PERI: A Configurable Posit Enabled RISC-V Core

SUGANDHA TIWARI, Indian Institute of Technology Madras, India

NEEL GALA, InCore Semiconductors Pvt. Ltd., India

CHESTER REBEIRO and V. KAMAKOTI, Indian Institute of Technology Madras, India

Owing to the failure of Dennard's scaling, the past decade has seen a steep growth of prominent new paradigms leveraging opportunities in computer architecture. Two technologies of interest are *Posit* and RISC-V. *Posit* was introduced in mid-2017 as a viable alternative to IEEE-754, and RISC-V provides a commercial-grade open source Instruction Set Architecture (ISA). In this article, we bring these two technologies together and propose a Configurable *Posit* Enabled RISC-V Core called PERI.

The article provides insights on how the Single-Precision Floating Point ("F") extension of RISC-V can be leveraged to support *posit* arithmetic. We also present the implementation details of a parameterized and feature-complete *posit* Floating Point Unit (FPU). The configurability and the parameterization features of this unit generate optimal hardware, which caters to the accuracy and energy/area tradeoffs imposed by the applications, a feature not possible with IEEE-754 implementation. The *posit* FPU has been integrated with the RISC-V compliant SHAKTI C-class core as an execution unit. To further leverage the potential of *posit*, we enhance our *posit* FPU to support two different exponent sizes (with *posit*-size being 32-bits), thereby enabling multiple-precision at runtime. To enable the compilation and execution of C programs on PERI, we have made minimal modifications to the GNU C Compiler (GCC), targeting the "F" extension of the RISC-V. We compare *posit* with IEEE-754 in terms of hardware area, application accuracy, and runtime. We also present an alternate methodology of integrating the *posit* FPU with the RISC-V core as an accelerator using the custom opcode space of RISC-V.

CCS Concepts: • **Hardware** → **Arithmetic and datapath circuits**; • **Computer systems organization** → **Reduced instruction set computing**;

Additional Key Words and Phrases: *Posit*, IEEE-754, RISC-V, floating-point, processor

ACM Reference format:

Sugandha Tiwari, Neel Gala, Chester Rebeiro, and V. Kamakoti. 2021. PERI: A Configurable Posit Enabled RISC-V Core. *ACM Trans. Archit. Code Optim.* 18, 3, Article 25 (April 2021), 26 pages.
<https://doi.org/10.1145/3446210>

New Paper, Not an Extension of a Conference Paper.

The authors thank the Department of Science and Technology, India, to support this work through the DST-FIST program Grant 2016.

Authors' addresses: S. Tiwari, C. Rebeiro, and V. Kamakoti, Indian Institute of Technology Madras, India; emails: {sugandha, chester, kama}@cse.iitm.ac.in; N. Gala, InCore Semiconductors Pvt. Ltd., India; email: neelgala@incoresemi.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

1544-3566/2021/04-ART25

<https://doi.org/10.1145/3446210>

1 INTRODUCTION

For years, computer architects have leveraged increased resources due to Moore's law [10] and Dennard's scaling [8] to improve the performance and functionality of general-purpose processors. However, their failure to scale further, the recent explosion in domain-specific applications, and the need for customized hardware have challenged architects to revisit basic computer architecture principles for hidden opportunities. One such opportunity lies in floating-point computation.

The IEEE-754 standard [39] has been the defacto floating-point standard for several decades. Hardware implementations catering to this standard are ubiquitous today in all major computing platforms. While it continues to drive many modern-day applications, there exist a few barriers in the IEEE-754 standard, which has forced researchers to find alternatives. For example, it is a well-known fact that the corresponding hardware units require a significant amount of area and energy of the chip [12, 42]. Moreover, the complexity and inconsistencies within the IEEE-754 standard has led to several implementation errors of FPUs [11, 38, 48]. This has made compliance with the standard a major effort. IEEE-754 also suffers from lack of guaranteed reproducibility for some operations and across platforms as many of the implementation details are left to the choice of compiler [39]. The specification for handling overflow and underflow further leads to loss of information.

The standard, even today, allows multiple and redundant representations of Not-a-Number (NaNs), making software portability across implementations a major challenge. IEEE-754 allows NaN payload to carry diagnostic information about the error, but the propagation of NaN payloads is not fully standardized. IEEE-754 standard does not clarify what happens when combining two NaNs or converting the precision of NaN. Different implementations handle this in a different way, which creates software portability issues. For example, Intel delivers the NaN with the larger fraction. IBM RISC System/6000 delivers a NaN according to operand number. In IBM System/390, signaling NaNs (sNaN) are chosen over quiet NaNs (qNaN) and NaNs of the same type are selected by operand number [1]. One more such example is that the recent HardFloat design [17] provides specific implementation of NaN propagation for x86, ARM, and RISC-V architectures.

While subnormals are rare-to-occur numbers, the corresponding hardware to handle them adds significant overheads to the entire design. While the standard today elaborates on 16-bit, 32-bit, 64-bit, 128-bit, and 256-bit representations, each bit-width can support only fixed precision and range. Despite these challenges, the lack of a suitable replacement has forced architects to continue with the IEEE-754 standard for floating-point numbers.

Recently, *posits* [15] have been proposed as an alternate and efficient way of representing floating-point numbers. *Posit* enables reproducible results across platforms by providing a representation that removes the ambiguity in floating-point arithmetic behaviour. Unlike IEEE-754, which supports overflow and underflow, *posit* does not support overflow and underflow. *Posit-based arithmetic is significantly simple to design due to the absence of subnormals and NaNs, thereby also simplifying handling of exceptions to a large extent.*

Several works in literature [7, 19, 22–24, 26, 34, 44] have proposed techniques and methodologies to implement hardware for *posit*-based arithmetic operations such as adders, multipliers, and dividers. Though these operations form an integral part of many *posit*-based FPUs, they are not sufficient to support the compute requirements of a commercial-grade ISA. Such support would require functional units that can perform conversion of a *posit* number to integer and vice versa, compare *posit* numbers, square-root operations, classification operations, and much more. To that extent, in this article, we propose designs of relevant functional units required to build a complete *posit*-based FPU, which supports the floating-point compute extension of the RISC-V [47] ISA. Furthermore, the article integrates the proposed *posit* FPU with an open source general-purpose SHAKTI C-class RISC-V core as a tightly coupled execution unit within the core pipeline.

Unlike IEEE-754, *posits* are not restricted by a constant exponent and fraction size. A *posit* representation is determined by the exponent size (*es*) parameter, allowing one to select different *es* values to get different precision and dynamic range for the same posit-size (*ps*). This enables configurable *posit* representation where the *es* value can be decided at design-time to meet the area and energy budget. The *ps* value is also configurable at design time. In this article, we provide the design of *posit* FPU, which is parameterized to support any combination of *es* and *ps* values. In addition, keeping in mind the diverse needs of different applications, the proposed *posit* FPU can be dynamically configured to support multiple *es* values within the same hardware. This enables dynamic switching of precision and range at runtime by the application, depending on the needs of the application. The dynamic switching is controlled by new control/status register (CSR) within the RISC-V core, thereby enabling the user to choose either higher precision or higher dynamic range for the same *ps* size anytime during the execution of a process through necessary system calls. The selection may be made at the beginning of the execution of a process or in the middle. The latter scenario arises in the case that some part of the program needs different precision than the remaining.

To enable compatibility of *posit* with the existing tool-chain, some recent works such as References [23, 43, 45] suggest converting IEEE-754 to *posit* in hardware. However, it is evident that such conversion schemes cannot leverage the benefits of *posit* format. A complete software stack support is the only way forward to leverage the benefits of *posits*. Hence we modified the RISC-V GCC to handle floating-point literals as *posits*.

In summary, the contributions of this article are as follows:

- (1) The article provides insights on how the RISC-V ISA can be leveraged, modified, and customized to support *posit*-based floating-point computation;
- (2) Design and implementation details of all functional units required to build a RISC-V enabled *posit* FPU have been proposed. These implementations are parameterized for (*ps*, *es*) values and have been designed using Bluespec System Verilog (BSV);
- (3) The proposed *posit* FPU has been integrated with an open source RISC-V SHAKTI C-class core;
- (4) The *posit* FPU has been further enhanced with minimal overheads, to support two *es* values in the same hardware, thereby enabling dynamic switching between higher precision or higher dynamic range during application execution;
- (5) The RISC-V GCC has been modified to support compilation of C-code containing floating-point literals into *posit* format enabling direct execution on the *posit* enabled RISC-V core;
- (6) The article presents an alternate methodology of integrating the *posit* FPU with RISC-V core as an accelerator;
- (7) The article presents the analysis of various software applications running on the core and compares the results with an equivalent RISC-V processor with an IEEE-754 FPU. To the best of our knowledge, this is the first reported comparison of complete applications executing on *posit* versus those executing on IEEE-754-based processors.

The rest of the article is organized as follows: Section 2 provides a brief introduction and background to the *posit* format and the RISC-V ISA. Section 3 discusses the details of extending the RISC-V ISA to accommodate *posit* arithmetic operations. The compiler changes required to run applications on a *posit*-based RISC-V core are covered in Section 4. The entire *posit* FPU is discussed in detail in Section 5. The integration of the proposed *posit* FPU with a RISC-V core is highlighted in Section 6. Section 7 presents the results and insights obtained by executing applications (commonly found in edge-based devices like smartphones), on PERI. The hardware

results are reviewed in Section 8. In Section 9, we propose an alternate methodology of integrating the *posit* FPU with RISC-V core. Section 10 presents a comparative study with the existing literature. Section 11 concludes the article.

2 BACKGROUND

2.1 The *Posit* Format

The *posit* representation is defined using two parameters: the *ps* and *es*. A formal representation of a *posit* number P is shown in Equation (1) [15]. The value x of a *posit* number P is determined by Equation (2), where s , exp , and f are, respectively, the sign, exponent, and fraction of P . The first bit of P (from the left), is the sign bit, and a *posit* number utilizes the 2's complement notation to represent a negative number,

$$P = \underbrace{\overbrace{s}^{\text{Sign}} \underbrace{r \ r \ r \ \dots \ r \ \bar{r}}^{\text{Regime}} \underbrace{e_1 \ e_2 \ e_3 \ \dots \ e_{es}}^{\text{Exponent, if any}} \underbrace{f_1 \ f_2 \ f_3 \ f_4 \ \dots}_{\text{Fraction, if any}}}_{ps}, \quad (1)$$

$$x = \begin{cases} 0 & P = 000\dots000 \\ NaR & P = 100\dots000 \\ (-1)^s \times 2^{exp} \times 1.f & \text{otherwise} \end{cases} \quad (2)$$

The identical bits following the sign bit are called the regime bits, and its binary value is denoted by rc . The es number of bits after the regime bits represent the value e . The exponent of the *posit* number is thus derived from rc , es , and e using Equation (3). The remaining trailing bits of P represent the fraction f ,

$$exp = \begin{cases} ((rc - 1) \ll es) + e & \text{if regime starts with 1} \\ (-rc \ll es) + e & \text{otherwise} \end{cases}. \quad (3)$$

As can be seen from Equation (2), the only two exception values in *posit* representation are 0 and Not-a-Real (NaR) numbers. This makes exception handling very simple in *posit* as compared to IEEE-754.

A significant difference between *posit* and IEEE-754 representation is that *posit* uses regime bits to represent the exponent value. This enables a *posit* number with a small value of es to represent higher precision numbers than IEEE-754. Similarly, larger es values can express a much higher dynamic range than IEEE-754. Another important property of *posit* is that there is no notion of “unorderedness” in *posit*s meaning that the *posit* numbers are ordered similar to integers. This property is absent in IEEE-754. An example of interpreting a number in *posit* format is given below:

For $ps = 32, es = 2$

Sign	Regime	Exponent	Fraction
0	110	01	00001110011001100110011010
$\underbrace{\hspace{15em}}_{32}$			
$s = 0, rc = 2, e = 1, f = 00001110011001100110011010$			
$exp = ((2 - 1) \ll 2) + 1 = 5$			
$x = (-1)^0 \times 2^5 \times 1.00001110011001100110011010 = 33.80000019073486$			

2.2 The RISC-V ISA

RISC-V is a completely open ISA, which can be extended with custom instructions and operations. There have been significant efforts worldwide to build open source processors around RISC-V. Some of the prominent works that use RISC-V include SHAKTI [14], Rocket-Chip [2], lowRisc [28],

Table 1. List of Instructions Comprising the F Extension of RISC-V

Instructions	Description
FMADD.S, FMSUB.S, FNMSUB.S, FNMADD.S, FADD.S, FSUB.S, FMUL.S	Fused-Multiply-Add ops
FDIV.S	Division op
FSQRT.S	Square Root op
FSGNJ.S, FSGNJN.S, FSGNJX.S	Sign Injection ops
FMIN.S, FMAX.S, FEQ.S, FLT.S, FLE.S	Comparison ops
FCVT.W.S, FCVT.W.U.S, FCVT.S.W, FCVT.S.W.U	Conversion ops
FMV.X.W, FMV.W.X	Transfer ops
FCLASS.S	Classification op

Fig. 1. *Posit* control and status register (pcsr).

and Ariane [36]. Commercial entities such as Western-Digital and Bluespec Inc. have also contributed their core implementations (SweRV [9] and Piccolo/Flute [3] respectively) to the open source community.

RISC-V defines *I*, *M*, *A*, *F*, *D*, and *C* as the standard extensions of the ISA representing the integer, multiplication, atomic, single-precision floating-point, double-precision floating-point, and compressed instruction support, respectively. Each of the “F” and “D” extension in RISC-V ISA comprises instructions compliant with the IEEE-754 standard. The “F” extension requires a separate 32-bit floating-point register file while the support for “D” extension requires a 64-bit floating-point register file. Though the RISC-V specification defines these extensions as *standard extensions*, it also allows users to modify these extensions as per their requirement, while still being compliant with other standard instruction extensions. For example, one can modify the “F” and “D” extensions and still be compliant with the standard definitions of the *I*, *M*, *A*, and *C* extensions. In the next section, we leverage this opportunity and describe how the current “F” extension can be modified to support *posit* arithmetic. Though in this article, we restrict our discussion to the “F” extension alone, it is straightforward to extend the same concept to the “D” extension as well.

3 LEVERAGING RISC-V “F” EXTENSION FOR *POSIT*

This section describes how the RISC-V ISA can be leveraged and modified to include *posit*-based arithmetic operations. We leverage the “F” extension of the ISA with minimal modifications to support *posit*-based arithmetic. Before proceeding further, the reader is recommended to be cognizant with the “F” extension of the RISC-V ISA [47]. A gist of instructions comprising the “F” extension are captured in Table 1. This extension was designed to fit the IEEE-754 requirements. To support *posit*, some of these features were re-used, while others were modified or were not required. This section provides more details.

We maintain the same register-file state for *posit* as that of the “F” extension, i.e., 32 *posit* registers: p0-p31 each 32-bits wide. The *posit* variant(*pcsr*) of the existing floating-point control and status register(*fcsr*) is shown in Figure 1. We now discuss the modifications in the *pcsr* for *posit*s.

Since *posit* supports only one rounding mode (round-to-nearest with tie-to-even), the *rounding mode* (*rm*) field, specified as part of the “F” extension, is not required and thus tied to zeros. Similarly, there is no requirement of flags for invalid (NV), inexact (NX), overflow (OF), and underflow (UF). The *posit* exception of NaR is silent and thus does not get captured in the flags. The exception of divide-by-zero (DZ) is mapped to the DZ field of *fflags*. The *pcsr* register also holds a 5-bit *es-mode* field, which indicates the current value of *es* being used by the *posit* FPU and is required to deduce the *posit* number. To maintain compatibility with the “F” extension, the *ps* value is set to 32, and we expect all practical implementations of a *posit* FPU to support *es* values that can be represented within the 5-bit field. While the majority of implementations would support only a single value of *es* (thus causing this field to be read-only), later parts of this article propose a *posit* FPU design that can support up to two different *es* values, thereby using the *es-mode* field to perform the switching. This field can be modified using the standard CSR instructions of RISC-V. Implementations that support multiple *es-mode* values should rely on the software to perform a probe-and-find mechanism to identify all legal *es* values supported by the platform.

Regarding IEEE-754, the RISC-V specification mandates that any floating-point operation resulting in a NaN should output a canonical NaN (i.e., $0x7fc00000$). However, in *posit*, NaR has a single representation that maps to the most negative 2’s complement signed integer. The fact that there is no notion of “unorderedness” in *posits*, allows a user to leverage integer-based comparison techniques to compare *posit* numbers. Unlike IEEE-754, in which two NaNs are not equal even after a similar pattern, two NaNs in *posit* are always equal.

All instructions proposed in the “F” extension behave the same way for *posits* as they do for IEEE-754. The encoding of all instructions remains the same. The *rm* field in all instructions, except the *posit*-to-integer conversion ops, is ignored, since *posit* only supports a single rounding mode. For the *posit*-to-integer conversion ops, we realized that by supporting the round-to-zero (RTZ) mode, certain applications, like JPEG compression, can provide much better results compared to using the default round-to-nearest mode. Thus, we propose to keep the *rm* field in these instructions to mean the same as they do in the default specification.

In the *posit*-based computation, the different exceptional scenarios, including subnormals, various types of NaNs, zeros, and infinity, do not arise. Thus, the *posit* FPU needs to classify its input operand into four: zero, NaR, negative, or positive. Hence, the lower four bits are set in the output, and the remaining bits are set to zero.

4 COMPILER SUPPORT FOR POSIT

The compilation of any application containing floating-point literals using RISC-V GCC generates the IEEE-754 format for the respective literals and the instructions listed in the “F” extension of RISC-V. To support *posit*, we utilized the existing “F” extension of RISC-V for the *posit* FPU and modified the GCC to generate the *posit* representation instead of the IEEE-754 representation. This section describes the modification done in RISC-V GCC (version 7.2.0) to generate a *posit* format for floating-point literals.

GCC internally uses an intermediate format called `REAL_VALUE_TYPE` data structure to store a floating-point number. A floating-point literal/constant is first converted to the `REAL_VALUE_TYPE` structure and then converted to the target number format (IEEE-754 or *posit*). This conversion is carried out by the function `encode_ieee_single` within GCC. As part of this work, we have modified this function to generate a 32-bit *posit* number as per the format mentioned in Equation (1). We have further parameterized this function to be independent of the choice of *es* value, which needs to be defined during the build process of the tool-chain. With these changes in place, we re-build the tool-chain using the conventional process, thereby enabling the compilation/execution of any application with floating-point literals on the *posit*-based core.

Fig. 2. C-code and the corresponding Assembly code of data section after compilation with the modified riscv-gcc. (a) C-code. (b) Assembly code of data section.

Fig. 3. Block diagram showing the different components of *Posit* FPU.

Figure 2(a) shows a sample C-code containing floating-point literals. The modified GCC reads the floating-point literals 33.8 and 6.9, converts them to equivalent *posit* formats, and stores them in the data section, as shown in Figure 2(b). The *posit*-enabled processor can now access these numbers and perform *posit* arithmetic on the same.

5 POSIT FLOATING POINT UNIT

This section describes the various components of a *posit* FPU. The *posit* FPU reads operands in the *posit* format, which is generated by the modified RISC-V GCC. The result of the operation is also in the *posit* format. We have implemented our design using BSV. Our *posit* FPU is designed similar to Berkeley HardFloat [17] and parameterized to generate hardware for any combination of *ps* and *es* value. Figure 3 shows the various components present in our implementation. It consists of separate modules for operations such as Fused Multiply-Add (FMA), division (DIV) /square root (SQRT), *posit* to int conversion, and so on. The *posit* FPU has the following BSV interface definition.

```

interface Ifc_fpu;
    method Action _start(
        Bit#(32) rs1, Bit#(32) rs2, Bit#(32) rs3, Bit#(4) opcode,
        Bit#(7) funct7, Bit#(3) funct3, Bit#(2) imm, Bit#(5) es-mode);
    method ActionValue#(Bit#(32)) get_rd;
    method ActionValue#(Bit#(5)) get_fflags;
endinterface

```

The input parameters of the interface refer to the fields of Table 15 and 16. The following sub-sections will describe the different modules and the instruction they implement in the proposed *posit* FPU.

5.1 Posit Decoder

ALGORITHM 1: Posit Decoding

Input: P : floating-point number in *posit* format with ps bits
Output: s : sign of P , exp : final exponent of P , f : fraction bits of P , $f0$: set if P is 0, $fNaR$: set if P is NaR

```

1:  $s \leftarrow P[ps - 1]$  ▷ sign of  $P$ 
2:  $lz \leftarrow (P[ps - 2 : 0])$  ▷ set if lower bits are zero
3:  $f0 \leftarrow s \& lz$  ▷ set if  $P$  is 0
4:  $fNaR \leftarrow s \& lz$  ▷ set if  $P$  is NaR
5: if ( $s = 1$ ) then
6:    $P \leftarrow \sim P + 1$  ▷ 2's complement of  $P$ 
7:  $t \leftarrow P[ps - 2 : 0]$ 
8: if ( $P[ps - 2] = 1$ ) then
9:    $t \leftarrow \sim t$  ▷ 1's complement of  $t$ 
10:  $rc \leftarrow \text{countZerosMSB}(t)$  ▷ count identical bits
11:  $P \leftarrow P \ll (rc + 2)$  ▷ remove identical bits
12:  $e \leftarrow P[ps - 1 : ps - es]$  ▷ select  $e$  bits
13: if ( $P[ps - 2] = 1$ ) then ▷ calculate final exponent
14:    $exp \leftarrow ((rc - 1) \ll es) + e$ 
15: else
16:    $exp \leftarrow (-rc \ll es) + e$ 
17:  $P \leftarrow P \ll es$  ▷ remove  $e$  bits
18:  $f \leftarrow P[ps - 1 : es + 3]$  ▷ select fraction bits
19: return  $s, exp, f, f0, fNaR$ 

```

ALGORITHM 2: Posit Encoding

Input: rs : sign for P , $rexp$: exponent for P , rf : fraction bits for P , sb : sb is set if any of previous shifted bits are 1, $f0$: if P is 0, $fNaR$: if P is NaR
Output: P : final result in *posit* format with ps bits

```

1: Derived Parameters:  $fes: (\log_2(ps) + es + 2)$ 
2:  $e \leftarrow rexp[es - 1 : 0]$  ▷ select  $e$  bits
3:  $k \leftarrow rexp[fes - 1 : es]$  ▷ select  $k$  value
4:  $rc \leftarrow abs(k)$ 
5:  $ef \leftarrow e \parallel rf$  ▷ concatenate  $e$  and frac bits
6:  $rg[0] \leftarrow 1$  ▷ regime bits are 0 for negative exp
7: if ( $k \geq 0$ ) then
8:    $rg \leftarrow 2^{ps} - 1$  ▷ regime bits are 1 for positive exp
9:    $rc \leftarrow rc + 1$ 
10:   $rg[0] \leftarrow 0$ 
11:  $P' \leftarrow rg \parallel ef$  ▷ append  $e$  bits and frac to regime
12:  $P' \leftarrow P' \ll (ps - 1 - rc)$  ▷ shift  $P$  to finalize  $k$  value
13:  $P \leftarrow P'[2ps - 3 : ps - 2]$ 
14:  $rb \leftarrow P'[ps - 3]$ 
15:  $sb \leftarrow sb \mid (P'[ps - 4 : 0])$ 
16:  $rb \leftarrow rb \& (P[0] \mid sb)$  ▷ round-to-nearest
17: if ( $P = 2^{ps-1} - 1$ ) then
18:    $rb \leftarrow 0$  ▷ do not round if  $P$  is maxpos
19: if ( $P = 0$ ) then
20:    $P \leftarrow 1$  ▷ round up if  $P$  is zero
21:  $rb \leftarrow rb \oplus rs$ 
22: if ( $rs = 1$ ) then ▷ complement negative number
23:    $P \leftarrow \sim P$ 
24:  $P \leftarrow P + rb$ 
25: if ( $f0 = 1$ ) then ▷  $P$  is 0, if  $f0$  is set
26:    $P \leftarrow 0$ 
27: if ( $fNaR = 1$ ) then ▷  $P$  is NaR if  $fNaR$  is set
28:    $P \leftarrow NaR$ 
29: return  $P$ 

```

This block is responsible for extracting the sign, exponent, and fraction of each *posit* number P , introduced at its inputs. This unit is also responsible for detecting if the inputs are 0 or NaR and set the appropriate flags. The functionality of this unit is captured in Algorithm 1. Table 2 explains the various symbols and functions used in the algorithms.

Compared to IEEE-754, which requires detecting five different special values: subnormal, zero, qNaN, sNaN, and infinity, *posit* requires detecting only two special values. Unlike IEEE-754, the sign bit is the only field with a fixed position in a *posit* number. This sign bit indicates if the number is negative and if it is, a 2's complement of the number should be taken (lines 5 and 6). To extract the exponent and fraction, we count the number of regime bits (lines 7–10). If the regime starts with a 0, then the exponent is treated as negative, else positive. Note that the sign of the exponent in IEEE-754 is derived through a bias. Lines 11–16 extract the e bits by removing the regime bits and implement Equation (3) to capture the final exponent value. Lastly, fraction bits are obtained by shifting out the exponent bits (lines 17 and 18). Unlike IEEE-754, *posit* does not have subnormal numbers, eliminating the need to handle subnormal exponent and fraction separately, thereby simplifying the *posit* arithmetic. The example given below shows the exp and f value after decoding a *posit* number P .

Table 2. Symbols and Functions Used in Algorithms

$ P$	unary bit-wise OR of P
$\sim P$	unary bit-wise negation of P
$P[n]$	select n^{th} bit of P
$P[n-1:0]$	select 0 to $(n-1)^{th}$ bits of P
$P \ll n$	left-shift P by n
$P \gg n$	right-shift P by n
$a b$	bit-wise OR of a and b
$a\&b$	bit-wise AND of a and b
$a \oplus b$	bit-wise xor of a and b
$a b$	concatenate a and b
$abs(k)$	absolute value of k
$countZerosMSB(t)$	count number of leading zeros in t
$chkOF(f, e)$	check overflow of f and adjust f and e accordingly
$normalize(f, e)$	adjust f and e so that f is normalized
$round(f, rb)$	round f as per the rb bit

```

Example:
ps = 32, es = 2
P : 0110010000111001100110011010 (0x6439999A)
s = 0, rc = 2, e = 1
exp = ((2-1) << 2) + 1 -> 5
f = 00001110011001100110011010

```

Here we would like to emphasize that, as shown in Figure 3, each *posit* arithmetic unit maintains its own set of posit-decoder(s) instead of maintaining a single common posit-decoder across all units. This decision has been made from the perspective of comparing power, area, and performance of the *posit* FPU with the HardFloat implementation of the IEEE-754, which also uses a similar structure for its arithmetic units.

5.2 Posit Encoder

This block caters to creating the final *posit* representation from the results computed by various arithmetic compute blocks. The capabilities of this block are captured in Algorithm 2. To calculate the final exponent we first extract the e bits and the value of k (lines 2 and 3). A negative exponent is represented with the regime bits beginning with a 0 (1 otherwise) (lines 6–10). The regime bits, e bits, and fraction bits are concatenated and the position of regime bits are finalized as per rc (lines 11 and 12). The sticky bit (sb) is calculated as the Logical-OR of all the remaining bits ((line 15). Following this, the *posit* is rounded to the nearest number (with tie-to-even) in lines 16–24. If either of the input flags, 0 or NaR, are set, then the final result is modified accordingly (lines 25–28), else the rounded result is forwarded to the output. The example given below shows the encoding of $rexp$ and rf values to *posit* representation.

```

Example:
ps = 32, es = 2
rs = 0, rexp = 5, rf = 00001110011001100110011010
e = 01, k = 1, rc = 2
P : 0110010000111001100110011010

```

The *posit* scheme differs significantly from the IEEE-754 scheme when it comes to rounding. Only one rounding mode is supported in *posit* as compared to five in IEEE-754. A *posit* number neither overflows to infinity nor underflows to zero like IEEE-754; rather, they overflow or underflow to the nearest *posit* number. If the encoded number is $maxpos (2^{ps-1} - 1)$, then it is not rounded up irrespective of round bit (*rb*). Similarly, if the encoded number is 0, then it is rounded up irrespective of the round bit.

For the same reasons mentioned in the previous section, we instantiate each arithmetic unit with a posit-encoder where necessary, instead of using a single common posit-encoder across units.

5.3 Fused Multiply-Add

ALGORITHM 3: Fused Multiply-Add

Input: op_1, op_2, op_3 : input operands, ng : if negate operation, op : if sub operation
Output: P : result
1: **Derived Parameters:** $ffs: (2 \times (ps - es - 2))$
2: $s_1, exp_1, f_1, f_{01}, fNaR_1 \leftarrow decode(op_1)$
3: $s_2, exp_2, f_2, f_{02}, fNaR_2 \leftarrow decode(op_2)$
4: $s_3, exp_3, f_3, f_{03}, fNaR_3 \leftarrow decode(op_3)$
5: **if** $((fNaR_1 | fNaR_2 | fNaR_3) = 1)$ **then**
6: $fNaR \leftarrow 1$ $\triangleright$ final result is NaR
7: **if** $((f_{01} | f_{02} | f_{03}) = 1)$ **then**
8: $f_0 \leftarrow 1$ $\triangleright$ final result is 0
9: $s_3 \leftarrow s_3 \oplus op \oplus ng$ $\triangleright$ sub operation affects sign of op_3
10: $rs \leftarrow s_1 \oplus s_2 \oplus ng$
11: $rexp \leftarrow exp_1 + exp_2$ $\triangleright$ calculate exp for product
12: $rf \leftarrow f_1 \times f_2$
13: $rexp, rf \leftarrow chkOF(rf, rexp)$ $\triangleright$ check prod. overflow
14: **if** $(rexp < exp_3)$ **then**
15: $sw \leftarrow True$
16: **if** $(rexp = exp_3 \ \&\& \ rf < f_3)$ **then**
17: $sw \leftarrow True$
18: **if** (sw) **then** $\triangleright$ swap if product is smaller than op_3
19: $swap(rs, s_3)$
20: $swap(rexp, exp_3)$
21: $swap(rf, f_3)$
22: $ediff \leftarrow rexp - exp_3$
23: $sb \leftarrow |(f_3 \ll (ffs - ediff))|$ $\triangleright$ shifted bits are ORed
24: $f_3 \leftarrow f_3 \gg ediff$ $\triangleright$ align f_3 so that exps are equal
25: **if** $(rs = s_3)$ **then**
26: $rf \leftarrow rf + f_3$ $\triangleright$ add op for similar sign
27: $rf, rexp \leftarrow chkOF(rf, rexp)$
28: **else**
29: $rf \leftarrow rf - f_3$ $\triangleright$ sub op for diff sign
30: $rf, rexp \leftarrow normalize(rf, rexp)$
31: $P \leftarrow encode(rs, rexp, rf, sb, f_0, fNaR)$
32: **return** P

ALGORITHM 4: Division/ Square root

Input: op_1, op_2 : input operands, sq : if square root operation
Output: P : result, dz : exception flag
1: $s_1, exp_1, f_1, f_{01}, fNaR_1 \leftarrow decode(op_1)$
2: $s_2, exp_2, f_2, f_{02}, fNaR_2 \leftarrow decode(op_2)$
3: **if** $(f_{01} = 1)$ **then**
4: $f_0 \leftarrow 1$ $\triangleright$ final result is 0
5: **if** $(sq=1)$ **then** $\triangleright$ sqrt operation
6: **if** $((fNaR_1 | s_1) = 1)$ **then**
7: $fNaR \leftarrow 1$ $\triangleright$ final result is NaR for sqrt
8: $rs \leftarrow s_1$
9: $rexp \leftarrow exp_1 \ll 1$ $\triangleright$ calculate exp for sqrt
10: **else** $\triangleright$ div operation
11: **if** $((fNaR_1 | fNaR_2) = 1)$ **then**
12: $fNaR \leftarrow 1$ $\triangleright$ final result is NaR for div
13: **if** $(f_{02} = 1)$ **then**
14: $dz \leftarrow 1$ $\triangleright$ update DZ flag in *pcsr*
15: $rs \leftarrow s_1 \oplus s_2$
16: $rexp \leftarrow exp_1 - exp_2$ $\triangleright$ calculate exp for div
17: $rf, sb \leftarrow div_sqrt(f_1, f_2, sq)$ $\triangleright$ multi-cycle algo
18: $P \leftarrow encode(rs, rexp, rf, sb, f_0, fNaR)$
19: **return** P, dz

The “F” extension of RISC-V ISA specifies four different fused ops, FMADD.S, FMSUB.S, FNMSUB.S, and FNMADD.S. These operations are carried out using Algorithm 3. The *ng* input bit indicates a negate operation, while the *op* input bit indicates a subtract operation. We have configured this block to support not only fused operations but also simple operations like FADD.S, FSUB.S, and FMUL.S, enabling maximum resources to be re-used across operations.

The FMA block checks whether either of the inputs is 0 or NaR. The corresponding circuitry in IEEE-754 would require checking for five exceptional cases per operand. Fused operations in IEEE-754 require checking the intermediate product exponent for underflow or overflow, which is absent in *posit*. Additionally, IEEE-754 also requires normalizing the product’s fraction in the case

of subnormal numbers, while *posit* only requires checking the overflow of the product's fraction. *Posit* do not set any flag in the *pcsr* register in contrast to IEEE-754 that can set NV, OF, UF, and NX flags in *fcsr*.

5.4 Division/Square-Root (FDIV/FSQRT)

The *Posit* FPU has a common module for implementing iterative FDIV.S/FSQRT.S as Algorithm 4. A divide by zero exception in FDIV.S is captured by setting the DZ flag in *pcsr* (lines 13 and 14). IEEE-754, however, has to account for setting all five flags for FDIV.S. FSQRT.S for *posit* does not set any flags in *pcsr* and sets the NX flag for IEEE-754. The sign and exponent of the result are calculated as per the requested operation. This block uses an iterative algorithm for computing the quotient/root of the fractions (line 17). The number of cycles required for the operation is proportional to the size of the fractions. In each cycle, one iteration of the non-restoring division/square root is performed. The non-restoring algorithm returns the quotient/root of the fractions. Similarly to FMA, IEEE-754 division/square root has to normalize subnormal numbers before fraction quotient/root calculation while *posit* incur no such hardware. Also, IEEE-754 checks the division exponent for overflow and underflow while *posit* do not.

5.5 Integer to *Posit* Conversion

ALGORITHM 5: Integer to *Posit* Conversion

Input: I : integer of ps bits, u : is set if I is unsigned
Output: P : result
 1: $rs \leftarrow I[ps - 1]$ ▷ sign of I
 2: $rs \leftarrow rs \& \sim u$ ▷ do not consider sign if unsigned int
 3: **if** ($rs = 1$) **then**
 4: $I \leftarrow I + 1$ ▷ 2's complement
 5: $z \leftarrow \text{countZerosMSB}(I)$
 6: $I \leftarrow I \ll z$ ▷ shift I to finalize fraction
 7: $rexp \leftarrow ps - 1 - z$ ▷ calculate final exponent
 8: $rf \leftarrow I[ps - 2 : es + 1]$ ▷ select fraction bits
 9: $P \leftarrow \text{encode}(rs, rexp, rf, 0, 0, 0)$
 10: **return** P

ALGORITHM 6: *Posit* to Integer Conversion

Input: op_1 : input operand, u : is set if result is unsigned, rm : rounding mode
Output: I : integer value corresponding to op_1
 1: **Derived Parameter:** $fs: (ps - es - 3)$
 2: $s_1, exp_1, f_1, f0_1, fNaR_1 \leftarrow \text{decode}(op_1)$
 3: $f_1 \leftarrow 0_1 0_2 \dots 0_{ps} \parallel f_1$ ▷ zero extend f_1 to $ps+fs$ bits
 4: $f_1 \leftarrow f_1 \ll exp_1$
 5: **if** ($u = 0$) **then** ▷ I is signed
 6: **if** ($exp_1 < ps - 1$) **then**
 7: $I \leftarrow f_1[ps + fs - 1 : fs]$
 8: **else**
 9: $I \leftarrow 2^{ps-1} - 1$ ▷ set I to max signed int
 10: **else** ▷ I is unsigned
 11: **if** ($exp_1 < ps$) **then**
 12: $I \leftarrow f_1[ps + fs - 1 : fs]$
 13: **else**
 14: $I \leftarrow 2^{ps} - 1$ ▷ set I to max unsigned int
 15: $rb \leftarrow f_1[fs - 1]$
 16: **if** ($rm = 1$) **then** ▷ check for round-to-zero
 17: $rb = 0$
 18: $I \leftarrow \text{round}(I, rb)$
 19: **return** I

FCVT.S.W/FCVT.S.WU instructions are used to convert a signed/unsigned value to *posit*, respectively. The conversion steps are highlighted in Algorithm 5. The u input bit indicates if the input is unsigned, in which case the negative sign is cleared (line 2). To get the actual exponent of *posit*, we set the maximum exponent according to the ps value. Then we count the number of leading zeros and subtract this count from the maximum exponent. The integer value is shifted left to get the fractional part of *posit* (lines 5–8). Integer-to-float (I2F) conversion operation for *posit* does not set any exceptional flag while IEEE-754 conversion involves the setting of the NX flag.

5.6 *Posit* to Integer Conversion

The “F” extension provides FCVT.W.S/FCVT.W.U.S instructions to convert *posit* to signed/unsigned integers respectively. These instructions are implemented using Algorithm 6. The *u* input bit indicates if the result should be unsigned. The extended *posit* fraction is left-shifted by the exponent (lines 3 and 4). The final integer value is calculated based on the *u* and *exp* inputs (lines 5–14). For FCVT.W.S and FCVT.W.U.S instructions, we propose to support an additional rounding mode (RTZ mode) along with the default *posit* rounding mode (motivation of this is discussed in Section 7.2). Thus, when the *rm* is RTZ, *rb* is set to 0 (line 16). IEEE-754 performs several checks for setting the NV and NX flags during float-to-integer (F2I) while *posit* does not set any flags as a consequence of these instructions.

5.7 *Posit* Comparison

FMIN.S, FMAX.S, FEQ.S, FLT.S, and FLE.S instructions are defined in the “F” extension to compare floating-point numbers. The *posit* representation resembles the 2’s complement representation of an integer. Hence, the comparison between two *posits* is exactly similar to comparing the integer value of the bit representation. There are no exceptional cases in the *posit* comparison operation. This is one of the simplifications in *posit* arithmetic compared to IEEE-754 arithmetic. Additionally, We have utilized integer comparison logic for *posit* comparison; hence, the need for a comparator in FPU is eliminated. Comparison in IEEE-754 requires checking of different exceptional cases such as +0 and −0, since they have different representations but are treated as equal. Similarly, two NaNs having similar bit representations are treated as different. IEEE-754 sets the NV flag for comparison operation (whenever applicable) while *posit* does not set any flags.

5.8 *Posit* Sign Injection

RISC-V ISA defines FSGNJ.S, FSGNJN.S, and FSGNJX.S instructions for floating-point sign injection. FSGNJ.S is used to move values between registers, FSGNJN.S is used to negate a floating-point number, and FSGNJX.S is used to get the absolute value of floats. The negation operation for a IEEE-754 number is just flipping of sign bit, but in the case of *posit* number, 2’s complement is taken to negate a number or calculate the absolute value of a negative number. Both *posit* and IEEE-754 do not set any exception flags for this operation.

5.9 *Posit* Classification

FCLASS.S instruction has been defined to determine the category of a floating-point number. IEEE-754 have different categories that are ± 0 , $\pm$ infinity, $\pm$ subnormals, $\pm$ normals, sNaN, and qNaN. Hence IEEE-754 needs to check for all those values. *Posit* only needs to check for 0, NaR, +ve, and −ve numbers; hence the logic is much simpler. Classify instruction also does not set any flags for either *posit* or IEEE-754.

5.10 Dynamic Switching

Several scientific applications like weather forecasting, automotive design and safety, and so on, demand high precision floating-point calculations. However, applications in the deep learning domain require a large dynamic range. One would require two separate designs to cater to both the demands. In regards to this, we propose a single *posit* FPU that can fulfill the requirement of high precision and high dynamic range applications within the same design. We enhance the proposed hardware unit of 32-bit *posit* to support two different *es* values ($es = 2$ and $es = 3$). The *posit* FPU can, thus, switch across various *es* values at runtime by manipulating the *es-mode* field in the *pcsr* register. We call this capability as *Dynamic Switching*. *Dynamic Switching* allows increasing the

Table 3. Instruction Format for Switching Operands across es Values

31	25	24	20	19	15	14	12	11	7	6	0
1111100		to-es		from-es		000		rs1/rd		opcode	

range of *posit* system during application execution when the result of the calculation exceeds the current range. For example, if the result of the calculation reaches the maximum *posit* number (with $es = 2$), the *posit* FPU can switch to $es = 3$ and continue the calculation.

For supporting dynamic switching, we would also need instruction support to convert *posit* numbers encoded with one es value to a *posit* number encoded with a different es value. In view of this, we introduce a new single-operand instruction: *FCVTES* formatted as shown in Table 3. The *from-es* field indicates the 5-bit es value with which the current register rs1 has been encoded, while the *to-es* field indicates the target es value to which the number should be encoded. While switching from one es value to another, if the number is not exactly representable in the target es domain, *posit* rounding logic ensures that the number is rounded correctly.

To support two different es values we have made few modifications in the parameterized design presented in Section 5. The modifications are as follows. In case of runtime defined es value, the exponent size is determined by the largest es value ($es = 3$) and the fraction size by smallest es value ($es = 2$). Along with this, few changes are incorporated in encode and decode modules to handle two es values.

For the *posit* decode module, Equation (4) adjusts e bits for $es = 2$ after line 12 of Algorithm 1,

$$e = e \gg 1. \quad (4)$$

For the *posit* encode module in Algorithm 2, Equation (5) shifts the concatenated ef bits for $es = 2$ and Equation (6) adjusts k value for $es = 3$ after line 5,

$$ef = ef \ll 1, \quad (5)$$

$$k = k \gg 1. \quad (6)$$

To switch from one es value to another, we first decode the *posit* number as per the *from-es* value and then encode the sign, exponent, and fraction as per the *to-es* value. The encode and decode modules are already available in the *posit* FPU, and the minimal changes mentioned above allows them to support two es values. Thus, the overheads for dynamic switching are minimal and discussed in Section 8. There is no impact of dynamic switching on application runtime as context switching and pipeline flush are not involved in the process. The es value to be used gets updated in the *pcsr* register. The updated es value is then used for decoding the operands from the next instruction onward.

6 INTEGRATION WITH CORE

This section describes how the proposed *posit* FPU can be integrated with a standard RISC-V core. We have chosen the SHAKTI C-class core [14] as our baseline core for integration. C-class provides a highly configurable in-order, RISC-V open source platform. The C-class core is designed using BSV [20] and is a basic five-stage in-order core, which includes: a branch predictor, blocking instruction and data caches, fully associative Translation Look-aside Buffers for instruction and data, a hardware page-table-walk unit, and AXI-4 compliant system bus interface. Figure 4 shows a high-level micro-architecture of the core.

For the purpose of this work, we have configured the core to support the RV32IMAFIC extensions of the ISA (i.e., it supports, 32-bit Integer (I), multiplication/division (M), atomic

Fig. 4. Micro-architecture of the C-class core with *posit* FPU integrated as a tightly coupled execution unit.

(A), single-precision floating (F), and compressed (C) ISA extensions). The *posit* FPU has been instantiated with parameter settings: $ps = 32$ and $es = 2$.

The default C-class core includes a single-precision IEEE-754-compliant FPU, which has the following BSV interface definition.

```
interface Ifc_fpu;
  method Action_start(
    Bit#(32) rs1, Bit#(32) rs2, Bit#(32) rs3, Bit#(4) opcode,
    Bit#(7) funct7, Bit#(3) funct3, Bit#(2) imm, Bit#(3) csr);
  method ActionValue#(Bit#(32)) get_rd;
  method ActionValue#(Bit#(5)) get_fflags;
endinterface
```

Since our *posit* FPU implements the same interface, except for the CSR (*es-mode* in case of *posit*) field being 5-bits in *posit* to include the *es* value, we are able to effortlessly replace the IEEE-754 FPU with our *posit* FPU.

As mentioned in Section 5.7, the comparison operations for *posit* can re-use the integer comparison hardware blocks. In the C-class, the branch unit in the execution stage compares the signed/unsigned integers. We modify this block to receive inputs for *posit* comparison instructions as well.

Figure 4 shows the micro-architecture of the *posit* FPU integrated with the C-class core as a tightly coupled execution unit. The flag updates in the *pcsr* happen similar to how the IEEE-754 unit updates the flags, i.e., in the write-back stage.

7 EVALUATION OF THE POSIT RISC-V CORE

In this section, we present details of porting several applications on our *posit*-enabled C-class core. These applications have been executed on the C-class core with *posit* FPU integrated as a tightly coupled execution unit. Additionally, we also provide results and insights of running the same applications with IEEE-754 on the default C-class core.

7.1 Checking Correctness

We have used SoftPosit [27] library for verifying our *posit* implementation. The verification effort for the design can be extended for $es = k$, but we have verified it extensively for $es = 2$ because SoftPosit library currently supports only $es = 2$. Random inputs were generated, and the corresponding outputs of each operation (with $es = 2$) were found to be in agreement with the result of

Fig. 5. Images used in image processing applications.

Table 4. Original and Compressed Image Size in kBs after JPEG Compression with 32-bit *posits* and IEEE-754, Where Original Images Refer to Three Different Variants of Figure 5(a)

Image #	Original	<i>Posit</i> (RNE)	<i>Posit</i> (RTZ)	IEEE-754
Image 1	3.7 KB	1.9 KB	1.5 KB	1.5 KB
Image 2	4.3 KB	2.3 KB	2.0 KB	2.0 KB
Image 3	5.6 KB	3.0 KB	2.5 KB	2.5 KB

the SoftPosit library. Along with the random tests, we have also verified our designs for special cases. Apart from the two exceptional values (0 and NaR) defined in *posit* representation, we have also verified our design for the smallest and largest positive as well as negative numbers that can be represented in the *posit* system (for both $es = 2$ and $es = 3$). Verification for $es = 3$ has been done manually for corner cases. Such manual verification can be extended to other es values ($es = k$) as well. Moreover, the design generator is common for any *posit*-size and exponent-size value, and we have extensively verified one instance ($es = 2$) of the design.

7.2 Image Processing

We performed image processing tasks like JPEG compression [46] and edge detection (using Canny edge detector [4]), with *posit* and IEEE-754 FPUs on the images given in Figure 5. In all the applications, the image inputs were represented as an array of integer values. We compressed three different variants of a image (Figure 5(a)) using IEEE-754 and *posits*. The original and compressed image sizes (in KBs) are given in Table 4. We observed that *posit*, with the default rounding mode (round-to-nearest), produced larger compressed images as compared to those obtained using IEEE-754. Based on our analysis, the culprit seems to be the rounding mode used while converting *posit* to integer. However, we observed that if the conversion operation supported the RTZ rounding mode, the compressed image size matched with IEEE-754. This observation and study form the basis of our proposal to maintain two rounding modes for the *posit*-to-integer instruction in Section 5.6. Similarly, with RTZ rounding mode in *posit*, we detected similar edges (Figure 5(c)) from Figure 5(b), using 32-bit IEEE-754 and *posits*.

7.3 Trigonometric and Exponential Series

Applications using trigonometric equations form an excellent platform to demonstrate how *posit* outperforms IEEE-754 in terms of accuracy. We calculated sine, cosine, and exponential values for 32-bit IEEE-754 and *posit* ($es = 2$) using power series. The input for sine and cosine spans across values between 0° and 359° . In case of e^x input takes values between 0 and 11. For our metric of comparison, we have chosen the mean of the percentage error with respect to a double-precision IEEE-754 result. We further calculate the confidence interval [16] with confidence of 95% to estimate the range of mean percentage error. These results are tabulated in Table 5. We observed

Table 5. Mean and Confidence Interval of Percentage Error for 32-bit *posit* and IEEE-754 for Trigonometric and Exponential Series

$f(x)$	Metric	<i>Posit</i>	IEEE-754
sin(x)	mean	3.50E-05	2.32E-04
	confidence interval	(1.20, 5.80) E-05	(7.63, 38.8) E-05
cos(x)	mean	2.18E-05	1.18E-04
	confidence interval	(1.45, 2.86) E-05	(8.56, 15.1) E-05
e^x	mean	9.66E-07	6.30E-06
	confidence interval	(2.55, 16.8) E-07	(2.85, 9.76) E-06

Table 6. Mean and Confidence Interval of Percentage Error for 32-bit *posit* and IEEE-754 for Fast Fourier Transform

Comp.	Metric	<i>Posit</i>	IEEE-754
magnitude	mean	2.10E-05	2.56E-04
	confidence interval	(8.82, 33.1) E-06	(6.34, 44.8) E-05
angle	mean	5.02E-06	4.95E-05
	confidence interval	(4.16, 5.87) E-06	(3.82, 6.08) E-05

that the mean percentage error for *posit* is $7\times$ less in case of sin(x), $5\times$ less in case of cos(x), and $7\times$ less in case of e^x as compared to IEEE-754. Thus, we conclude from Table 5 that *posit* stores more information than IEEE-754 for the same bit-width and hence provides better accuracy.

7.4 Fast Fourier Transform

Applications like designing and using antennas, image processing and filters, data processing and analysis, and so on, use Fast Fourier Transform (FFT) for different purposes. We calculated FFT for a complex input vector with the real component as cosine values of numbers between 0 and 127 and the imaginary component as sine values. The resultant complex vector was converted to polar form. For this vector, we calculated the percentage error of 32-bit IEEE-754 and *posit* ($es = 2$) results for magnitude and angle separately, with respect to double-precision IEEE-754. The values obtained in Table 6 shows that the mean percentage error for *posit* is $12\times$ less in case of magnitude and $10\times$ less in case of angle with respect to IEEE-754.

7.5 *k*-means Clustering

The *k*-means algorithm is widely used in data science. We clustered some well-known datasets [13, 18, 41] using the *k*-means algorithm. After performing clustering, we calculated different cluster quality metrics [33]. The values of all the metrics lie between 0 and 1, and a score near 1 indicates better clustering. These metrics require the true and predicted label of data to measure the cluster quality. The true labels are present in the dataset, and predicted labels are generated by 32-bit IEEE-754 and *posit*. We observed that 32-bit IEEE-754 and *posit* gave similar clusters for these datasets.

To demonstrate the superiority of *posit* over 32-bit IEEE-754, we generated a synthetic dataset of 100 instances of 1000 random points in two-dimension. Here, the true labels are generated using double precision IEEE-754 and predicted labels are generated by 32-bit IEEE-754 and *posit*. The results are presented in Table 7, where we consider *posit*/IEEE-754 to *pass* the case for a given *k* value if it is able to produce the desired number of clusters without throwing an exception. Our observation is discussed in the following sub-sections.

7.5.1 max-precision Mode. 32-bit *posit* ($es = 2$) provides higher precision than 32-bit IEEE-754. For each of the 100 instances, we compared the clustering quality across all metrics for *posit* and IEEE-754. We observed that neither of the floating-point representations presented an exception (underflow or overflow), i.e., all the instances for all values of *k* passed simulation. However, we observed that, for many instances, *posit* outperformed or matched the IEEE-754 clustering quality across all metrics. These numbers are tabulated in column-4 of Table 7. The remaining instances

Table 7. Comparison between Quality Metrics of Clusters Obtained Using k -means Algorithm for 32-bit IEEE-754 and *posit* in Max-precision and Max-dynamic Range Mode for Synthetic Dataset with Different k Values, Where k Indicates the Number of Clusters

k	<i>Posit</i> passed	max-precision		max-dynamic range	
		IEEE-754 passed	<i>Posit</i> is better or similar	IEEE-754 passed	<i>Posit</i> is better or similar
2	100	100	100	100	100
3	100	100	81	78	36
4	100	100	66	46	23
5	100	100	69	30	17
6	100	100	57	29	16
7	100	100	66	0	0

indicate that *posit* under-performed in all or some of the metrics as compared to IEEE-754. On average, *posit* provides similar or better results than IEEE-754 in 73% cases.

7.5.2 max-dynamic Range Mode. The dynamic range of 32-bit *posit* ($es = 3$) is greater than 32-bit IEEE-754. Hence, *posit* can handle much larger values compared to IEEE-754. For this exercise, we multiplied the random inputs of each instance with a large number and clustered them using both IEEE-754 and *posits*-based applications. We observed that as the value of k grew larger, more number of IEEE-754-based applications failed due to overflow exceptions being raised. These numbers are tabulated in column-5 of Table 7. However, the *posit*-based application presents no such error and converges on a solution for all instances for all values of k .

Similarly to the above experiment, we observed that on an average, for all instances for which the IEEE-754-based application passed, *posit* outperformed or matched their quality of clustering 51% of the time. If we include the failed IEEE-754 instances in this ratio, then the average bumps to 85%.

8 HARDWARE RESULTS

This section presents the synthesis results of the *posit* FPU designed based on the algorithms mentioned in Section 5. The encoder and decoder units are implemented as combinational units. The Division/Square-root module implements a single step of the non-restoring algorithm, which takes a maximum of 32 cycles (for $es = 2$). All the other compute blocks (FMA, F2I, I2F, etc.) are also implemented as combinational circuits. However, each of these has been pipelined by leveraging the re-timing option of modern synthesis tools. This design approach matches the one available in Berkeley HardFloat [17], thereby allowing us to compare *posit* and IEEE-754 on relatively similar grounds.

The synthesis of both the FPUs (IEEE-754 and *posit*) was carried out on the 65-nm Low-power TSMC library characterized at the worst-corner using the Cadence Genus Synthesis Solution 19.11-s087_1 tool. Each of the designs was constrained to a frequency of 400 MHz, 10% derating factor, 10% clock uncertainty, and 30% input/output delay. Table 8 shows the area utilization on a modular basis for IEEE-754 and *posit*-based implementations. The overheads, mentioned in this table, are with respect to the 32-bit IEEE-754 implementations.

It is observed that for modules such as the FMA, the encoder and decoder blocks consume nearly 25% of the design area. Unlike IEEE-754, which has a fixed size for exponent and fraction, *posit* does not have a fixed exponent and fraction size. In *posit* representation, the regime's size is not fixed, and we need to detect the size of regime bits during decoding using special circuitry.

Table 8. ASIC Synthesis: Modulewise Area for *posit* and IEEE-754 FPU with *posit* Variants
Being $es = 2$ and $es = 3$ for 32-bit at 400 MHz

Module	IEEE-754	<i>Posit</i> ($es = 2$)			<i>Posit</i> ($es = 3$)		
	Area (μm^2)	Area (μm^2)	Dec/Enc	Overhead	Area (μm^2)	Dec/Enc	Overhead
FMA	28448.64	46340.16	11754.24	62.89	45132.48	11657.76	58.65
DIV/ SQRT	15805.44	19658.88	14402.88	24.38	20226.72	14968.8	27.97
I2F	4193.76	4582.08	2595.84	9.26	4755.36	2745.12	13.39
F2I	4135.68	4206.24	1689.6	1.71	3673.92	1710.24	-11.17

Table 9. No. of Cycles Required by Each Module in the 32-bit *posit*
and IEEE-754 FPU at 400 MHz

Module	IEEE-754	<i>Posit</i> ($es = 2$)	<i>Posit</i> ($es = 3$)	<i>Posit</i> ($es = 2,3$)
FMA	6	6	6	6
DIV	[2,26,27]	[3,31,32]	[3,30,31]	[3,31,32]
SQRT	[2,24,25,26]	[3,29,30,31]	[3,28,29,30]	[3,29,30,31]
F2I	3	3	3	3
I2F	3	3	3	3

Table 10. ASIC Synthesis: Modulewise Area for *posit* FPU
with Dynamic Switching at 400 MHz

Module	<i>Posit</i> ($es = 2,3$)		
	Area (μm^2)	Dec/Enc	Overhead
FMA	52738.56	14668.32	13.81
DIV/ SQRT	21547.2	16266.24	9.61
I2F	5672.16	3305.76	23.79
F2I	4673.28	1973.28	11.10

Similarly, during encoding, it is required to set the regime bits in the result. Hence, the decoding and encoding module consumes more area due to the run-length encoding of the regime. In this work, we focus on comparing the hardware area of stand-alone FMA, DIV/SQRT, I2F, and F2I operations in *Posit* and IEEE-754. However, it is possible to optimize the area of *Posit* FPU by using common encoder and decoders for all operations.

Table 9 shows the number of cycles taken by different modules in *posit* and IEEE-754 FPU. The *posit* FMA, I2F, and F2I modules achieve the same number of pipeline stages as IEEE-754. Division/Square root operation uses iterative non-restoring algorithms to calculate quotient/root. Hence the number of cycles required is proportional to the fraction length. Since the fraction length is more in the case of *posits*, the number of cycles required is more than IEEE-754.

Table 10 shows the area overheads, for the same timing constraints, of the *posit* unit supporting dynamic switching as compared the baseline *posit* implementation with $es = 2$.

8.1 Application Runtime

Table 9 shows the number of cycles taken by different operations in *posit* and IEEE-754 FPU. The *posit* FMA, I2F, and F2I operations require the same number of cycles as IEEE-754, whereas Division/Square root operation requires more cycles for *posit*. We executed applications mentioned

Table 11. Runtime of Applications Using IEEE-754 and *Posit* FPU's Operating at 400 MHz

Application	IEEE-754 runtime (in seconds)	<i>Posit</i> runtime (in seconds)
Edge Detection	30.754	30.97
JPEG Compression (Image 1)	13.23	13.365
JPEG Compression (Image 2)	19.789	19.991
JPEG Compression (Image 3)	27.72	28.006
$\sin(x)$	1.126	1.175
$\cos(x)$	1.136	1.185
e^x	0.025	0.026
FFT	5.305	5.501

Table 12. Modulewise Area of PERI and PERC for 32-bit *Posit* with $es = 2$

Module	PERI		PERC [29]	
	LUTs	FFs	LUTs	FFs
Decoder	141	0	220	0
Encoder	238	0	284	0
FMA	1797	0	1740	0
DIV/SQRT	794	184	1033	145
I2F	295	0	347	0
F2I	323	0	422	0

Table 13. Time Taken by Different Operations in PERI and PERC for 32-bit *Posit* ($es = 2$)

Operations	PERI Delay (in ns)	PERC [29] Delay (in ns)
FMA	47.46	54.394
I2F	19.161	19.997
F2I	17.012	21.322

in Section 7 on the *posit* and IEEE-754-based RISC-V cores and compared the runtime. Table 11 gives the comparison between runtime of applications on IEEE-754 and *posit* FPU's. We observe that the application runtime is slightly higher for *posit* FPU as the *posit* division/square root takes more cycles than IEEE-754.

8.2 Comparison with Existing *Posit* FPU

From the available open source implementations of *posit* arithmetic, we have compared our work with PERC [29] due to the similar design of *posit* FPU. We have synthesized the designs for an Artix-7 FPGA (xc7a100tcs324-1) device using Xilinx Vivado 2018.3. The FMA, I2F, and F2I operations are implemented as combinational logic, whereas division/square root is implemented as a multi-cycle operation. Table 12 gives the module-wise area of PERI and PERC for 32-bit *posit* with $es = 2$. We can see from Table 12 that PERI consumes less LUTs than PERC for all the modules except FMA. Also, PERI takes more FFs than PERC for division/square root module. Table 13 shows

Table 14. RoCC Interface Standard Instruction Format

31	25	24	20	19	15	14	13	12	11	7	6	0
fn		rs2		rs1		xd	xs1	xs2		rd		op

Table 15. RISC-V 32-bit Instruction Types Used in RV32F

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
rs3		funct2		rs2		rs1		funct3		rd		opcode		R4-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3	imm[4:0]			opcode		S-type

that time taken by PERI for 32-bit *Posit* ($es = 2$) FMA and F2I operation is slightly less, and I2F operation is comparable to PERC.

9 POSIT ACCELERATOR

While the “F” extension of RISC-V includes a nice subset of standard floating-point instructions that can cater to a wide variety of applications, more often than not, there is always a need to extend ISA to add more complex functionalities. Given the fact that the “F” extension is reserved and cannot be extended or modified for custom use, one would have to resort to the custom opcode space of RISC-V to extend ISA support. In this section, we propose an alternate methodology of integrating the *posit* FPU with the RISC-V core as an accelerator using custom opcodes. The approach is discussed in detail in the following subsections.

9.1 Leveraging the “Custom” Opcode Space

The approach relies on utilizing the *custom* major-opcodes space of the 32-bit instruction format. This opcode space has been frozen by the ISA to integrate custom instruction sets and be future compatible. In this article, we adopt the Rocket Custom Co-processor (RoCC) [50] interface for our *posit* FPU co-processor. All RoCC compliant accelerators follow a standard instruction format, as shown in Table 14. The *xs1*, *xs2*, and *xd* bits control how the base integer registers are read and written by the accelerator instructions. If the *xs1/xs2* bit is set, then the corresponding integer register specified by *rs1/rs2* is passed on to the accelerator. If the *xs1/xs2* bit is clear, then the accelerator can either re-use the *rs1/rs2* field to encode other information or use it to access its own register-file (the *posit* register file in our case). Setting the *xd* fields operate in a similar manner but performs writes instead of reads on the respective register. RoCC not only enables the transfer of register-file data but also equips the accelerator with a memory-interface.

The challenge of using a co-processor interface is defining the instructions and their binary encoding. The RISC-V ISA has reserved four major opcodes, namely *custom-0/1/2/3*, for non-standard extensions. We shall leverage these opcodes for our RoCC-based *posit* FPU accelerator. For simplicity, we will map the instructions of the “F” extension onto the RoCC format, as shown in Table 14. This requires, understanding the various instruction types of the “F” extension (shown in Table 15) and creating equivalent instruction types using the custom opcode space. From Table 15, one can infer that the “F” extension uses: 9 I-type, 1 S-type, 4 R4-type, and 12 R-type instructions. As discussed earlier, *posits* support only a single rounding mode, and thus the *rm* field of the particular instructions can be ignored completely. This further enables us to encode more information within the instruction.

Table 16. Equivalent Mappings of Table 15 Instruction Formats to RoCC Instruction Format

31	27	26	25	24	20	19	15	14	13	12	11	7	6	0		
funct7				rs2		rs1		xd	xs1	xs2		rd	custom-0/1/2/3		R-type	
rs3	funct2			rs2		rs1		xd	xs1	xs2		rd	custom-0/1/2/3		R4-type	
imm[11:0]								rs1		xd	xs1	funct1		rd	custom-0/1/2/3	I-type
imm[11:5]					rs2		rs1	funct1	xs1	xs2		imm[4:0]	custom-0/1/2/3		S-type

Fig. 6. Micro-architecture of the C-class core with *posit* FPU integrated as an accelerator through RoCC.

Table 16 shows the proposed RoCC format-based instruction mappings of the instructions in the “F” extension. The encoding does not include a separate *xs3* field, since, for all floating operations, the *rs3* value is always read from floating register-file. A quick observation also reveals that the custom space for *posit* can accommodate up to 512 R-type and 16 R4-type unique instructions, which is much larger than what can be achieved by the standard “F” extension of the ISA. Additionally, from the I-type format, one can either have up to 8 instructions that utilize immediate and *rs1* or can have up to 32K single-operand instructions (i.e., only *rs1* is used. E.g., FSQRT). Similarly, one can implement eight unique store-like instructions, employing the S-type format.

9.2 Integration with RISC-V Core

Figure 6 shows how the proposed *posit* FPU is integrated with the C-class core as an accelerator. The *posit* FPU is connected to the core at the write-back stage through the RoCC interface. The *posit* FPU also is given access to the data-cache through the RoCC interface to carry out memory-related operations. In this approach, the *posit* register file is maintained within the accelerator rather than in the core. The register-file within the core would include the integer and IEEE-754 register files.

We leverage the *major-custom* opcodes of the RISC-V ISA to facilitate a *posit*-based arithmetic accelerator for integration with a RISC-V core. When an instruction containing any of the custom opcodes is detected in the decode stage, the *xs1*/*xs2* fields are checked to see if any of the integer registers are required. The execute and memory stage bypass this instruction without any changes. The write-back stage off-loads this instruction (along with the integer operands, if any) to the *posit* accelerator and waits for an execution-complete response from the accelerator. Depending on the value of the *xd* field in the custom instruction, the write-back updates the relevant integer registers with the accelerator’s response.

Table 17. Advantages and Disadvantages of Integrating the *posit* FPU within the Core Using “F” Extension or as an Accelerator Using RoCC

Integration within the core using “F” extension	Integration as an accelerator using RoCC
Advantages: <ul style="list-style-type: none"> • Requires minimal changes in the software tool-chain and enables quick bring up of the design. 	Disadvantages: <ul style="list-style-type: none"> • Requires adding new instructions in the compiler, which increases the design time.
Disadvantages: <ul style="list-style-type: none"> • Functionality of the FPU is limited to “F” extension. • The <i>posit</i> FPU cannot be integrated to a stand-alone core such as a hard IP. • Does not allow to leverage the benefits of both IEEE-754 and <i>posit</i> formats in the same core. • Maximum frequency of the FPU is restricted by the frequency of the core. 	Advantages: <ul style="list-style-type: none"> • Allows extending the ISA to add more complex functionalities that are not defined in the “F” extension. • Enables a stand-alone <i>Posit</i> FPU design, which can be integrated to any other RISC-V core, which adheres to a standard co-processor interface. For example, we can integrate the <i>Posit</i> FPU as an accelerator with a hard IP. • Empowers the co-existence of IEEE-754 and <i>Posit</i> FPU in the same design. • Reduces the risk of modifying the pipeline of the core.
Applications:	
<ul style="list-style-type: none"> • Applications that use only <i>posit</i> format and the instructions present in RISC-V “F” extension can use this integration approach. 	<ul style="list-style-type: none"> • Applications that require both <i>posit</i> and IEEE-754 format and explore more <i>posit</i> operations such as log or exponential, which are not part of the current “F” extension, can utilize this integration approach.

9.3 Compiler Support

To provide compiler support for *posit* accelerator, one would require to add new instructions in the RISC-V GCC. These new instructions will be mapped to the custom opcodes of RISC-V. The decode unit should forward these instructions to the *posit* accelerator.

9.4 Performance

There is no significant performance difference between the two integration approaches. In the integration as a tightly coupled execution unit, the decode stage identifies the instruction as “F” extension, and the execute stage offloads computation to the *Posit* FPU. In the *Posit* accelerator case, the decode stage identifies the instruction as custom opcode; the execute and memory stage bypass the instruction, and the write-back stage offloads computation to the accelerator. Hence, the number of cycles required to execute the applications in both approaches is the same. The only significant difference between the two approaches lies in the decoding mechanism. In the case of RoCC, the decoder is minimal in the pipeline, and a significant part of the decoder is in the accelerator, whereas in the case of “F” extension, the decoder is entirely within the pipeline. The advantages and disadvantages of integrating the *posit* FPU as an accelerator with the RISC-V core compared to integration as a tightly coupled execution unit, replacing the IEEE-754 FPU, are listed in Table 17.

Table 18. Comparison of PERI with Related Works

	PERI (This work)	PERC [29]	CLARINET [21]	Calligo ¹ Technologies [43]	Vivid ¹ Sparks [45]
Implements all Instructions in “F” Extension	✓	✓	✗	✓	✓
Comparison with IEEE-754	✓	✗	✗	✗	✗
RISC-V Integration	✓	✓	✓	✓	✓
Application Demonstration	✓	✗	✓	✗	✗
RISC-V GCC Support	✓	✗	✗	✓	✓
Open Source	✓	✓	✓	✗	✗

10 RELATED WORK

This section provides a comparison of the contributions of this article with related works in literature. Initial *posit* arithmetic implementations [22, 23] present hardware units for addition, subtraction, and multiplication of *posit* numbers but do not provide the necessary rounding mode (round-to-nearest with tie-to-even) support. There exist works like [7, 34, 44], which compare their *posit* designs with IEEE-754-based designs to evaluate *posit* as a suitable replacement for IEEE-754. Along with this, our work provides a solution where *posit* and IEEE-754 FPU can co-exist on the same chip and enable users to choose either for their applications.

Works presented in References [19, 24, 31, 35, 44, 49] are stand-alone *posit* FPUs. To integrate them into a processor, they would need to implement all the instructions in an ISA. Such as, one would require implementing operations like F2I conversion and vice versa, which are used in type-casting a data-type. Our work, not only integrates the *posit* FPU in RISC-V core but also presents the compiler modifications that are required to execute applications on a *posit*-based RISC-V core. One more such stand-alone *posit* multiplier is presented in Reference [51] with an emphasis on power optimization.

Along with the parameterized design of the *posit* unit, our work provides the capability to support two *es* values within the same hardware unit, thereby catering to applications from a wide range of domains. Implementation discussed in Reference [37] also provides capability to define different *es* values for different applications but does not allow to switch between *es* values during application execution. Hence PERI is a runtime reconfigurable implementation, whereas Reference [37] does not allow to change *es* value at runtime.

Table 18 provides a comparison of PERI with some recent works. CLARINET [21] is an open source *posit* arithmetic implementation that supports only FMA operation. Moreover, to use the *posit* operations, CLARINET involves converting IEEE-754 format to *posit* format and does not provide compiler modifications. PERC [29], however, provides implementation of FMA, division, square root, and conversion operations but do not provide compiler modifications. Moreover, *posit* is a proposed replacement of IEEE-754 and promises to be better than IEEE-754. Any work on *posit* arithmetic should be able to characterize *posit* with respect to IEEE-754. We have designed our *posit* FPU similar to the HardFloat implementation to provide a comparison of *posit* with IEEE-754 in terms of the hardware area, application accuracy, and runtime. Such a comparison with IEEE-754 is missing in PERC.

¹Details in the table for Calligo Technologies and Vivid Sparks are based on the information present in their website and brochure. The actual implementation details are not available.

Table 19. Usage Scenario of 32-bit *posit* and IEEE-754 Representation for Applications

<i>Posit</i> Representation	IEEE-754 Representation
Applications requiring higher accuracy or range.	Applications having area constraints or requiring lesser execution time.

Several works on *posit* arithmetic like [5, 6, 25, 30, 32, 40] focus on designing application-specific hardware for deep learning and neural network, whereas in PERI, we focus on integrating *posit* FPU with a RISC-V core and enabling the associated framework for the evaluation of *posit*.

11 CONCLUSION

In this article, we propose a parameterized *posit*-based FPU design in BSV and its integration in a RISC-V compliant core. The article elaborates how the RISC-V “F” extension can be extended/modified to support *posit* arithmetic. The article further highlights the specific differences between *posit* and IEEE-754 in view of RISC-V and thereby concludes that *posit* simplifies designing of floating-point arithmetic significantly. The article also proposes a novel idea of supporting multiple *es* values within the same hardware unit with dynamic switching capabilities through minimal overheads. We also demonstrate the compiler modifications required to execute applications on a RISC-V core with *posit* support. Based on our analysis from the comparison of *posit* with IEEE-754, we have mentioned the usage scenario of 32-bit *posit* and IEEE-754 representation for applications in Table 19. We also present an alternate methodology of exploiting the *custom* space of RISC-V ISA to integrate the *posit* FPU as an accelerator with any RISC-V core supporting a RoCC like interface.

ACKNOWLEDGMENTS

The authors thank Dr. V. Krishna Nandivada for his help with compiler modifications for *posit* and Dr. John L. Gustafson for his help with *posit* arithmetic.

REFERENCES

- [1] P. H. Abbott, D. G. Brush, C. W. Clark, C. J. Crone, J. R. Ehrman, G. W. Ewart, C. A. Goodrich, M. Hack, J. S. Kapernick, B. J. Minchau, W. C. Shepard, R. M. Smith, R. Tallman, S. Walkowiak, A. Watanabe, and W. R. White. 1999. Architecture and software support in IBM S/390 parallel enterprise servers for IEEE floating-point arithmetic. *IBM J. Res. Dev.* 43, 5.6 (1999), 723–760. DOI: <https://doi.org/10.1147/rd.435.0723>
- [2] Krste Asanovi, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.
- [3] Bluespec. Open-source RISC-V CPUs. Retrieved September 24, 2019 from <https://github.com/bluespec/Piccolo>.
- [4] J. Canny. 1986. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 8, 6 (Nov. 1986), 679–698. DOI: <https://doi.org/10.1109/TPAMI.1986.4767851>
- [5] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. 2019. Deep positron: A deep neural network using the posit number system. In *Proceedings of the 2019 Design, Automation Test in Europe Conference Exhibition (DATE'19)*. 1421–1426. DOI: <https://doi.org/10.23919/DATE.2019.8715262>
- [6] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. 2019. Performance-efficiency trade-off of low-precision numerical formats in deep neural networks. In *Proceedings of the Conference for Next Generation Arithmetic 2019 (CoNGA'19)*. Association for Computing Machinery, New York, NY, Article 3, 9 pages. DOI: <https://doi.org/10.1145/3316279.3316282>
- [7] Rohit Chaurasiya, John Gustafson, Rahul Shrestha, Jonathan Neudorfer, Sangeeth Nambiar, Kaustav Niyogi, Farhad Merchant, and Rainer Leupers. 2018. Parameterized posit arithmetic hardware generator. In *IEEE 36th International Conference on Computer Design (ICCD'18)*. 334–341. DOI: <https://doi.org/10.1109/ICCD.2018.00057>

- [8] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid-State Circ.* 9, 5 (Oct. 1974), 256–268. DOI: <https://doi.org/10.1109/JSSC.1974.1050511>
- [9] Western Digital. RISC-V SweRV Core. Retrieved September 24, 2019 from <https://blog.westerndigital.com/risc-v-swerv-core-open-source/>.
- [10] Gordon E. Moore. 2006. Cramming more components onto integrated circuits. *IEEE Solid-State Circ. Newslett.* 11, 3 (2006), 33–35. DOI: <https://doi.org/10.1109/N-SSC.2006.4785860>
- [11] W. Eric Wong, Xuelin Li, Philip A. Laplante, and Mike Siok. 2017. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. *J. Syst. Softw.* 133 (2017), 68–94. DOI: <https://doi.org/10.1016/j.jss.2017.06.069>
- [12] Xin Fang and Miriam Leeser. 2016. Open-source variable-precision floating-point library for major commercial FP-GAs. *ACM Trans. Reconfig. Technol. Syst.* 9, 3, Article 20 (Jul. 2016), 17 pages. DOI: <https://doi.org/10.1145/2851507>
- [13] R. A. Fisher. 1936. The use of multiple measurements in taxonomic problems. *Ann. Eugen.* 7, 7 (1936), 179–188. DOI: <https://doi.org/10.1111/j.1469-1809.1936.tb02137.x>
- [14] Neel Gala, Arjun Menon, Rahul Bodduna, G. S. Madhusudan, and V. Kamakoti. 2016. SHAKTI processors: An open-source hardware initiative. In *Proceedings of the 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID'16)*. IEEE Computer Society, Los Alamitos, CA, 7–8. DOI: <https://doi.org/10.1109/VLSID.2016.130>
- [15] Gustafson and Yonemoto. 2017. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov. Int. J.* 4, 2 (Jun. 2017), 71–86. DOI: <https://doi.org/10.14529/jsfi170206>
- [16] Hamelg. Confidence interval. Retrieved September 24, 2019 from <http://hamelg.blogspot.com/2015/11/python-for-data-analysis-part-23-point.html>.
- [17] John Hauser. HardFloat. Retrieved September 24, 2019 from <http://www.jhauser.us/arithmetic/HardFloat-1/doc/HardFloat-Verilog.html>.
- [18] Paul Horton and Kenta Nakai. 1996. A probabilistic classification system for predicting the cellular localization sites of proteins. In *Proceedings of the 4th International Conference on Intelligent Systems for Molecular Biology*. AAAI Press, 109–115. <https://dl.acm.org/doi/10.5555/645631.662879>
- [19] Junjie Hou, Yongxin Zhu, Sen Du, and Shijin Song. 2018. Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on FPGA. *J. Sign. Process. Syst.* 91, 10 (2018), 1137–1148. DOI: <https://doi.org/10.1007/s11265-018-1420-5>
- [20] Bluespec Inc. 2006. Bluespec System Verilog. Retrieved June 7, 2006 from <https://bluespec.com/>.
- [21] Riya Jain, Niraj Sharma, Farhad Merchant, Sachin Patkar, and Rainer Leupers. 2020. CLARINET: A RISC-V based framework for posit arithmetic empiricism. *arxiv:cs.AR/2006.00364*. Retrieved from <https://arxiv.org/abs/2006.00364>.
- [22] M. K. Jaiswal and H. K. So. 2018. Architecture generator for Type-3 Unum posit adder/subtractor. In *Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS'18)*. 1–5. DOI: <https://doi.org/10.1109/ISCAS.2018.8351142>
- [23] M. K. Jaiswal and H. K. So. 2018. Universal number posit arithmetic generator on FPGA. In *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition (DATE'18)*. 1159–1162. DOI: <https://doi.org/10.23919/DATE.2018.8342187>
- [24] M. K. Jaiswal and H. K. So. 2019. PACoGen: A hardware posit arithmetic core generator. *IEEE Access* 7 (2019), 74586–74601. DOI: <https://doi.org/10.1109/ACCESS.2019.2920936>
- [25] H. F. Langroudi, V. Karia, J. L. Gustafson, and D. Kudithipudi. 2020. Adaptive posit: Parameter aware numerical format for deep learning inference on the edge. In *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'20)*. 3123–3131. DOI: <https://doi.org/10.1109/CVPRW50498.2020.00371>
- [26] Zoltán Lehóczky, András Retzler, Richárd Tóth, Álmos Szabó, Benedek Farkas, and Krisztián Somogyi. 2018. High-level .NET software implementations of Unum Type I and posit with simultaneous FPGA implementation using hastlayer. In *Proceedings of the Conference for Next Generation Arithmetic (CoNGA'18)*. ACM, New York, NY. DOI: <https://doi.org/10.1145/3190339.3190343>
- [27] Cerlane Leong. SoftPosit Library. Retrieved September 24, 2019 from <https://gitlab.com/cerlane/SoftPosit>.
- [28] lowRISC. The lowRISC Project. Retrieved September 24, 2019 from <https://www.lowrisc.org/>.
- [29] Arunkumar M V, Ganesh Bhairathi, and Harshal Hayatnagarkar. 2020. PERC: Posit enhanced rocket chip. In *Proceedings of the 4th Workshop on Computer Architecture Research with RISC-V (CARRV'20)*.
- [30] Raúl Murillo Montero, Alberto A. Del Barrio, and Guillermo Botella. 2019. Template-based posit multiplication for training and inferring in neural networks. *arxiv:cs.CV/1907.04091*. Retrieved from <https://arxiv.org/abs/1907.04091>.
- [31] R. Murillo, A. A. Del Barrio, and G. Botella. 2020. Customized posit adders and multipliers using the FloPoCo core generator. In *Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS'20)*. 1–5. DOI: <https://doi.org/10.1109/ISCAS45731.2020.9180771>

- [32] N. Neves, P. Tomás, and N. Roma. 2020. Reconfigurable stream-based tensor unit with variable-precision posit arithmetic. In *Proceedings of the 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP'20)*. 149–156. DOI : <https://doi.org/10.1109/ASAP49362.2020.00033>
- [33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830. <https://dl.acm.org/doi/10.5555/1953048.2078195>
- [34] A. Podobas and S. Matsuoka. 2018. Hardware implementation of POSITs and their application in FPGAs. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*. 138–145. DOI : <https://doi.org/10.1109/IPDPSW.2018.00029>
- [35] A. Raveendran, S. Jean, J. Mervin, D. Vivian, and D. Selvakumar. 2020. A novel parametrized fused division and square-root POSIT arithmetic architecture. In *Proceedings of the 2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID'20)*. 207–212. DOI : <https://doi.org/10.1109/VLSID49098.2020.00053>
- [36] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini. 2015. PULP: A parallel ultra low power platform for next generation IoT applications. In *Proceedings of the 2015 IEEE Hot Chips 27 Symposium (HCS'15)*. 1–39. DOI : <https://doi.org/10.1109/HOTCHIPS.2015.7477325>
- [37] S. Sarkar, P. M. Velayuthan, and M. D. Gomony. 2019. A reconfigurable architecture for posit arithmetic. In *Proceedings of the 2019 22nd Euromicro Conference on Digital System Design (DSD'19)*. 82–87. DOI : <https://doi.org/10.1109/DSD.2019.00022>
- [38] H. P. Sharangpani and M. L. Barton. 2004. Statistical Analysis of Floating Point Flaw in the Pentium Processor. Retrieved from https://web.archive.org/web/20160406055056/http://download.intel.com/support/processors/pentium/sb/FDIV_Floating_Point_Flaw_Pentium_Processor.pdf#.
- [39] IEEE Computer Society. 2008. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug. 2008), 1–70. DOI : <https://doi.org/10.1109/IEEESTD.2008.4610935>
- [40] L. Sommer, L. Weber, M. Kumm, and A. Koch. 2020. Comparison of arithmetic number formats for inference in sum-product networks on FPGAs. In *Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'20)*. 75–83. DOI : <https://doi.org/10.1109/FCCM48280.2020.00020>
- [41] Nick Street, William H. Wolberg, and O. L. Mangasarian. 1993. Nuclear feature extraction for breast tumor diagnosis. *Proc. Soc. Photo-Opt. Inst. Eng.* 1905 (1993), 861–870. DOI : <https://doi.org/10.1117/12.148698>
- [42] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2018. A transprecision floating-point platform for ultra-low power computing. In *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition (DATE'18)*. 1051–1056. DOI : <https://doi.org/10.23919/DATE.2018.8342167>
- [43] Calligo Technologies. Posit Numeric Unit (PNU) implementation by Calligo technologies. Retrieved September 24, 2019 from <https://posithub.org/conga/2018/docs/9-Calligo-Technologies.pdf>.
- [44] Yohann Uguen, Luc Forget, and Florent de Dinechin. 2019. Evaluating the hardware cost of the posit number system. In *Proceedings of the 29th International Conference on Field-Programmable Logic and Applications (FPL'19)*. 1–8.
- [45] VividSparks. Posit implementation by VividSparks. Retrieved September 24, 2019 from <http://vivid-sparks.com/>.
- [46] G. K. Wallace. 1992. The JPEG still picture compression standard. *IEEE Trans. Consum. Electr.* 38, 1 (Feb. 1992), xviii–xxxiv. DOI : <https://doi.org/10.1109/30.125072>
- [47] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley.
- [48] Nathan Whitehead and Alex Fit-florea. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Retrieved September 01, 2019 from http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Floating_Point_on_NVIDIA_GPU_White_Paper.pdf.
- [49] Feibao Xiao, Feng Liang, Bin Wu, Junzhe Liang, Shuting Cheng, and Guohe Zhang. 2020. Posit arithmetic hardware implementations with the minimum cost divider and SquareRoot. *Electronics* 9, 10 (Oct. 2020), 1622. DOI : <https://doi.org/10.3390/electronics9101622>
- [50] C. Yarp. An Introduction to the Rocket Custom Coprocessor Interface. Retrieved September 24, 2019 from <http://www-inst.eecs.berkeley.edu/~cs250/sp16/disc/Disc02.pdf>.
- [51] H. Zhang and S. Ko. 2020. Design of power efficient posit multiplier. *IEEE Trans. Circ. Syst. II: Express Briefs* 67, 5 (2020), 861–865. DOI : <https://doi.org/10.1109/TCSII.2020.2980531>

Received June 2020; revised December 2020; accepted December 2020