrace{\text{E\_O, M[(N-ES-2) bits MSBs],}}^{Exponent\ -\ Mantissa}$
        $\overbrace{\text{M[Next 2 bits],|(M[:0])}}^{GRS-Bits}$
29: REM ← REM >> R_O
30: Rounding: Round to nearest even
31:    IF (R_O < N-ES-2)
32:       ULP_add = G.(R + S) + L.G.(!(R + S))
33:       REM ← REM + {(N − 1)'b0,ULP_add}
34: REM ← (LS == 0) ? REM : 2's Complement of REM
35: Final Output:
36:    Combine LS with MSB (N − 1) bit of rounded REM
37:    Discharge Output while considering Exceptions
```

The third stage perform the mantissa addition/subtraction operation based on the effective required operation (S1Ŝ2) using a 24-bit add-sub unit. The result is checked for any mantissa overflow. Further, a LOD operation is carried out on the result to find the leading 1 in case of mantissa underflow following an effective subtraction operation earlier, and consequently in fourth stage the mantissa add/sub result is accordingly left shifted to normalize it. The fourth stage also computes the final sign bit and update the large exponent with mantissa overflow and left shift normalization amount to produce the total effective exponent value.

Algorithm 11 Mantissa Division Generation Flow Using Newton-Raphson Method

```
GIVEN:
  N: Posit Word Size
  ES: Posit Exponent Field Size
  RS: $log_2\ (N)$ (Posit Regime Value Store Space Bit Size)
  NR_Iter: Newton-Raphson Iteration Count
  NRB: 2**(NR_Iter)
  MW = N-ES                    (Max mantissa width + 2)
  IW_MAX = 8    (Max data width for initial approximation)
  IW = ceil(MW/NRB)    (< = IW_MAX, 1/4th bit-width of Mantissa)
  AW_MAX = 8           (Max Address bit-width for initial approximation storage
  AW = ceil(MW/NRB)   (< = AW_MAX, address width for initial approx)

  M2_INV_LUT (Addr[AW_MAX-1:0], Dout[IW_MAX-1:0])      (Initial Approx Storage)


Input Operands: M1, M2

Mantissa Division Processing using NR Method: M ← M1/M2
  Get Initial Approximation of M2 Inverse
  IF (MW > = AW_MAX)   (If Mantissa Width is > = Max Address width)
    IF (AW == AW_MAX)
      Dout ← M2_INV_LUT (M2[MW-1:MW-AW_MAX])
    ELSE
      Dout ← M2_INV_LUT ({M2[MW-1:MW-AW],{AW_MAX-AW{1'b0}}})
  ELSE
    Dout ← M2_INV_LUT ({M2[MW-1:0],{AW_MAX-MW{1'b0}}})
  M2_INV0[IW-1:0] ← Dout[IW_MAX-1:IW_MAX-1-(IW-1)]

  Newton-Raphson Iteration to Improve M2 Inverse
  GENERATE
    M2_INV [NR_Iter:0][2*MW+1:0]   (NR_Iter number of register for M2_INV, with size 2*MW+2)
    M2_INV[0] = M2_INV0   (Assign Initial Approximation to first register)
    IF (NR_Iter > 0)
      M2_INVX_X_M2 [NR_Iter-1:0][2*MW+1:0]
      two-M2_INV_X_M2 [NR_Iter-1:0][MW:0]
      for (i = 0; i < NR_Iter; i = i + 1) (begin NR_Iteration)
        M2_INV_X_M2[i] = M2_INV[i][i*IW MSBs] * M2

        two-M2_INV_X_M2[i] = 2 − M2_INV_X_M2[i]
        M2_INV[i + 1] = M2_INV[i][i*IW MSBs] *
                                    two-M2_INV_X_M2[i]
    M = |(M2[MW-1:0]) ? M1 :
                        M1 * M2_INV[NR_Iter][2*MW:MW]
```

The core computation processing completed at this point and posit construction processing begins (Algorithm-refposit-construct). The output regime value (R_O) and exponent value (E_O) are evaluated using Algorithm-6&7, which requires a 2's complement converter and an adder. The regime sequence, exponent value and mantissa along with GRS (Guard, Round and Sticky bits) 2*N + 3 bits packing is also performed in 4th stage (as shown in Fig. 6).

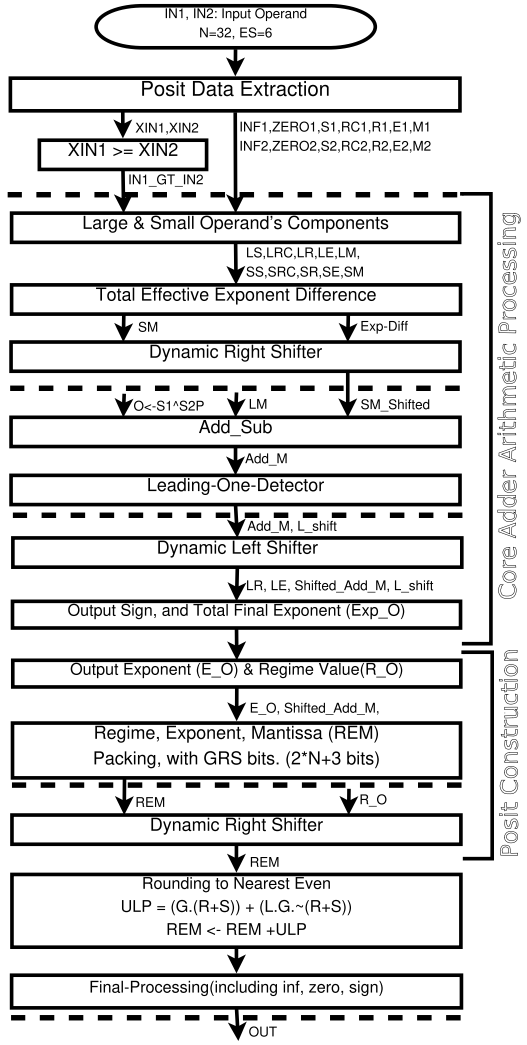
IN1, IN2: Input Operand
N=32, ES=6
Posit Data Extraction
XIN1,XIN2
INF1,ZERO1,S1,RC1,R1,E1,M1
INF2,ZERO2,S2,RC2,R2,E2,M2
XIN1 >= XIN2
IN1_GT_IN2
Large & Small Operand's Components
LS,LRC,LR,LE,LM,
SS,SRC,SR,SE,SM
Total Effective Exponent Difference
SM
Exp-Diff
Dynamic Right Shifter
O<-S1^S2P
LM
SM_Shifted
Add_Sub
Add_M
Leading-One-Detector
Add_M, L_shift
Dynamic Left Shifter
LR, LE, Shifted_Add_M, L_shift
Output Sign, and Total Final Exponent (Exp_O)
Core Adder Arithmetic Processing
Output Exponent (E_O) & Regime Value(R_O)
E_O, Shifted_Add_M,
Regime, Exponent, Mantissa (REM)
Packing, with GRS bits. (2*N+3 bits)
REM
R_O
Dynamic Right Shifter
REM
Posit Construction
Rounding to Nearest Even
ULP = (G.(R+S)) + (L.G.~(R+S))
REM <- REM +ULP
Final-Processing(including inf, zero, sign)
OUT

FIGURE 7. Posit (32,6) adder, 5 stage pipelined architecture (dotted lines are pipeline registers).

The fifth/last stage of posit adder architecture first dynamically right shift the above constructed REM sequence by output regime value (R_O) in order to shift required regime sequence towards right. Then it perform the rounding operation (round to nearest even), where it computes ULP (unit at last place) using precision bit (L), Guard-bit, Round-Bit and Sticky-bit and then add it to shifted REM at precision bit. The result is then updated according to the sign bit (take 2's complement if true sign bit), infinity and zero exceptional cases and provide the final posit addition output.

B. MULTIPLIER ARCHITECTURE WITH $N = 32$ AND $ES = 6$

The pipelined architecture for 32-bit posit with ES = 6 is presented in Fig. 8. It consists of 6 pipeline stages. Here, only core arithmetic processing are discussed and remaining processing are similar to the corresponding adder units (the posit extraction, posit construction and final processing).

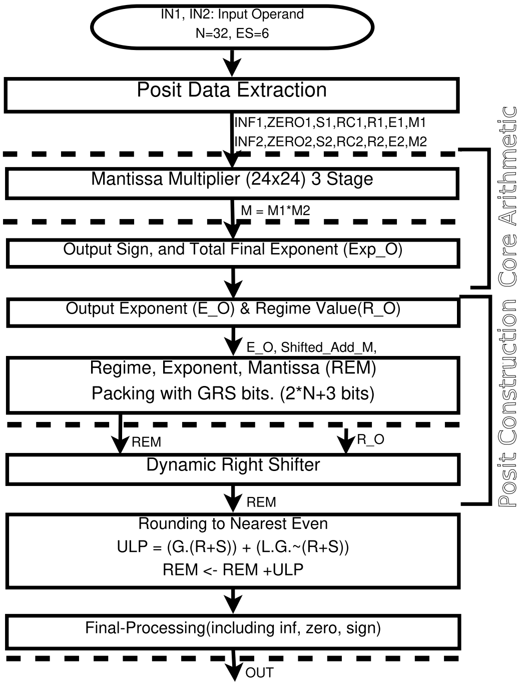
IN1, IN2: Input Operand
N=32, ES=6
Posit Data Extraction
INF1,ZERO1,S1,RC1,R1,E1,M1
INF2,ZERO2,S2,RC2,R2,E2,M2
Mantissa Multiplier (24x24) 3 Stage
M = M1*M2
Output Sign, and Total Final Exponent (Exp_O)
Output Exponent (E_O) & Regime Value(R_O)
E_O, Shifted_Add_M,
Regime, Exponent, Mantissa (REM)
Packing with GRS bits. (2*N+3 bits)
REM
R_O
Dynamic Right Shifter
REM
Rounding to Nearest Even
ULP = (G.(R+S)) + (L.G.~(R+S))
REM <- REM +ULP
Final-Processing(including inf, zero, sign)
OUT
Posit Construction Core Arithmetic

FIGURE 8. Posit (32,6) multiplier, 6 stage pipelined architecture (dotted lines are pipeline registers).

The largest unit in the posit multiplier architecture is the mantissa multiplication. For N = 32 and ES = 6, it requires a 24 × 24 unsigned integer multiplier. The architecture for 24 × 24 multiplier is shown in Fig. 9. It consists of 3 pipeline stages and constructed using a DSP48E embedded unit in Xilinx FPGA device. Primarily, a DSP48E unit can process a 25 × 18 signed multiplication and a 48 bit accumulation, along with several other operations. Here, one DSP48E unit is composed with a 24 × 7 multiplier to form the 24 × 24 multiplier. The final pipeline register in this unit is merged with the next pipeline register of main architecture (Fig. 8).

Further, the mantissa multiplication result is checked for the mantissa overflow and processed accordingly. The sign bit is computed as an XOR operation between both sign bits. The total output exponent value is computed by first combining the effective regime values with the respective exponent values and then adding them together with the mantissa overflow bit. After this E_O and R_O are computed which is followed by the posit construction and final processing stages.

C. DIVISION ARCHITECTURE WITH N = 32 AND ES = 6

The architecture for 32-bit posit with ES = 6 is shown in Fig. 10. It is constructed with 12 pipeline stages. Only core arithmetic processing are discussed here, as other related computations are similar to the respective processing in above arithmetic units.

The mantissa division architecture is shown in Fig 11. It is based on 2 iterations of Newton-Raphson division method. It consists of 9 pipeline stages: 6 for 2 NR iterations and 3 for final multiplication between dividend mantissa and final divisor inverse approximation. First, a 9-bit inverse approximation of divisor mantissa is obtained from a $2^8 \times 9$ look-up table with 8 MBS bits of divisor mantissa (excluding hidden bit) as address bits.

The first NR iteration consists of two 25 × 9 multiplier and a subtractor. The 25 × 9 multiplier architecture is shown in Fig 12. It consists of one DSP48E unit and a 1 × 8 multiplier. The sum of both multiplier is carried out by adder available on DSP48E unit. It requires 3 stages for first NR iteration. The second NR iteration consists of two 25 × 17 multiplier and one subtractor. It also requires 3 pipeline stages. The architecture for 25 × 17 multiplier is shown in Fig. 13, which is composed by one DSP48E unit and a 1 × 17 multiplier. This produces the improved divisor

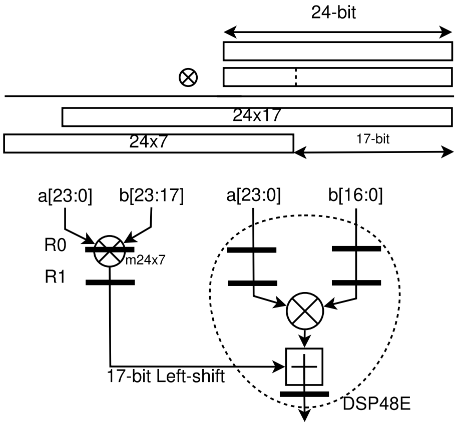

FIGURE 9. 24 × 24 multiplier architecture.

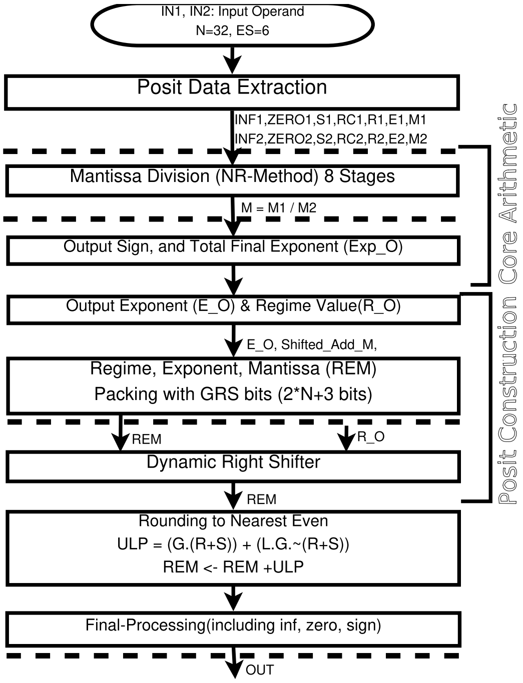
IN1, IN2: Input Operand
N=32, ES=6
Posit Data Extraction
INF1,ZERO1,S1,RC1,R1,E1,M1
INF2,ZERO2,S2,RC2,R2,E2,M2
Mantissa Division (NR-Method) 8 Stages
M = M1 / M2
Output Sign, and Total Final Exponent (Exp_O)
Output Exponent (E_O) & Regime Value(R_O)
E_O, Shifted_Add_M,
Regime, Exponent, Mantissa (REM)
Packing with GRS bits (2*N+3 bits)
REM
R_O
Dynamic Right Shifter
REM
Rounding to Nearest Even
ULP = (G.(R+S)) + (L.G.~(R+S))
REM <- REM +ULP
Final-Processing(including inf, zero, sign)
OUT
Posit Construction Core Arithmetic

FIGURE 10. Posit (32,6) division, 12 stage pipelined architecture (dotted lines are pipeline registers).

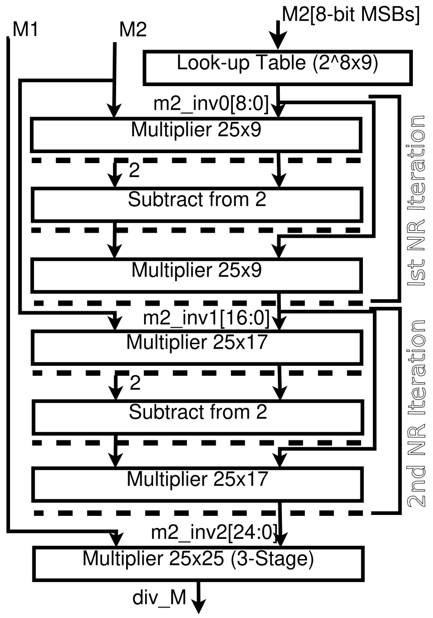

FIGURE 11. 9 stage mantissa division architecture (dotted lines are pipeline registers).

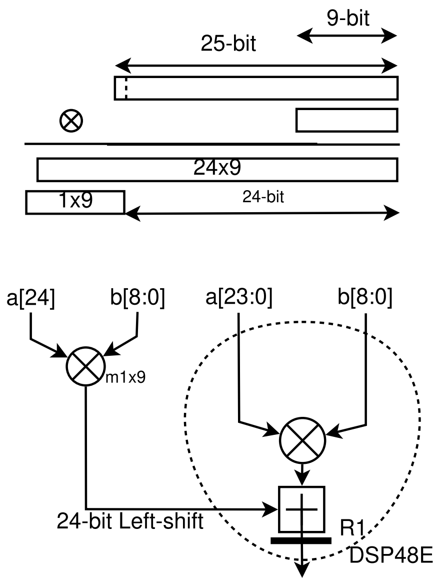

FIGURE 12. 25 × 9 multiplier architecture.

mantissa inverse approximation which suitable for 24-bit precision.

Finally, the dividend mantissa is multiplied by inverse approximation using a 25 × 25 multiplier. The architecture for 25 × 25 multiplier is shown in Fig. 14. It consists of a DSP48E, one 25 × 8 and one 1 × 17 multiplier. The strategy for this multiplication break-up is also shown in Fig. 14. It is a 3 stage multiplier. Thus, mantissa division using 2 NR iterations requires 5 DSP48 multipliers along with several small multipliers, and a look-up table for initial inverse approximation.

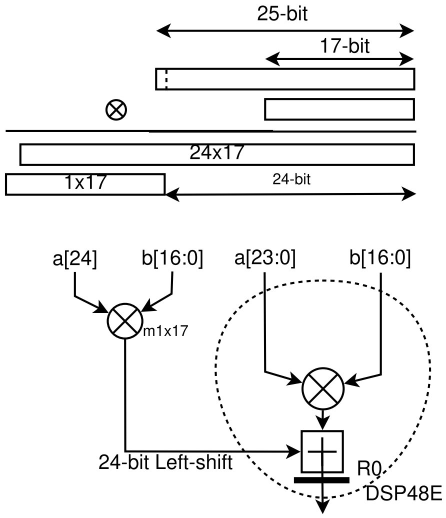

FIGURE 13. 25 × 17 multiplier architecture.

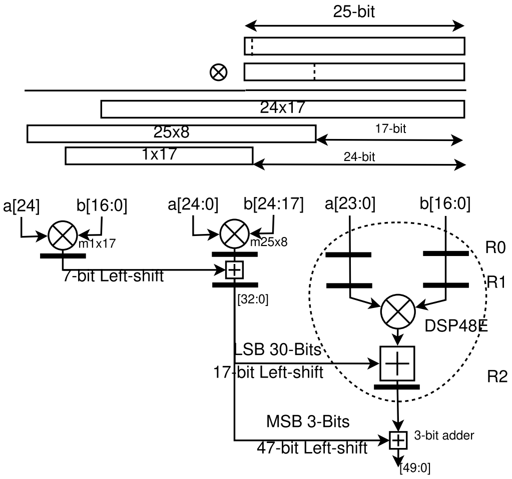

FIGURE 14. 25 × 25 multiplier architecture.

V. IMPLEMENTATION RESULTS

All the generated posit arithmetic units are functionally verified against the Julia package for posit [18] provided by the posit developers using several millions of random test cases. Moreover, a complete validation is done for 8-bit posit for all the generated arithmetic.

The proposed HDL generators for posit arithmetic units are demonstrated on FPGA as well as ASIC platform. These posit arithmetic units are generated for various combinations of word size (N) and exponent size (ES) and implemented on Xilinx Virtex-7 (xc7vx330t-3ffg1157) FPGA device and Nangate 15nm ASIC platform. A single cycle implementation details of proposed posit arithmetic units are presented in Figs. 15 & 16, respectively for Virtex-7 FPGA device and Nangate 15nm ASIC, for different posit configurations (N,ES). The timing metrics are obtained after placing registers at the primary inputs and outputs. Also, in order to get a clear view on the variations in resource utilization, only logics

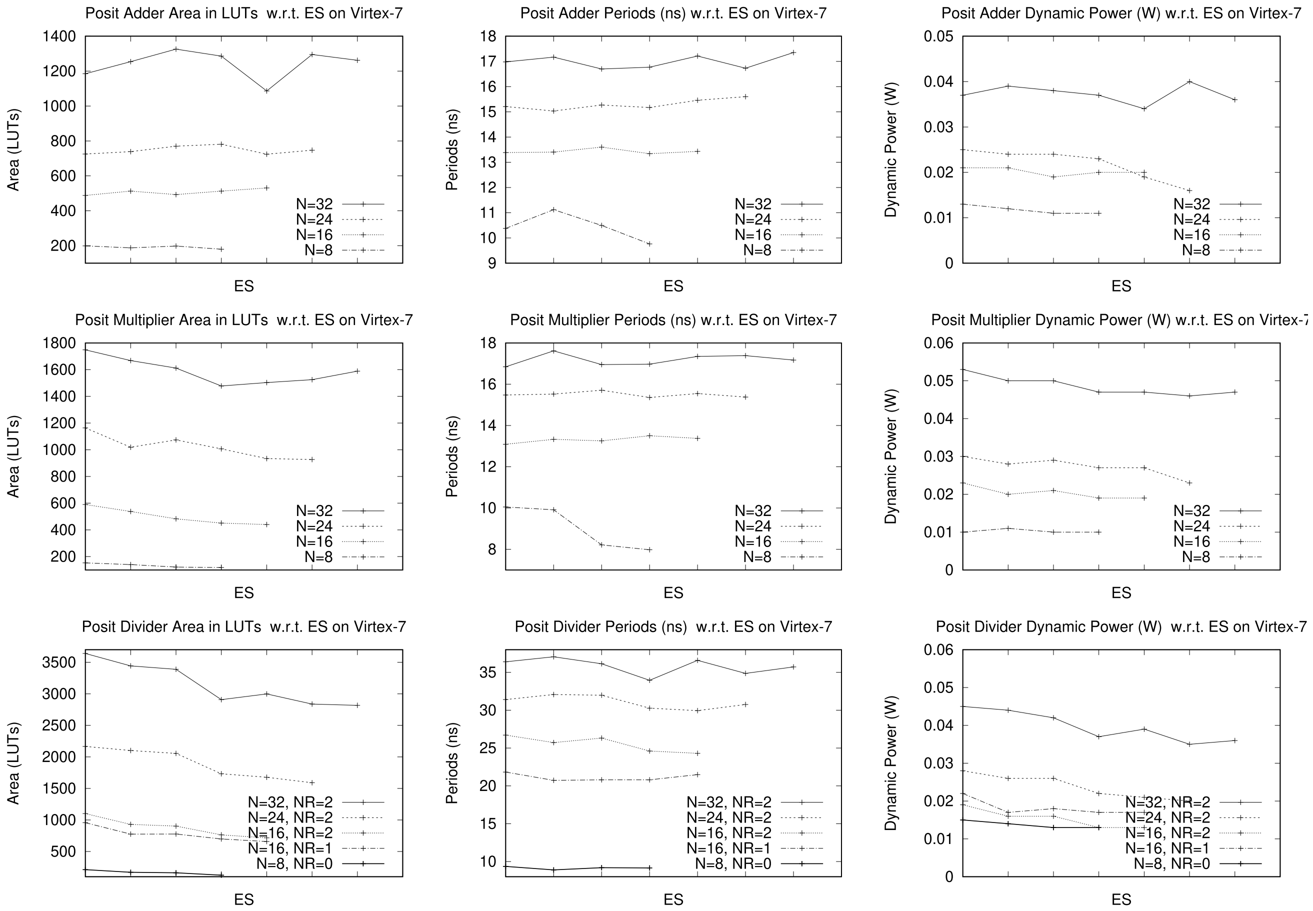

FIGURE 15. Implementation details for posit adder, multiplier and divider on Virtex-7 FPGA.

are used for integer multiplications and look-up-table in posit multiplier and posit divider implementation on FPGA device, instead of DSP48 and BRAM IPs. It can be observed that for the adder arithmetic, for a given word size the resource utilization does not vary much for different ES values. It is because all the major sub-components of adder unit are mostly dependent on word-size dimension as compared to ES. Whereas, in the case of multiplier and divider units, the resource utilization decreases with the increasing ES value. This is because in these units the inherent integer multiplications are resource dominating units, and by increasing ES value the effective mantissa width decreases which leads to decrease in the multipliers resource requirements. Further, in the posit divider unit two NR iterations are used on 32-bit posit, while no NR iteration is used for 8-bit posit (only enough look-up-table approximation is generated for posit mantissa less than 9-bit). Moreover, a parametric variation in NR iterations can be seen for 16-bit posit, which is implemented with one and two NR iterations.

The proposed pipelined architectures are also implemented on Xilinx Virtex-7 FPGA device for (32,6) posit adder, multiplier and divider arithmetic units and respective design metrics are presented in Table-1. These pipelined architectures are designed using FPGA specific key resources (DSP48E, BRAM) in order to optimize them for the design platform, as discussed in their respective description. Here, value of ES is taken as 6 as it would leads to a maximum mantissa width of 23-bit (N-ES-3), similar to single precision (SP) mantissa bit-width and it can used to visualize the design metrics of both the posit and SP floating point. None-the-less, same approach can be used for different ES values.

TABLE 1. Implementation details for (32,6) posit arithmetic pipelined architectures on Virtex-7 FPGA.

	LUTs	DSP48E	BRAM	Period (ns)	Latency (cycle)
Adder	946	0	0	4.1	5
Multiplier	854	1	0	4.4	6
Divider	962	6	1	4.6	12

A comparison of proposed posit adder and multiplier generator is shown in Table-2. It is compared against Chaurasiya et al. [12]. Since [12] has reported the synthesis results using DSP48E for mantissa multiplication in posit multiplier, and also carried out synthesis without any primary input/output registers. So, we have also carried out a similar synthesis for the proposed posit adder and multiplier

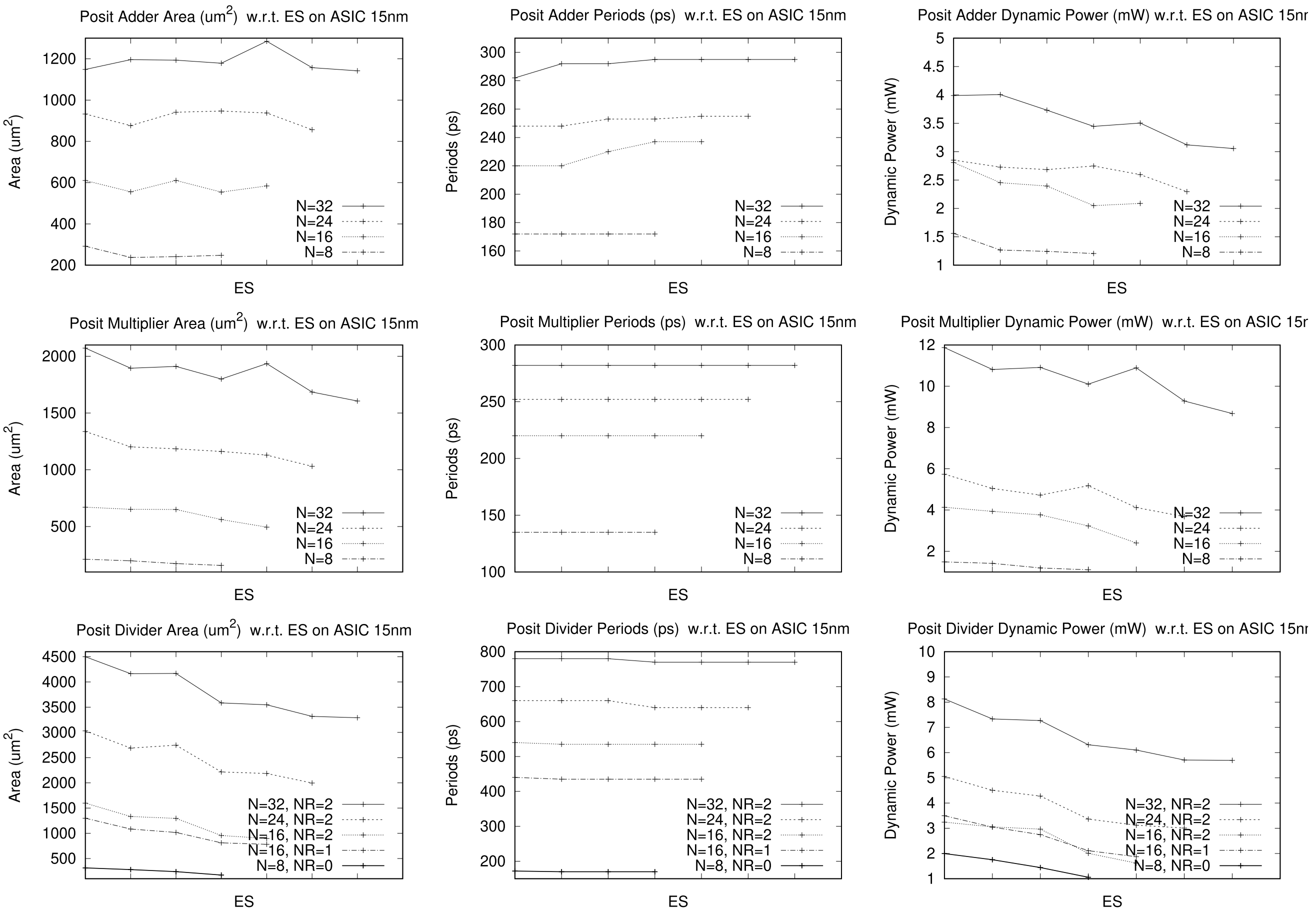

FIGURE 16. Implementation details for posit adder, multiplier and divider on Nangate 15nm ASIC.

TABLE 2. Comparison of "Area (LUT) X Period (ns)" product for posit adder and multiplier on Virtex-7 FPGA.

Posit Format	Posit Adder		Posit Multiplier	
	Proposed	[12]	Proposed	[12]
(16,1)	445x16.146 7184.9	391x32.374 12658	273x14.574 3978	218x24.041 5240.9
(16,2)	492x16.410 8073.7	404x33.974 13725.5	273x14.332 3912.6	223x23.680 5280.6
(16,3)	450x16.989 7645	386x32.466 12531.9	280x14.615 4092.2	219x24.078 5273.1
(32,1)	1211x19.150 23190.65	934x38.041 35530.3	668x19.076 12742.77	576x31.013 17863.5
(32,2)	1275x19.984 25480	981x40.032 39271.4	680x18.921 12866.28	572x33.021 18888.1
(32,3)	1272x19.752 25125	951x39.254 37330.554	720x19.203 13826.2	582x32.263 18777.1

generator to get the resource utilization under similar synthesis environment. For both cases, the DSP48E uses are same in all posit formats. A comparison of "area (LUT) × period (ns)" product is shown is Table-2 for overall comparison. It can be seen that the proposed posit HDL generator outperform the [12] in various posit configurations of adder and multiplier arithmetic.

A visualization of posit arithmetic implementation details as against the single precision floating point arithmetic is discussed here. Another recent pipelined implementation of a fixed size (32,3) posit adder and multiplier architectures is also included here, together with present proposed.

Table-3 includes the pipelined architectures of 32-bit posit adders and single precision floating point adders. Various methods of single precision floating point adder is included here. The FP architecture from Flopoco [19] and Xilinx [20] supports only simplified normal single precision implementation and thus use smaller resources compared of a fully IEEE-754 compliant implementation. It can be visualize that the design metrics for posit architectures found an average place among various methods of floating point adder implementation.

Table4 presents the design metrics of pipelined posit multipliers and some recent single precision floating point multipliers. Similar to the adder unit, here also we can see that posit multipliers do not seek beyond the FP multipliers. Also, the presented floating-point multipliers only support normal implementation, and it would need more resources and periods for fully IEEE-754 supported implementation.

Similarly, Table-5 includes design metric of proposed pipelined posit division and floating point division from Flopoco and Xilinx. The Flopoco and Xilinx uses digit

TABLE 3. 32-bit posit vs 32-bit FP adder architectures.

	LUTs	Period (ns)	Latency (cycle)
Posit Adders			
(32,6) Proposed (V7)	946	4.1	5
(32,3) [11] (V7)	1053	4.92	12
FP SP Adder Methods			
Flopoco Single-Path [19] (V7)(Simplified Normal)	421*	3.997	4
Xilinx FPU v7.1 [11], [20] (V7) (Normal)	508	3.09	8
Single-Path [12] (V7) (Normal)	1049	41.567	1
Standard Single-Path [21] (V2P) (Denormal)	541 (Slices)	27.06	1
LOP [21] (V2P) (Denormal)	748 (Slices)	25.33	1
Two-Path [21] (V2P) (Denormal)	1018 (Slices)	21.82	1

*Flopoco adder also uses BRAMs (of size 16x2, 64x2) along with above LUTs, V7: Virtex-7, V2P: Virtex-II pro

TABLE 4. 32-bit posit vs 32-bit FP multiplier architectures.

	LUTs	DSP48E	Period (ns)	Latency (cycle)
Posit Multipliers				
(32,6) Proposed (V7)	854	1	4.4	6
(32,3) [11] (V7)	1303		4.97	11
FP SP Multipliers				
Xilinx FPU v7.1 (V7) [11]	630		3.13	8
Xilinx FPU v7.1 (V7) [20]	238	1	2.165	8
Xilinx FPU v7.1 (V7) [20] (Simplified Normal)	92	2	2.165	8
[12] (V7)(Normal)	533	4	29.051	1
[22] (V5) (Normal)	392	1	3.02	5

recurrence method for division and support only normal floating point processing. The proposed posit division and [23] is based on Newton-Raphson division method, which is a multiplicative division method. However, [23] has presented the SP inverse metrics, a full division, which would need an extra 24 × 24 integer multiplier. In general the multiplicative division methods are used for better performance, low latency but at some hardware cost. However, the digit recurrence method requires smaller area but with large latency's. With similar division method (NR) posit appears to find similar design metrics, as the core mantissa arithmetic processing is exactly similar in both cases and it is dominating architectural component.

In general, the core arithmetic computational flow is mostly similar for both the posit and floating point. The main difference appears in the input extraction and output construction related architecture, which is little costly in posit due to run-time varying fields. However, in floating point the exceptional and sub-normal handling also require a complex architectural handling and mostly affect the timing metrics. On the other hand, at qualitative point of view, the posits provides more dynamic range for similar word-size along with more exact and accurate results [6], [9] along with several other benefits.

TABLE 5. 32-bit posit vs 32-bit FP division architectures.

	LUTs	DSP48E	Period (ns)	Latency (cycle)
Posit Division				
(32,6) Proposed (V7)	962	6	4.6	12
FP SP Division Methods				
SP Inverse (V4) [23] (Newton-Raphson) (Normal)	2448	-	14.957	15
Flopoco (V7) [19] (SRT Digit Recurrence) (Simplified Normal)	1110		3.044	18
Xilinx FPU v7.1 [20] (Digit Recurrence) (Normal)	173		2.66	26

TABLE 6. FIR filter implementation details using posit cores.

Posit Size	LUTs	DSP48E	Period (ns)
Posit (32,6)	5889	16	40
Posit (32,6)	8585	0	40
Posit (16,3)	2719	4	25
Posit (16,3)	3434	0	31
Posit (8,1)	1026	0	18

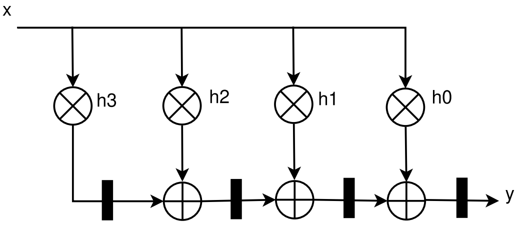

FIGURE 17. 4-Tap FIR filter architecture.

All the proposed posit arithmetic HDL generator units are parameterized for N and ES which can generate hardware for any desired value. The source code of all the proposed posit arithmetic HDL generator and pipelined architectures modules will be provided as an open-source material which will be available at [24]. To the best of authors' knowledge the proposed posit HDL generator for division arithmetic remains the only available proposal. These open-source proposal on posit arithmetic HDL generator would provide a basic platform for more advance and innovative research on posit arithmetic and its application on hardware platform.

A. FIR FILTER IMPLEMENTATION USING POSIT ARITHMETIC

Here we are presenting the implementation details of a 4-tap FIR filter using posit arithmetic cores as an example case. The architecture of FIR filter is shown in Fig. 17 and is based on the function-4. The implementation is carried out with full-use as well as with no-use of DSP48 IP. The implementation details for FIR filter using 3 different posit word size is presented in Table-6. Here, the single cycle posit units are used, however, pipelined posit units can be used for performance improvement.

$$y[n] = x[n].h0 + x[n-1].h1 + x[n-2].h1 + x[n-3].h3 \tag{4}$$

VI. CONCLUSIONS

Posit is an recent development in numerical computing and has shown some significant benefit over IEEE-754 floating point standard. This paper proposed an open-source parameterized posit arithmetic core generator (PACoGen) for posit adder, posit multiplier and divider arithmetic. It can generated respective posit arithmetic hardware for any word size (N) and exponent size (ES) combination and comply fully with posit definitions. It also presented a pipelined version of posit arithmetic architecture for a given posit format. It addresses the algorithmic development flow of the posit arithmetic units. This work would enable the community for more exploration of posit arithmetic and its application. This would provide an initial platform for posit arithmetic hardware. The proposed posit HDL generators are demonstrated on Virtex-7 FPGA device as well as on Nangate 15nm ASIC platform form various combination of posit format. The proposal is compared with available literature on posit adder and multiplier and shows better design metrics, apart from being open-access for further development. Moreover, the posit division HDL generator proposal remains the only available work. The source code of proposed posit HDL generators and pipelined architectures would be available at [24]

REFERENCES

[1] J. L. Gustafson, *The End of Error: Unum Computing*, 1st ed. Boca Raton, FL, USA: CRC Press, 2015.

[2] W. Tichy, "The end of (numeric) error: An interview with John L. Gustafson," *Ubiquity*, vol. 2016, p. 1, Apr. 2016. doi: 10.1145/2913029.

[3] 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/

[4] J. Gustafson, "A radical approach to computation with real numbers," *Supercomput. Frontiers Innov.*, vol. 3, no. 2, pp. 38–53, Jul. 2016. [Online]. Available: http://superfri.org/superfri/article/view/94

[5] W. Tichy, "Unums 2.0: An interview with John L. Gustafson," *Ubiquity*, vol. 2016, p. 1, Oct. 2016. doi: 10.1145/3001758.

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

[7] John L. Gustafson. (Feb. 1, 2017). *Beyond Floating Point: Next-Generation Computer Arithmetic Stanford EE Computer Systems Colloquium*. [Online]. Available: http://web.stanford.edu/class/ee380/Abstracts/170201.html and https://www.youtube.com/watch?v=aP0Y1uAA-2Y&feature=youtu.be

[8] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, Aug. 2008, pp. 1–70.

[9] J. L. Gustafson. (2017). *Posit Arithmetic*. [Online]. Available: https://posithub.org/docs/Posits4.pdf

[10] M. K. Jaiswal and H. K.-H. So, "Universal number posit arithmetic generator on FPGA," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1159–1162.

[11] J. Hou, Y. Zhu, S. Du, and S. Shong, "Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on FPGA," *J. Signal Process. Syst.*, pp. 1–12, Nov. 2018. doi: 10.1007/s11265-018-1420-5.

[12] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, and R. Leupers, "Parameterized posit arithmetic hardware generator," in *Proc. IEEE 36th Int. Conf. Comput. Design (ICCD)*, Oct. 2018, pp. 334–341.

[13] M. K. Jaiswal and H. K.-H. So, "Architecture generator for type-3 unum posit adder/subtractor," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[14] S. F. Oberman and J. M. Flynn, "An analysis of division algorithms and implementations," Standford Univ., Stanford, CA, USA, Tech. Rep., 1995.

[15] S. F. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7/sup TM/ microprocessor," in *Proc. 14th IEEE Symp. Comput. Arithmetic*, Apr. 1999, pp. 106–115.

[16] P. Montuschi, L. Ciminiera, and A. Giustina, "Division unit with Newton-Raphson approximation and digit-by-digit refinement of the quotient," *IEE Proc. Comput. Digit. Techn.*, vol. 141, no. 6, pp. 317–324, Nov. 1994.

[17] NC State Univ. (2014). *FreePDK15*. [Online]. Available: https://research.ece.ncsu.edu/eda/freepdk/freepdk15/

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

[19] F. D. Dinechin and B. Pasca. *Flopoco Project: Floating-Point Cores (but not Only) for FPGAs (but not Only)*. Accessed: Dec. 27, 2018. [Online]. Available: http://flopoco.gforge.inria.fr/

[20] Xilinx. *LogiCORE IP Floating-Point Operator v7.1*. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html

[21] A. Malik, D. Chen, Y. Choi, M. H. Lee, and S. B. Ko, "Design tradeoff analysis of floating-point adders in FPGAs," *Can. J. Elect. Comput. Eng.*, vol. 33, nos. 3–4, pp. 169–175, 2008.

[22] M. K. Jaiswal and H. K.-H. So, "DSP48E efficient floating point multiplier architectures on FPGA," in *Proc. 30th Int. Conf. VLSI Design, 16th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2017, pp. 1–6.

[23] P. Malík, "High throughput floating-point dividers implemented in FPGA," in *Proc. IEEE 18th Int. Symp. Design Diagnostics Electron. Circuits Syst.*, Apr. 2015, pp. 291–294.

[24] M. K. Jaiswal. (2019). *PACoGen: Posit Arithmetic Core Generator*. [Online]. Available: https://github.com/manish-kj/PACoGen

MANISH KUMAR JAISWAL (M'12) received the B.Sc. and M.Sc. degrees from D.D.U. Gorakhpur University, India, in 2002 and 2004, respectively, the M.S. degree (By Research) from IIT Madras, in 2009, and the Ph.D. degree from the City University of Hong Kong, in 2014. He is currently a Research Scientist with The University of Hong Kong. His research interests include digital VLSI design, reconfigurable computing, ASIC/FPGA SoC design, and reconfigurable machine learning. He received the Outstanding Academic Performance Award for his Ph.D. degree.

HAYDEN K.-H. SO (S'03–M'07–SM'15) received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer sciences from the University of California at Berkeley, Berkeley, CA, USA, in 1998, 2000, and 2007, respectively. He is currently an Associate Professor with the Department of Electrical and Electronic Engineering, The University of Hong Kong. He received the Croucher Innovation Award, in 2013, for his work in power-efficient high-performance heterogeneous computing systems. He received the Faculty Best Teacher Award, in 2011, and the University Outstanding Teaching Award (Team), in 2012.