PERI: A Posit Enabled RISC–V Core

4 authors, including:

Neel Gala
Indian Institute of Technology Madras
20 PUBLICATIONS 209 CITATIONS

SEE PROFILE

Chester Rebeiro
Indian Institute of Technology Madras
90 PUBLICATIONS 998 CITATIONS

SEE PROFILE

Kamakoti Veezhinathan
Indian Institute of Technology Madras
190 PUBLICATIONS 1,391 CITATIONS

SEE PROFILE

PERI: A Posit Enabled RISC-V Core

Sugandha Tiwari$^{\top}$, Neel Gala$^{\perp}$, Chester Rebeiro$^{\top}$ and V. Kamakoti$^{\top}$

$^{\top}$ Indian Institute of Technology Madras

$^{\perp}$ InCore Semiconductors Pvt. Ltd.

$^{\top}$ {cs17s001, chester, kama}@cse.iitm.ac.in, $^{\perp}$ neelgala@incoresemi.com

Abstract—Owing to the failure of Dennard's scaling the last 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 754-2008. Posit promises more accuracy, higher dynamic range and fewer unused states along with simpler hardware designs as compared to IEEE 754-2008. RISC-V, on the other hand, provides a commercial-grade open-source ISA. It is not only elegant and simple but also highly extensible and customizable, thereby facilitating novel micro-architectural research and exploration. In this paper, we bring these two technologies together and propose the first Posit Enabled RISC-V core. The paper provides insights on how the current 'F' extension and the custom op-code space of RISC-V can be leveraged/modified to support Posit arithmetic. We also present implementation details of a parameterized and feature-complete Posit FPU which is integrated with the RISC-V compliant SHAKTI C-class core either as an execution unit or as an accelerator. To fully leverage the potential of Posit, we further enhance our Posit FPU, with minimal overheads, to support two different exponent sizes (with posit-size being 32-bits). This allows applications to switch from high-accuracy computation mode to a mode with higher dynamic-range at run-time. In the absence of viable software tool-chain to enable porting of applications in the Posit domain, we present a workaround on how certain applications can be modified minimally to exploit the existing RISC-V tool-chain. We also provide examples of applications which can perform better with Posit as compared to IEEE 754-2008. The proposed Posit FPU consumes 3507 slice LUTs and 1294 slice registers on an Artix-7-100T Xilinx FPGA while capable of operating at 100 MHz.

Index Terms—Posit, IEEE-754, RISC-V, floating-point, processor

I. INTRODUCTION

For years, computer architects have cruised Moore's law [1] and Dennard's scaling [2] to leverage increased resources to improve the quality of service of general-purpose processors. However, recent explosion in domain-specific applications and the need for customized hardware has challenged architects to re-visit basic computer architecture principles for hidden opportunities. One such opportunity lies in computing with floating-point numbers.

The IEEE-754 standard [3] has been considered the de facto 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 quite a few acknowledged 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 significant amount of area and energy of the chip [4], [5]. Moreover, the complexity and inconsistencies within the IEEE-754 standard have led to several implementation errors of FPU (Floating Point Units) [6], [7]. This has made compliance with the standard a major effort. IEEE-754 also suffers from lack of guaranteed re-producibility for some operations and across platforms as it recommends that compilers should ensure the same [3]. The support for overflow and underflow further leads to loss of accuracy. The standard, even today, allows multiple and redundant representations of NaNs (Not-a-Number) making software portability across implementations a major challenge. While subnormals are rare-to-occur numbers, the corresponding hardware to handle them adds significant overheads to the entire design. Though today, the standard elaborates on 16-bit, 32-bit, 64-bit, 128-bit, and 256-bit representations, each bit-width can support only fixed precision and dynamic range. Despite these challenges, the lack of a suitable replacement has forced architects to continue with the IEEE-754 standard for floating-point numbers.

Quite recently, *posits* [8] have been proposed as an alternate and efficient way of representing floating-point numbers. Unlike IEEE-754, *posits* enable reproducible results across platforms. By not supporting overflow and underflow, *posits* preserve the remaining information by rounding down or up. *Posit* based arithmetic is significantly more simplified due to the absence of subnormals and NaNs, thereby also simplifying handling of exceptions to a large extent. 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 (*p*s).

Several works in literature [9]–[15] have proposed techniques and methodologies to implement/generate hardware for *posit* based arithmetic operations such as: adders, multipliers and dividers. Though these operations form an integral part of many *posit* based FPUs (Floating Point Units), they are not sufficient to support the compute requirements of a commercial-grade ISA (Instruction Set Architecture). Such a support would require functional units which can perform conversion of a number from the *posit* domain to integer domain and vice-versa, compare *posit* numbers, square-root operations, classification operations and much more. To that extent, in this paper we propose designs of relevant functional units required to build a complete *posit* based FPU which can support the floating-point compute extension of the RISC-V [16] ISA. Furthermore, the paper also describes

methodologies of integrating the proposed *posit* FPU with an open-source general-purpose RISC-V core either as a tightly-coupled execution unit within the core pipeline or as co-processor.

One of the major attributes of *posit* is its ability to facilitate different dynamic range (and accuracy) for the same *ps* value by simply manipulating the value of *es*. Nonetheless, all of the previous works on *posit* have focused on designing/generating hardware compute blocks for a single pair-value of *es* and *ps*. In this paper, we extend the functionality of our *posit* ALU to support two *es* values (*es* = 2 and 3), with minimal overheads. We propose to extend the control and status register to hold the current value of *es* for all operations. Thereby, enabling the user to choose either higher accuracy or higher dynamic range for the same *ps* size.

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

1) The paper provides insights on how the RISC-V ISA can be leveraged, modified and customized to support *posit* based compute.
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.
3) The proposed *posit* FPU has been integrated with an open-source RISC-V core as both: a tightly-coupled execution unit and a co-processor. To the best of the authors' knowledge, this is the first work to provide a complete processor with *posit* support.
4) The *posit* FPU has been further enhanced, with minimal overheads, to support two *es* values in same HW, thereby enabling dynamic switching between higher accuracy or higher dynamic range at run-time.
5) The paper also presents analysis of various software applications running on the core, which provide similar or better performance in terms of quality as compared to IEEE-754.

The rest of the paper is organized as follows. Section-II provides a brief introduction and background to the *posit* format and the RISC-V ISA. Section-III discusses the details of extending the RISC-V ISA to accommodate *posit* arithmetic operations. The entire *posit* floating-point unit is discussed in detail in Section-IV. Integration of the proposed *posit* unit with a RISC-V core is highlighted in Section-V. The software modifications required to port *posit* based applications on a RISC-V are covered in Section-VI. Section-VII presents the results and insights obtained from running a few interesting applications on the *posit* enabled RISC-V core. The hardware results are reviewed in Section-VIII followed by brief literature survey in Section-IX. Section-X finally concludes the paper.

II. Background

A. The Posit Format

The *posit* representation is defined using two parameters: the *posit size (ps)* and *exponent size (es)*. A formal representation of a *posit* number is shown in Equation-1.

$$\overbrace{s}^{Sign} \; \overbrace{r \; r \; r \; ... \; r \; \bar{r}}^{Regime} \; \overbrace{e_1 \; e_2 \; e_3 \; ... \; e_{es}}^{Exponent, if \; any} \; \overbrace{f_1 \; f_2 \; f_3 \; f_4 \; ...}^{Fraction, if \; any} \qquad (1)$$
$$\underbrace{\phantom{s \; r \; r \; r \; ... \; r \; \bar{r} \; e_1 \; e_2 \; e_3 \; ... \; e_{es} \; f_1 \; f_2 \; f_3 \; f_4 \; ...}}_{posit \; size}$$

A *posit* number utilizes the 2's complement notation to represent a negative number. The first bit (from the left) of the *posit* number is the sign bit (denoted by *s*). The number of identical bits following the sign bit, are called regime bits, and they determine the value of the variable *k*. If there are *rc* identical bits in a number, then value of *k* is determined by Equation-2. The *es* number of bits after the regime bits (denoted by *e*) help determine the *posit* exponent. The *posit* exponent value (denoted by *exp*) is a combination of regime and *e* bits and is derived using Equation-3. The remaining bits trailing the *e* bits, forms the fraction of the number. The implicit (hidden) bit is always 1 in case of *posit* fraction (denoted by *f*). Thus, the value *x* of a *posit* number *P* is determined by Equation-4.

As can be seen from Equation-4 the only two exceptions in *posit* representations are 0 and *NaR* (Not-a-Real) numbers. This makes exception handling very simple in *posit* as compared to IEEE-754.

$$k = \begin{cases} rc - 1 & \text{if regime starts with 1} \\ -rc & \text{otherwise} \end{cases} \qquad (2)$$

$$exp = (k \ll es) + e \qquad (3)$$

$$x = \begin{cases} 0 & P = 000...000 \\ NaR & P = 100...000 \\ (-1)^s \times 2^{exp} \times 1.f & \text{otherwise} \end{cases} \qquad (4)$$

Another significant difference between *posit* and IEEE-754 representation is that *posit* uses run-length encoding 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.

B. The RISC-V ISA

RISC-V (pronounced "risk-five") is a fairly new ISA, which has received tremendous momentum and support in the recent years by both, academia and industry. RISC-V is a completely open ISA suitable for real HW implementations and not just simulation or binary translation. The ISA has been well designed with a *small*, but usable, base Integer ISA (RV32I) and does not mandate any particular micro-architecture style. The attribute of optional standard extensions and the ability to add custom ISA extensions has been the most attractive features of RISC-V.

RISC-V today is supported by a strong and vibrant software ecosystem which includes support for: gcc, binutils, llvm, gdb, open-ocd, linux kernel, sel4 and much more. There have also been significant efforts world-wide to build open-source processors around RISC-V. Some of the prominent works include: SHAKTI [17], Rocket-Chip [18], lowRisc [19] and

Ariane [20]. Commercial entities such as Western-Digital and Bluespec Inc. have also contributed their core implementations (SweRV [21] and Piccolo/Flute [22] respectively) to the open-source community.

With regards to floating-point compute, the RISC-V ISA includes two standard extensions: 'F' for single-precision floating-point and 'D' for double-precision floating-point compute. Each of these extensions comprises of instructions compliant with the IEEE-754 arithmetic 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 will, while still being compliant with other standard extensions of the specification. In the next section, we leverage this opportunity and describe how the current 'F' extension can be modified to support *posit* arithmetic.

Within the 32-bit instruction format, the ISA has also locked down on two major-opcodes as *custom*, meaning that these opcodes can be used to define user-specific custom instructions with a guarantee that no future standard extensions shall use this opcode space. Subsequent sections of this paper exploit this opportunity to present how this custom opcode space can be leveraged to build a larger *posit* based co-processor compliant with any RISC-V device.

III. THE RISC-V POSIT EXTENSIONS

This section describes how the RISC-V ISA can be leveraged and modified to include *posit* based arithmetic operations. We propose two separate approaches to ISA extensions. The first approach is to leverage the 'F' extension of the ISA itself with minimal modifications to support *posit* based arithmetic. This approach requires none/minimal changes in the software tool-chain and thus enabling quick bring up of the design.

The second approach relies on utilizing the *custom-0/1/2/3* major-opcodes space of the 32-bit instruction format. This opcode space has been frozen by the ISA to be used for integrating custom instruction sets and will be future compatible. This approach allows us to expand the *posit* FPU capabilities beyond the operations specified in the 'F' extension. Furthermore, this method enables integrating a *posit* based FPU as a co-processor to any RISC-V core which forwards all *custom* opcodes externally using a standard co-processor interface. Unlike, the previous approach, this approach does not require any modifications to occur within the core-pipeline and also facilitates the co-existence of the IEEE-754 and *posit* compute units on the same chip. However, utilizing the *custom-0/1/2/3* opcodes requires changes to the software tool-chain.

Each of the above approaches is discussed in detail in the following subsections.

A. Leveraging the 'F' extension

Before proceeding further, the reader is recommended to be cognizant with the 'F' extension of the RISC-V ISA. A gist of instructions comprising the 'F' extension are captured in Table-I.

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.WU.S, FCVT.S.W, FCVT.S.WU	Conversion ops
FMV.X.W, FMV.W.X	Transfer ops
FCLASS.S	Classification op

TABLE I: List of instructions comprising the F extension of RISC-V.

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

We propose to 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 of the control and status register is shown in Figure-1. Since *posit* support only one rounding mode (round-to-nearest with tie-to-even), the *rounding mode (rm)* field is not required and thus tied to zeros. Similarly, there is no requirement of flags for invalid, inexact, overflow and underflow. The *posit* exception of NaR (Not-a-Real) is silent and thus does not get captured in the flags. The exception of divide-by-zero 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 to deduce the *posit* number. To maintain compatibility with the 'F' extension, the *ps* value is set to 32-bits, and we expect all practical implementations of a *posit* FPU to support *es* values which 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 paper 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. Implementations which 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 spec mandates that any floating-point operation resulting in a NaN (Not-a-Number) should output a canonical NaN (i.e. `0x7fc00000`). However, in *posit*, NaR has a single representation which maps to the most negatives 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.

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 mode, certain applications, like JPEG compression, are able to provide much better results as 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 spec.

Since *posit* does not have subnormals, different types of NaN values, infinities or different kind of zeros, the classify operation will only capture if the operand is a zero, NaR, negative or positive, leaving the other bits to be zeros always.

B. Leveraging the 'Custom' opcode space

While the F extension includes a nice subset of standard floating-point instructions which 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. Moreover, as the complexity continues to grow, it would seem beneficial to integrate such an FPU as an accelerator rather than a tightly coupled execution unit within the core pipeline. This reduces the risk of modifying the pipeline of the core, and more importantly, enables a stand-alone *posit* FPU design which can be integrated to any other RISC-V core which adheres to a standard co-processor interface.

In this paper, we choose to adopt the RoCC [23] (Rocket Custom Co-processor) interface for our *posit* FPU co-processor. All RoCC compliant accelerators follow a standard instruction format, as shown in Table-II. 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. We now discuss how the 'F' extension can be mapped to the standard instruction format of Table-II using the custom opcode spaces: *custom-0/1/2/3*.

Table-III shows the four instruction formats used by the default 'F' extension. 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 introduces more bits to encode information. Table-IV shows equivalent mappings of I, S, R4 and R-type instruction formats to a RoCC based instruction format which can be leveraged by the *posit* based FPU. We have not provided a separate *xs3* field since for all floating operations the rs3 value is always read from floating register-file. A quick observation reveals that the custom space for *posit*

can accommodate up to 512 R-type and 16 R4-type unique instructions. Additionally, from the I-type format, one can either have up to 8 instructions which 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 8 unique store-like instructions employing the S-type format. Later sections of this paper, will provide a brief overview of how these instructions are fetched by the core processor and offloaded to the accelerator for computing.

At this point, the authors would like to highlight the fact that none of the proposed approaches have any significant impact on the arithmetic compute units of the *posit* based FPU. Both of the above-discussed approaches only impact the integration scheme of an FPU with a core and have a close-to-none impact on the decoder. All the arguments regarding NaR, flags, es-mode, etc made in the previous sub-section hold in this solution as well.

IV. POSIT FPU

This section describes the various components of a *posit* FPU for a RISC-V processor. We have implemented our design using Bluespec System Verilog (BSV). Our *posit* FPU is parameterized to generate hardware for any combination of *ps* and *es* value. Figure-2 shows the various components present in our implementation. Our *posit* FPU has the following BSV interface definition. The input parameters of the interface refer to the fields of Table-III and IV.

```
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
```

A. Common Posit Decoder

This is the first block of our *posit* unit and 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.

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 which has 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 6-7). To extract the exponent and fraction, we count the number of regime bits (lines 8-11). If the regime starts with a 0, then the exponent is treated as negative, else positive. Please note, that the sign of the exponent in IEEE-754 is derived through a bias. Lines 12-18 implement Equations-2 and 3 to capture the final exponent value. Lastly, fraction bits are obtained by shifting out the exponent bits (lines 19-20). Unlike IEEE-754, *posit* do not have subnormal numbers, eliminating

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

TABLE II: RoCC interface standard instruction format.

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

TABLE III: RISC-V 32-bit instruction types used in RV32F

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

TABLE IV: Equivalent mappings of Table-III instruction formats to RoCC instruction format.
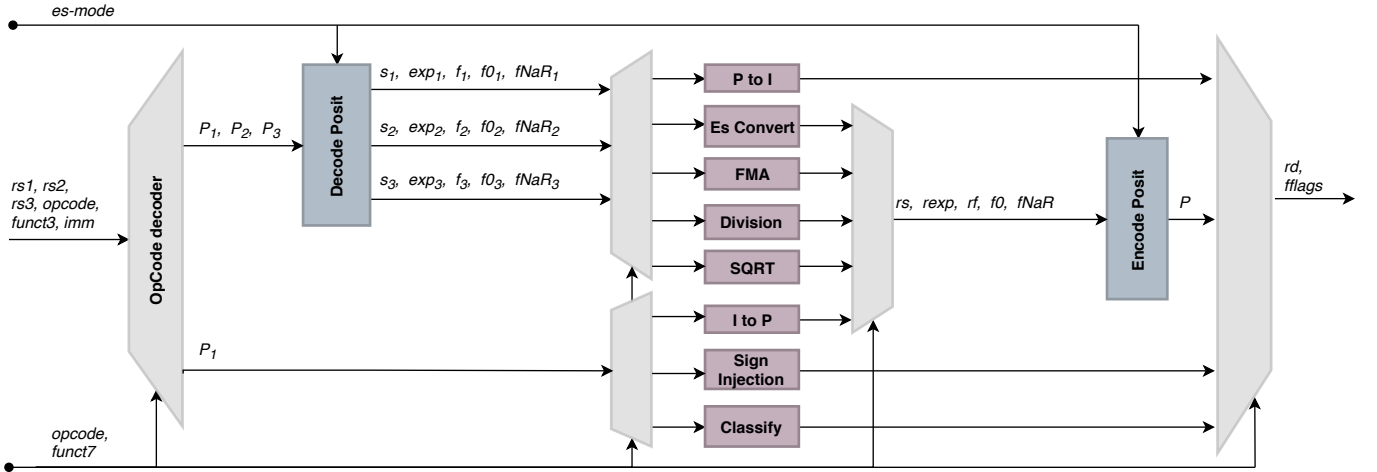
Fig. 2: Block diagram showing the different components of *Posit* unit.

the need to handle subnormal exponent and fraction separately, thereby simplifying the *posit* arithmetic.

B. Common Posit Encoder

This is the final block of the design and 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 3-4). A negative exponent is represented with the regime bits beginning with a 1 (0 otherwise) (lines 5-12). The sticky bit (*sb*) is calculated as the Logical-OR of all the shifted-bits. Once the regime bits have been deduced, the exponent and fraction bits are appended appropriately (lines 13-17). Following this, the posit is rounded to the nearest even (with tie-to-even) in lines 18-28. If either of the input flags, 0 or NaR, are set, then the final result is modified accordingly (lines 29-32), else the rounded result is forwarded to the output.

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 nor underflows. If the encoded number is *maxpos* (integer value of $2^{ps-1}-1$), it is not rounded up irrespective of round bit (*rb*). Similarly, if the encoded number is *0*, it is rounded up irrespective of the round bit.

The following subsections will define the different modules and the instruction they implement in the proposed *posit* unit.

C. Fused Multiply-Add (FMA)

The 'F' extension of RISC-V ISA specifies four different fused ops, FMADD.S, FMSUB.S, FNMSUB.S and FN-MADD.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 resource to be re-used across operations.

Our FMA block checks whether either of the inputs are 0 or NaR. The corresponding circuitry in IEEE-754 would require checking for 5 exceptional cases per operand. Fused

Algorithm 1 Algorithm for *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* including hidden bit, *f0:* set if *P* is 0, *fNaR:* set if *P* is NaR

1: **Derived Parameters:** *fs:* maximum fraction size of posit format given by $(ps - es - 3)$
2: **Initialize:** $k=0$
3: $f0 \leftarrow\sim |P$ ▷ set if 0
4: $fNaR \leftarrow P[ps - 1] \& (\sim |(P[ps - 2 : 0])$ ▷ set if NaR
5: $s \leftarrow P[ps - 1]$
6: **if** $(s = 1)$ **then**
7:    $P \leftarrow\sim P + 1$ ▷ 2's complement of P
8: $t \leftarrow P[ps - 1 : 0]$
9: **if** $(P[ps - 2] = 1)$ **then**
10:    $t \leftarrow\sim t$ ▷ 1's complement of t
11: $rc \leftarrow countZerosMSB(t)$
12: **if** $(P[ps - 2] = 1)$ **then**
13:    $k \leftarrow rc - 1$
14: **else**
15:    $k \leftarrow (-ve)rc$
16: $P \leftarrow P \ll rc + 2$
17: $e \leftarrow P[ps - 1 : ps - es]$
18: $exp \leftarrow e + k \ll es$
19: $P \leftarrow P \ll es$
20: $f \leftarrow 1 \parallel P[ps - 1 : ps - fs]$ ▷ append hidden bit
21: **return** *s, exp, f, f0, fNaR*

Algorithm 2 Algorithm for *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*: maximum exponent size of ps bit posit, given by $(\log_2(ps) + es + 2)$
2: **Initialize:** *P = 0, c=2*
3: $e \leftarrow rexp[es - 1 : 0]$
4: $k \leftarrow rexp[fes - 1 : es]$
5: **if** $(k \geq 0)$ **then**
6:    $P \leftarrow\sim P$ ▷ regime start with 1 for positive exp
7:    $c = 3$
8: **else**
9:    $P[0] \leftarrow 1$
10: $k' \leftarrow abs(k)$
11: $P \leftarrow P \ll (ps + 1 - k`)$ ▷ shift P to finalize regime
12: $P[ps] \leftarrow 0$ ▷ add sign bit
13: $esft \leftarrow k' + c$
14: $ef \leftarrow e \parallel rf$
15: $sb \leftarrow sb \mid (|(ef \ll ps + 1 - esft))$ ▷ update sb
16: $ef \leftarrow ef \gg esft$ ▷ adjust e,f wrt regime
17: $P \leftarrow P \mid ef$ ▷ e and frac bits are added
18: $rb \leftarrow P[0], gb \leftarrow P[1]$
19: $rb \leftarrow rb \& gb \mid sb$ ▷ set rb as per round-to-nearest
20: $max \leftarrow\sim P[ps - 1]) \& (\& P[ps - 2 : 0]$
21: **if** $(max = 1)$ **then**   ▷ handles overflow
22:    $rb \leftarrow 0$ ▷ do not round if P is maxpos
23: **if** $(P = 0)$ **then**   ▷ handles underflow
24:    $P \leftarrow 1$ ▷ round up if P is zero
25: $rb \leftarrow rb \oplus s$
26: **if** $(rs = 1)$ **then**   ▷ complement negative number
27:    $P \leftarrow\sim P$
28: $P \leftarrow P + rb$
29: **if** $(f0 = 1)$ **then**   ▷ P is 0, if f0 is set
30:    $P \leftarrow 0$
31: **if** $(fNaR = 1)$ **then**   ▷ P is NaR if fNaR is set
32:    $P \leftarrow NaR$
33: **return** *P*

---

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* is burdened with only 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*.

D. Division (FDIV)

This block implements the FDIV.S instruction as Algorithm-4. A divide by zero exception in *posit* is captured by setting the DZ flag in *pcsr* (lines 3-4). IEEE-754, on the other hand, has to account for setting all 5 flags for FDIV.S. The sign and exponent of the result are quite simply calculated as per lines 7-8. This block uses an iterative non-restoring division algorithm for computing the division of the fractions (line 9). The number of cycles required for the operation is proportional to the size of the fractions. In each cycle, two iterations of non-restoring division are performed. The non-restoring algorithm returns the quotient and remainder, where the remainder part is used to calculate the sticky bit (line 10). The fractional part is checked for normalization before passing these values to the encoding module (line 11).

Similar to FMA, IEEE-754 divison has to normalize subnormal numbers before fraction division while *posit* incur no such hardware. Also, IEEE-754 checks the division exponent for overflow and underflow while *posit* do not.

E. Square-Root (FSQRT)

Algorithm-5 captures the implementation details of the square-root operation (FSQRT.S). If the input is an NaR or negative input the algorithm returns NaR (lines 1-2). The exponent of the result is obtained by dividing the input exponent by two (line 5). If the input exponent is odd, the fraction is left-shifted by one (lines 6-7). Similar to division, we have implemented an iterative non-restoring square root algorithm (line 8), where each cycle performs a single iteration. Square root operation also does not set any flags in *pcsr*, while IEEE-754 set the inexact flag. IEEE-754 algorithms are forced to normalize any subnormal inputs before finding the root while *posits* are not.

Algorithm 3 Algorithm for Fused Multiply-Add

Input: s_1, exp_1, f_1, $f0_1$, $fNaR_1$: components of operand$_1$ from decode stage, s_2, exp_2, f_2, $f0_2$, $fNaR_2$: components of operand$_2$, s_3, exp_3, f_3, $f0_3$, $fNaR_3$: components of operand$_3$, ng: if negate operation, op: if sub operation

Output: rs: sign of result, $rexp$ final exponent of fma, rf: final fraction of fma, sb: sticky bit, $f0$: set if result is 0, $fNaR$: set if result is NaR

1: **Derived Parameters:** ffs: final fraction size for fma given by $(2 \times (ps - es - 2))$
2: **Initialize:**
3: **if** $((fNaR_1 | fNaR_2 | fNaR_3) = 1)$ **then**
4:      $fNaR \leftarrow 1$ ▷ final result is NaR
5: **if** $(((f0_1 | f0_2) \& f0_3) = 1)$ **then**
6:      $f0 \leftarrow 1$ ▷ final result is 0
7: $s_3 \leftarrow s_3 \oplus op \oplus ng$ ▷ sub operation affects sign of op$_3$
8: $rs \leftarrow s_1 \oplus s_2 \oplus ng$
9: $rexp \leftarrow exp_1 + exp_2$
10: $rf \leftarrow f_1 \times f_2$
11: $rexp, rf \leftarrow chkMulOF(rf)$
12: **if** $((exp_3 > rexp) || (exp_3 = rexp \& f_3 > rf))$ **then**
13:      $swap(rs, s_3), swap(rexp, exp_3), swap(rf, f_3)$
14: $ediff \leftarrow rexp - exp_3$
15: $sb \leftarrow |(f_3 \ll (ffs - ediff))$ ▷ shifted bits are ORed
16: $f_3 \leftarrow f_3 \gg ediff$
17: **if** $(rs = s_3)$ **then**
18:      $rf \leftarrow rf + f_3$
19:      $rf, rexp \leftarrow chkAddOF(rf)$
20: **else**
21:      $rf \leftarrow rf - f_3$
22:      $rf, rexp \leftarrow normalize(rf)$
23: **return** $rs, rexp, rf, sb, f0, fNaR$

Algorithm 4 Algorithm for Division

Input: s_1, exp_1, f_1, $f0_1$, $fNaR_1$: components of operand$_1$ from decode stage, s_2, exp_2, f_2, $f0_2$, $fNaR_2$: components of operand$_2$

Output: rs: sign of result, $rexp$ final exponent of division, rf: final fraction of division, sb: sticky bit, $f0$: set if result is 0, $fNaR$: set if result is NaR, ex: exception flags **Initialize:**

1: **if** $((fNaR_1 | fNaR_2) = 1)$ **then**
2:      $fNaR \leftarrow 1$
3: **if** $(f0_2 = 0)$ **then**
4:      $ex[3] \leftarrow 1$ ▷ update DZ flag in $pcsr$
5: **if** $(f0_1 = 1)$ **then**
6:      $f0 \leftarrow 1$
7: $rs \leftarrow s_1 \oplus s_2$
8: $rexp \leftarrow exp_1 - exp_2$
9: $rf, rem \leftarrow nonRstrDiv(f_1, f_2)$ ▷ multi-cycle algo
10: $sb \leftarrow |rem$
11: $rf, rexp \leftarrow normalize(rf)$
12: **return** $rs, rexp, rf, sb, f0, fNaR, ex$

Algorithm 5 Algorithm for Square-Root

Input: s_1, exp_1, f_1, $f0_1$, $fNaR_1$: components of operand$_1$ from decode stage

Output: rs: sign of result, $rexp$ final exponent of sqrt, rf: final fraction of sqrt, sb: sticky bit, $f0$: set if result is 0, $fNaR$: set if result is NaR

1: **if** $((fNaR_1 | s_1) = 1)$ **then**   ▷ return NaR for -ve input
2:      $fNaR \leftarrow 1$
3: **if** $(f0_1 = 1)$ **then**
4:      $f0 \leftarrow 1$
5: $rexp \leftarrow exp_1 \gg 1$
6: **if** $(exp_1[0] == 1)$ **then**
7:      $f_1 \leftarrow f_1 \ll 1$ ▷ shift left for odd exponent
8: $rf, rem \leftarrow nonRstrSqrt(f_1)$ ▷ multi-cycle algo
9: $sb \leftarrow |rem$
10: **return** $0, rexp, rf, sb, f0, fNaR$

Algorithm 6 Algorithm for Integer to Posit Conversion

Input: I: integer of ps bits, u: is set if I is unsigned

Output: rs: sign of result, $rexp$: final exponent of result, rf: final fraction of result

1: $rs \leftarrow I[ps - 1]$
2: $rs \leftarrow rs \& \sim u$ ▷ do not consider sign if unsigned int
3: **if** $(rs = 1)$ **then**
4:      $I \leftarrow \sim I + 1$ ▷ 2's complement
5: $z \leftarrow countZeroMSB(I)$
6: $I \leftarrow I \ll z$
7: $rexp \leftarrow ps - 1 - z$
8: $rf \leftarrow I[ps - 2 : 0]$
9: **return** $rs, rexp, rf$

F. Integer to Posit Conversion

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-6. 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* (line 5-8). Integer to floating-point conversion operation for *posit* do not set any exceptional flag while IEEE-754 conversion involves setting of the inexact flag. In addition to the above, IEEE-754 also requires check for special values of n bit signed integers (0 and 2^{n-1}).

G. Posit to Integer Conversion

The 'F' extension provides FCVT.W.S/FCVT.WU.S instructions to convert *posit* to signed/unsigned integers respectively. These instructions are implemented using Algorithm-7. The u input bit indicates if the result should be unsigned. The extended *posit* fraction is left-shifted by the exponent (lines 2-3). The final integer is decided based on the u and exp inputs (lines 4-13). For FCVT.W.S and FCVT.WU.S instructions,

Algorithm 7 Algorithm for Posit to Integer Conversion

Input: s_1, exp_1, f_1: components of operand$_1$ from decode stage, u: is set if result is unsigned, rm: rounding mode

Output: I: integer value corresponding to posit

1: **Derived Parameter:** fs: maximum fraction size of posit format given by $(ps - es - 3)$
2: $f_1 \leftarrow 0_1 0_2 ... 0_{ps} \parallel f_1$ ▷ zero extend f_1 to ps+fs bits
3: $f_1 \leftarrow f_1 \ll exp_1$
4: **if** $(u = 0)$ **then**
5:     **if** $(exp_1 < ps - 1)$ **then**
6:         $I \leftarrow f_1[ps + fs - 1 : fs]$
7:     **else**
8:         $I \leftarrow 2^{ps-1} - 1$
9: **else**
10:     **if** $(exp_1 < ps)$ **then**
11:         $I \leftarrow f_1[ps + fs - 1 : fs]$
12:     **else**
13:         $I \leftarrow 2^{ps} - 1$
14: $rb \leftarrow f_1[fs - 1]$
15: **if** $(rm = 1)$ **then**   ▷ check for round-to-zero
16:     $rb = 0$
17: $I \leftarrow round(I, s_1, rb)$
18: **return** I

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

TABLE V: Instruction format for switching operands across *es* values.

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 case of *posit* number, 2's complement is taken to negate a number or to calculate the absolute value of a negative number. Both *posit* and IEEE-754 do not set any exception flags for this operation.

J. Posit Classification

FCLASS.S instruction has been defined to determine the category of a floating-point number. It is much required for IEEE-754 but not so much with *posit*. IEEE-754 have different categories which are ± 0, $\pm$infinity, $\pm$subnormals, $\pm$normals, sNaN and qNaN. Hence IEEE-754 need to check for all those values. *Posit* only need 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.

K. Dynamic Switching

Several scientific applications like weather forecasting, automotive design and safety etc. demand high precision floating-point calculations. On the other hand, applications in the deep learning domain require large dynamic ranges. One would require two separate designs to cater to both the demands. In regards to this, we propose a single *posit* FPU which can fulfil 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* unit can, thus, switch across various *es* values at run-time by manipulating the *es-mode* field in the *pcsr* register. We call this capability as *Dynamic Switching*.

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: *FCVT.ES* formatted as shown in Table-V. 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 with. 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. One should note that this instruction does not use the *es-mode* present in the *pcsr* register.

To support two different *es* values we have made few modifications in the parameterized design presented in Section-IV. The modifications are mentioned below. For a compile-time defined *es* values, these sizes are already mentioned in Algorithm-1 and 2 as derived parameters. In case of run-time defined *es* value, the exponent size is determined by the largest *es* value (*es=3*) and the fraction size by smallest *es*

we propose to support an additional rounding mode (round-to-zero mode) along with the default *posit* rounding mode (motivation of this is discussed in Section-VII-A). Thus, when the rounding mode (*rm*) is round-to-zero, *rb* is set to 0 (line 15). IEEE-754 performs several checks for setting the invalid and inexact flags during conversion to integer while *posit* does not set any flags as a consequence of these instructions.

H. Posit Comparison

FMIN.S, FMAX.S, FEQ.S, FLT.S and FLE.S instructions are defined in the 'F' extension for comparison between 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 *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 is quite burdensome as compared to *posit*. 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 shall set the NV flag for comparison operations (whenever applicable) while *posit* does not set any flags.

I. Posit Sign Injection

RISC-V ISA defines FSGNJ.S, FSGNJN.S, FSGNJX.S instructions for floating-point sign injection. FSGNJ.S is used to move values between registers, FSGNJN.S is used to negate
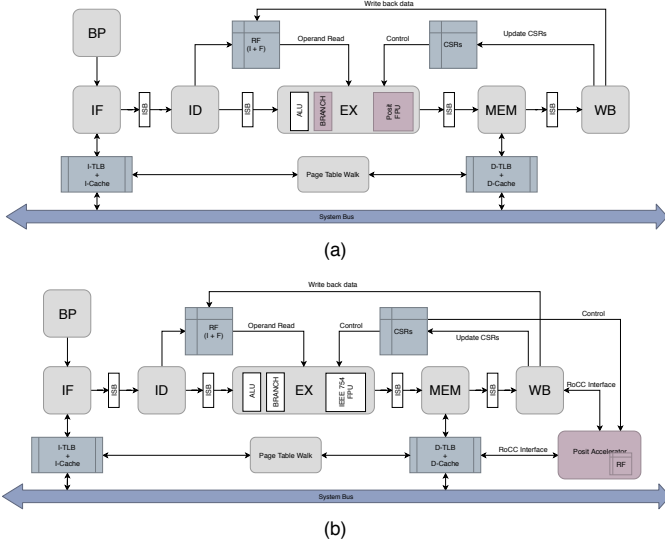
Fig. 3: Micro-architecture of the C-class core with *posit* FPU integrated. (a) *posit* FPU integrated as a tightly-coupled execution unit. (b) *posit* FPU integrated as an accelerator through RoCC

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-5 adjusts *e* bits for *es=2* after line 17 of Algorithm-1.

$$e = e \gg 1 \tag{5}$$

For the *posit* encode module in Algorithm-2, Equation-6 modifies *e* bits after line 3 for *es = 2*. Where as Equation-7 adjusts *k* value after line 4 for *es = 3*. Lastly, Equation-8 after line 14 shifts the concatenated *ef* valuer for *es = 2*.

$$e[2] = 0 \tag{6}$$
$$k = k \gg 1 \tag{7}$$
$$ef = ef \ll 1 \tag{8}$$

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 mantissa 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-VIII.

V. 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 [17] as our baseline core for integration. C-class is amongst one of the most configurable Linux-capable, in-order, RISC-V open-source core available. The C-class core is designed using Bluespec-System-Verilog (BSV) [24] and is a basic 5-stage in-order core which includes: a branch predictor, blocking instruction and data caches, fully-associative TLBs for instruction and data, a HW page-table-walk unit and AXI-4 compliant system bus interface. Figure-3 shows a high-level micro-architecture of the core.

For the purpose of this work, we have configured the core to support the RV32IMAFC extensions of the ISA (i.e. it supports, 32-bit Integer (I), multiplication/division (M), atomic (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 next subsections elaborate further on how the proposed *posit* FPU can be integrated as either a tightly-coupled execution unit or as an accelerator through RoCC interface.

A. Integration as a tightly-coupled execution unit

The default C-class core includes a single-precision IEEE-754 compliant floating point unit 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 unit with our *posit* unit.

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

Figure-3(a) 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.

B. Integration as an accelerator

Figure-3(b) 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. A major difference here as compared to the previous approach of integration is that 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.

As explained in Section-III-B, 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 custom opcodes is detected in the decode stage, the *xs1/xs2* fields are checked to see if any of integer registers are required for the current custom instruction. The execute and memory stage simply 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 response received from the accelerator.

With this approach, we are able to empower the existence of IEEE-754 and *posit* FPU in the same design and enable the *posit* community to develop more sophisticated and powerful *posit* compute blocks without having to touch the core-pipeline.

C. Verification of Posit FPU

We have used SoftPosit [25] library for verifying our *posit* implementation. Random inputs were generated and the corresponding outputs of each operation(*es=2*) were found to be in agreement with the result of the soft-posit 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*).

VI. SOFTWARE SUPPORT FOR POSIT

Since *Posit* has only recently been introduced, to the best of our knowledge compiler support for *posits* is not available for RISC-V. In light of this, some recent works such as [9], [26], [27] suggest converting IEEE-754 to *posit* in hardware, thereby allowing compatibility with existing tool-chain. However, we have observed that such conversion schemes can not leverage the benefits of *posit* format. The dynamic range and precision that can be expressed are still limited by a 32-bit IEEE-754. For, eg. we cannot represent the number 3.0E+40 in 32-bit IEEE-754 representation whereas it can be represented in 32-bit *posit* (es=3) as 3.000865123284026E+40. The dynamic range of 32-bit *posit* (for es=3 being 2.0E-75 to 5.0E+74) is greater than 32-bit IEEE-754 (7.0E-46 to 3.0E+38). Similarly, 15.996093809604645 is represented in 32-bit IEEE-754 as 15.99609375 because IEEE-754 fraction have a precision of 24 bits, whereas it is exactly represented in 32-bit *posit* (es=2) because *posit* fraction have a *max-precision* of 28 bits.

Another approach would be to store the data as a double-precision IEEE-754 and convert it to 32-bit *posit* in hardware. While this approach represents a short-time solution, it defeats the original purpose of having *posits*. The initial motivation to switch to *posit* was the memory wall problem. It is well known that once we reach the memory wall, the program execution time will depend almost entirely on the speed at which RAM can send data to the CPU. The intention of using *posit* was to replace 64-bit data with 32-bit data and thereby reduce the bandwidth requirement by half.

From the above arguments, we conclude that a complete software stack support is the only way forward to leverage the true potential that *posit* offers. While the up-streamed version of GCC supports the latest RISC-V ISA, we acknowledge the fact that the effort to provide *posit* support based on the

proposals made in Section-III is significant and beyond the scope of this work. Moreover, providing custom instruction support (as mentioned in Section-III-B) to GCC will be even more challenging. In consideration of this situation, we adopt the following approach to modify basic C/C++ applications to facilitate porting of these applications on our *posit* enable C-class core. Here, we leverage the 'F' extension provided by RISC-V as is and integrate the *posit* unit as a tightly-coupled execution unit in the C-class core as specified in Section-V-A.

Whenever a float variable is initialized in a program, the compiler assigns it the value according to IEEE-754 representation. For our purpose, we require this value to be represented as a *posit* equivalent. We achieve this by assigning the hexadecimal value of *posit* representation (computed manually) to an integer variable and using *memcpy* to transfer the contents of the integer variable to float variable. The below code snippet is an example of floating-point addition.

```
float f1pt5 = 1.5;
float f1pt2 = 1.2;
float a = f1pt5 + f1pt2;
```

In the above example, the float variables f1pt5 and f1pt2 have been assigned values as per the IEEE-754 format. We change this to equivalent *posit* format, as shown in the code snippet below:

```
float f1pt5 = 1.5;
float f1pt2 = 1.2;

int i1pt5 = 0x44000000;
memcpy(&f1pt5, &i1pt5, 4);
int i1pt2 = 0x4199999A;
memcpy(&f1pt2, &i1pt2, 4);

float a = f1pt5 + f1pt2;
```

Similarly, when a float constant is used in the program, we need to replace it with the *posit* equivalent. In the below code MAX is defined as a float constant. The compiler replaces MAX by its value during compilation.

```
#define MAX 1.5;
float f1pt2 = 1.2;
if(f1pt2 < MAX) ...
```

In the below snippet, we have used a float variable instead of constant. Now we can assign the *posit* equivalent of float constant as mentioned above.

```
float f1pt2 = 1.2;

float fmax;
int imax = 0x44000000;
memcpy(&fmax, &imax, 4);
int i1pt2 = 0x4199999A;
memcpy(&f1pt2, &i1pt2, 4);

if(f1pt2 < fmax) ...
```

We acknowledge the fact that making compilers compatible with *posits* can take up some time and the computing stack will not be complete for *posit* until such support is available.

#	Original	Posit (RNE)	Posit (RTZ)	IEEE 754
1	3.7KB	1.9KB	1.5KB	1.5KB
2	4.3KB	2.3KB	2.0KB	2.0KB
3	5.6KB	3.0KB	2.5KB	2.5KB

TABLE VI: Original and compressed image size in kBs after JPEG compression for 32-bit *posits* and IEEE-754

Function	Metric	Posit	IEEE 754 Float
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 VII: Mean and confidence interval of percentage error for 32-bit *posit* and IEEE-754 for trigonometric and exponential series

Component	Metric	Posit	IEEE 754 Float
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

TABLE VIII: Mean and confidence interval of percentage error for 32-bit *posit* and IEEE-754 for Fast Fourier Transform

However, the above-mentioned solution serves our purpose of running applications on *posit* based RISC-V core.

VII. APPLICATIONS

In this section, we present details of porting several applications on our *posit* enabled C-class core. Each of these applications has been modified as per Section-VI and 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.

A. Image Processing

We performed image processing tasks like JPEG compression, image filtering and edge detection with *posit* and IEEE-754 FPUs on the images given in Figure-4. In all the applications, the image inputs were represented as an array of integer values. We compressed three different variants of the image given in Figure-4a using IEEE-754 and *posits*. The original and compressed image sizes (in KBs) are given in Table-VI. We observed that *posit*, with the default rounding mode, produced larger compressed images as compared to those obtained using IEEE-754. The culprit, based on our analysis, seems to the rounding mode used while converting *posit* to integer. However, we observed that if conversion operation supported the RTZ (round-to-zero) rounding mode, the compression quality matched with those of 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-IV-G.

The image filtering application took a bitmap image (Figure-4b) as input and applied filters like blur, converting to grayscale and converting to sepia. The respective images are given in Figure-4c, Figure-4d, Figure-4e. With RTZ rounding mode in *posit*, we obtained similar results for 32-bit IEEE-754 and *posits* for this application. Similarly canny edge detection technique, which was used to detect edges in Portable Gray Map (PGM) images, was able to detect similar edges (Figure-4f) for IEEE-754 and *posits* with RTZ rounding.

B. Trigonometric and Exponential Series

Applications using trigonometric equations form an excellent platform to demonstrate how *posit* outperform 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 0-359 degrees. In case of e^x input takes values between 0-11. 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 [28] with confidence of 95% to estimate the range of mean percentage error. These results are tabulated in Table-VII. We observed that the mean percentage error for *posit* is 7x less in case of sin(x), 5x less in case of cos(x) and 7x less in case of e^x as compared to IEEE-754. Also, the confidence interval is non-overlapping in all cases. Thus, we conclude from Table-VII that *posit* stores more information than IEEE-754 for the same bit width; hence providing better accuracy.

C. Fast Fourier Transform (FFT)

Applications like designing and using antennas, image processing and filters, data processing and analysis, etc. use FFT for different purposes. We calculated FFT for a complex input vector with the real component as cosine values of numbers between 0-127 and 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-VIII shows that the mean percentage error for *posit* is 12x less in case of magnitude and 10x less in case of angle with respect to IEEE-754.

D. K-means Clustering

K-means algorithm is widely used in data science. We clustered some well-known data sets [29]–[31] using the k-means algorithm. After performing clustering, we calculated different cluster quality metrics using [32]. Values of all the metrics lie between 0 and 1, and a score near to 1 indicates better clustering. These metrics require the true and predicted

| (a) JPEG compress | (b) Original | (c) Blurred | (d) Gray | (e) Sepia | (f) Edge detect |

Fig. 4: Images used in image processing applications

k	Posit Passed	IEEE 754 Float Passed	No. of instances where *posit* outperforms
2	100	100	100
3	100	100	81
4	100	100	66
5	100	100	69
6	100	100	57
7	100	100	66

TABLE IX: Comparison of cluster quality for 32-bit IEEE-754 and *posit* in max-precision mode for synthetic data set with different k Values, where k indicates the number of clusters

k	Posit Passed	IEEE-754 Passed	No. of instances where *posit* outperforms
2	100	100	100
3	100	78	36
4	100	46	23
5	100	29	16
6	100	28	15
7	100	0	0

TABLE X: Comparison of cluster quality for 32-bit IEEE-754 and *posit* in max-dynamic range mode for synthetic data set with different k Values, where k indicates the number of clusters

label of data to measure the cluster quality. The true labels are present in the data set and predicted labels are generated by 32-bit IEEE-754 and *posit*. We observed that 32-bit IEEE-754 and *posit* gave similar clusters. To prove the superiority of *posit* over 32-bit IEEE-754, we generated a data set consisting 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 the ones generated by 32-bit IEEE-754 and *posit*.

1) max-precision mode: 32-bit *posit (es=2)* provides higher precision than 32-bit IEEE-754. We compared *posit* and IEEE-754 cluster using the cluster quality metrics to get the numbers of cases where *posit* is similar as or outperform IEEE-754. Table-IX shows the comparison of *posit* and IEEE-754 result. On average, *posit* provide similar or better results than IEEE-754 in 73% cases.

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. We multiplied the random input with a large number and clustered them using IEEE-754 and *posits*. Table-X shows the

comparison of *posit* and IEEE-754 result. IEEE-754 did not pass all the cases due to overflow during the calculation of large values, whereas *posit* passed all the cases as it can handle a larger dynamic range. Out of all the cases in which IEEE-754 passes, 32-bit *posit* provide similar or better results than 32-bit IEEE-754 on average of 51% cases. Including all the case in which 32-bit IEEE-754 fails, *posit* provide similar or better results in 85% cases.

VIII. HARDWARE RESULTS

This section presents the synthesis results of parameterized *posit* unit for *(ps=32,es=2)* and *(ps=32,es=3)* instances. Also, we present the overheads for dynamic switching in this section. We have synthesized our designs for an Artix-7 FPGA (xc7a100tcsg324-1) device using Xilinx Vivado 2018.3. We chose to constraint the designs to operate at 100 MHz as most open-source RISC-V cores operate at this frequency for the same FPGA target. It is to be noted that the results presented in this section are independent to the integration choices mentioned in Section-V.

Table-XI shows the LUT and slice-register utilisation on a modular basis. For this table, each module has been synthesized separately using the *Default Strategy* of Vivado with re-timing enabled. It should be noted that there is no separate module for dynamic switching as the entire operation is achieved through the decoding and encoding modules itself. Due to the presence of run-length encoding in *posit*, the encoding and decoding modules pose a strong design challenge and thus consume considerable resources.

One can observe from Table-XI that dynamic switching support requires 15% more LUTs and 8% more registers than the base *es=2* implementation.

Table-XII shows the number of cycles taken by different floating-point instructions in the *posit* based RISC-V core. We have pipelined the FMA module into 8 stages and to operate at 100 MHz. Since addition and subtraction operation do not need to multiply the fraction, they skip the product stage and thus require 6 cycles to complete. Similarly, multiplication result comes out of the pipeline as soon as the product is calculated, taking a total of 6 cycles. Division and square root operation use iterative non-restoring approaches to calculate quotient and remainder. Hence the number of cycles required is proportional to fraction length. Conversion operations (integer to *posit* and *posit* to integer) either involves decoding or encoding of the number and they require 3 cycles to complete the operation. Sign injection, move and classify instructions complete in 1 cycle. Switching the *es* value involves decoding and encoding logic and hence require 4 cycles.

Module	Posit(es=2)		Posit(es=3)		Posit(es=2,3)	
	Slice LUTs	Slice Registers	Slice LUTs	Slice Registers	Slice LUTs	Slice Registers
Fused Multiply-Add	1128	416	1154	415	1176	424
Division	281	270	303	237	279	271
Square root	138	145	135	143	138	147
Integer to Posit	131	37	128	36	131	37
Posit to Integer	163	39	161	39	165	39
Sign Injection	18	0	18	0	18	0
Classify	2	0	2	0	2	0
Decode Posit	554	0	555	0	684	0
Encode Posit	306	0	307	0	371	0
Glue Logic	307	284	281	311	543	376
Total	**3028**	**1191**	**3044**	**1181**	**3507**	**1294**

TABLE XI: FPGA Synthesis: Module-wise Slice LUTs and Slice Registers with *posit* FPU of *es=2*, *es=3* and *es=2,3* for 32-bit posit size at 100 MHz

Instructions	Cycles
FMADD.S, FMSUB.S, FNMSUB.S, FNMADD.S	8
FADD.S, FSUB.S	6
FMUL.S	6
FDIV.S	20
FSQRT.S	32
FCVT.W.S, FCVT.WU.S	3
FCVT.S.W, FCVT.S.WU	3
FMIN.S, FMAX.S, FEQ.S, FLT.S, FLE.S	1
FSGNJ.S, FSGNJN.S, FSGNJX.S	1
FMV.X.W, FMV.W.X	1
FCLASS.S	1
FCVT.ES	4

TABLE XII: No. of cycles required, by each instruction in RV32F standard extension of RISC-V spec in *posit* FPU (*es=2,3*) for 32-bit posit size at 100 MHz

IX. Related Work

This section provides a comparison of the contributions of this paper with related works in literature.

Some of the works in literature have failed in complying their implementations with the *posit* spec. For example, the works in [9], [10] 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. The *posit* FPU presented in this paper supports the rounding mode and complies exactly with the *posit* standard.

There exist a few works, like [12], [14], which try to compare their *posit* designs with IEEE-754 based designs to claim that *posit* is a suitable option to replace IEEE-754. In contrast to this, our work provides a solution where *posit* and IEEE-754 FPUs can co-exist on the same chip and enable users to choose either for their applications. In addition to this argument, we also believe that comparing hardware overheads of *posit* and IEEE-754 is not fair since IEEE-754 allows a large amount of flexibility in implementation choices (e.g., supported rounding modes, support for subnormals, nan-propagation, etc.) and choosing a fair design point for comparison with *posit* is difficult. The comparison becomes even more challenging when one accounts for circuit optimizations specific to each paradigm.

One of the crucial contributions missing in all the above-cited works (including [13], [15]) is the fact that none of the proposed FPUs are feature-complete to be integrated with a general-purpose processor. In this paper, we not only propose how a *posit* based FPU can be integrated with a RISC-V core but also present a few applications which truly capture the benefits of the *posit* over IEEE-754.

Works like [14], [15] have proposed a parameterized design of *posit* unit which can be used to generate hardware for any *ps* and *es* values. In our work, along with this, we go one step further and provide the capability to support multiple *es* modes within the same hardware unit, thereby catering to applications from a wide range of domains. This, to the best of our knowledge, has not been proposed by any previous work in literature.

Our work not only provides implementation details of *posit* arithmetic but also captures the difference between *posit* and IEEE-754 arithmetic, which support the claim of *posit* being more efficient than IEEE-754.

X. Conclusion

In this paper, we propose the first parameterized *posit* based FPU designed in BSV and integrate with a RISC-V compliant core. The paper provides a concrete path on how the RISC-V 'F' extension should be extended or modified to support *posit*. 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. The latter strategy thus enables a RISC-V core to support both IEEE-754 and *posit* on the same chip allowing applications to choose one over the other based on their requirements. The paper further goes on to highlight the specific differences between *posit* and IEEE-754 in regards to supporting the various 'F' extension instructions of RISC-V. This analysis leads us to conclude that *posit* simplifies floating-point arithmetic significantly as compared to IEEE-754. The paper also proposes a novel idea of supporting multiple *es* values within the same hardware unit with dynamic switching capabilities through minimal overheads. The paper also demonstrates how applications can be ported to a RISC-V core with *posit* support in the absence of appropriate compiler

support. Scrutiny of these applications further emphasize the benefits and claims made by *posit* over IEEE-754. Through FPGA prototyping, we show that an instance of the proposed *posit* FPU consumes around 3.5K LUTs and 1.3K slice registers on a 7-series FPGA.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff," *Solid-State Circuits Newsletter, IEEE*, vol. 11, pp. 33 – 35, 10 2006.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

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

[4] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1051–1056, 2018.

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

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

[7] W. Eric Wong, X. Li, P. A. Laplante, and M. Siok, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *Journal of Systems and Software*, vol. 133, 06 2017.

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

[9] M. K. Jaiswal and H. K. . So, "Universal number posit arithmetic generator on fpga," in *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2018, pp. 1159–1162.

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

[11] Z. Lehóczky, A. Retzler, R. Tóth, A. Szabó, B. Farkas, and K. Somogyi, "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*, ser. CoNGA '18. New York, NY, USA: ACM, 2018, pp. 4:1–4:7. [Online]. Available: http://doi.acm.org/10.1145/3190339.3190343

[12] A. Podobas and S. Matsuoka, "Hardware implementation of posits and their application in fpgas," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 138–145.

[13] J. Hou, Y. Zhu, S. Du, and S. Song, "Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on fpga," *Journal of Signal Processing Systems*, 11 2018.

[14] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized posit arithmetic hardware generator," 10 2018, pp. 334–341.

[15] M. K. Jaiswal and H. K. . So, "Pacogen: A hardware posit arithmetic core generator," *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019.

[16] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," 2014.

[17] N. Gala, A. Menon, R. Bodduna, G. S. Madhusudan, and V. Kamakoti, "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)*, ser. VLSID '16. Washington, DC, USA: IEEE Computer Society, 2016, pp. 7–8. [Online]. Available: http://dx.doi.org/10.1109/VLSID.2016.130

[18] K. Asanovi, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[19] "The lowRISC project," https://www.lowrisc.org/.

[20] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "Pulp: A parallel ultra low power platform for next generation iot applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015, pp. 1–39.

[21] W. Digital, "RISC-V SweRV Core," https://blog.westerndigital.com/risc-v-swerv-core-open-source/.

[22] Bluespec, "Open-source RISC-V CPUs," https://github.com/bluespec/Piccolo.

[23] C. Yarp, "An Introduction to the Rocket Custom Coprocessor Interface," http://www-inst.eecs.berkeley.edu/~cs250/sp16/disc/Disc02.pdf, accessed on 10 November 2017.

[24] "Bluespec Inc. Bluespec System Verilog," https://bluespec.com/.

[25] "Softposit library," https://gitlab.com/cerlane/SoftPosit.

[26] "Posit Numeric Unit (PNU) Implementation by Calligo Technologies," https://posithub.org/conga/2018/docs/9-Calligo-Technologies.pdf.

[27] "Posit Implementation by VividSparks," http://vivid-sparks.com/.

[28] "Confidence interval," http://hamelg.blogspot.com/2015/11/python-for-data-analysis-part-23-point.html.

[29] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 7, pp. 179–188, 1936.

[30] N. Street, W. H. Wolberg, and O. L Mangasarian, "Nuclear feature extraction for breast tumor diagnosis," *Proc. Soc. Photo-Opt. Inst. Eng.*, vol. 1993, 01 1999.

[31] P. Horton and K. Nakai, "A probabilistic classification system for predicting the cellular localization sites of proteins," *Proceedings / ... International Conference on Intelligent Systems for Molecular Biology ; ISMB. International Conference on Intelligent Systems for Molecular Biology*, vol. 4, pp. 109–15, 02 1996.

[32] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.