PERC: Posit Enhanced Rocket Chip

Arunkumar M. V.
Sai Ganesh Bhairathi
Harshal G. Hayatnagarkar
arunkumar.mv@thoughtworks.com
saiganeb@thoughtworks.com
harshalh@thoughtworks.com
Engineering for Research
ThoughtWorks Technologies India
Pune, Maharashtra, India - 411006

ABSTRACT

Balancing between precision and performance is a known trade-off in system design. Universal Number system attempts to dissolve this trade-off with its flexible arbitrary precision bound within fixed bit length. Its Type-III class, also known as Posits, has been especially introduced to be hardware implementation friendly. Posit is a dynamic floating-point representation that ensures better accuracy and precision using a system that minimizes the number of unusable representations and introduces a higher dynamic range which can serve as a substitute for the IEEE-754 2008 floating-point standard.

In this paper, we share our experience with implementation and integration of a Posit Processing Unit (PPU) into the Rocket Chip SoC generator. This PPU replaces the IEEE-754 2008 FPU inside the chip, and supports both of RISC-V ISA floating-point extensions namely 'F' for single precision and 'D' for double precision using 32-bit and 64-bit posits respectively. We discuss various design choices that were available to us and the decisions made in this work. We elaborate our use of Chisel, a Scala embedded hardware construction DSL, to describe our design. Later we observe how various constructs in Chisel help not only to describe a product but also aids the description process in a robust, flexible and efficient manner.

We further delve into how the design has been tested using a version of the RISC-V ISA test suite which has been modified for Posit arithmetic numbers. The paper also discusses the scope for future work that can be done, including a posit arithmetic accelerator and higher-level toolchain support for posits.

ACM Reference Format:
Arunkumar M. V., Sai Ganesh Bhairathi, and Harshal G. Hayatnagarkar. 2020. PERC: Posit Enhanced Rocket Chip. In *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

CARRV 2020, May 30, 2020, Valencia, Spain

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Precision and performance of arithmetic processing is a known trade-off of a system design. The IEEE 754 fixed width 32-bit and 64-bit precision arithmetic, while tuned for performance, suffers in the precision. The arbitrary precision computing while solves the precision side of the problem, it suffers on the performance front [13]. The precision and performance requirements of a real number representation depends on its usage. A number representation that can be adjusted for the specific needs of precision can become useful. Universal Number format (*unum*) attempts to address this need.

Unum attempts to tackle this trade-off with its flexible arbitrary precision bound within fixed bit length via its Type-III class, also known as Posits. Posit is a dynamic floating point representation that ensures better accuracy and precision using a system that minimizes the number of unusable representations and introduces a higher dynamic range which can serve as a substitute for the IEEE-754 2008 floating point standard. Posit has been specifically introduced to be hardware implementation friendly. A question could arise as to which hardware to choose for implementation. For research and prototyping, it is convinient to choose an open source option. RISC-V offers such an option for being an open standard instruction set architecture (ISA) [26], and the Rocket Chip, a synthesizable implementation of RISC-V [1].

In this paper, we share our experience to implement and integrate a Posit Processing Unit (PPU) into the Rocket Chip SoC generator. This PPU replaces the IEEE-754 2008 FPU inside the chip, and supports both of RISC-V ISA floating-point extensions namely 'F' for single precision and 'D' for double precision using 32-bit and 64-bit posits respectively.

We discuss various design choices that were available to us and the decisions made in this work. We elaborate our use of Chisel, a Scala embedded hardware construction DSL, to describe our design. We also share our observations about how various constructs in Chisel help not only to describe a product design but also aids the process of description in a robust, flexible and efficient manner. We continue to explain the design of the PPU, with a focus on modularity and later reuse. We further delve into how the design has been tested using a version of the RISC-V ISA test suite which has been modified for Posit arithmetic numbers. We finally conclude this paper mentioning possible enhancements for future work.

In the next section, we discuss similar related work as well as technologies on which we have built PPU.

2 BACKGROUND TECHNOLOGIES AND WORK

Even though posit arithmetic was introduced recently, extensive work has gone towards its adoption. This includes studies in the viability of posit arithmetic as a replacement for IEEE-754 floats [7, 19, 24], software implementations[18, 22] and hardware implementations for different posit arithmetic units [4, 6, 16, 17]. These hardware implementations were of independent units and did complete computing support for a complete ISA. A recent similar work namely *PERI* implements a posit arithmetic unit on top of the RISC-V implementation *SHAKTI* processor [10, 23]. In the rest of this section, we will discuss technologies that form the foundation of our work.

2.1 Unum and Posit Arithmetic

The Universal Number (or ***Unum*** for short) is a binary representation format for real numbers, to address the precision error rooted in the fixed width fraction and exponent representation in floating point formats such as IEEE 754 [12]. However, due to variable length representation, original unum types I and II are not friendly to hardware implementation. The latest standard ***Unum Type-III*** called ***posits*** is a hardware friendly version of unums. Posits ensure more accuracy and precision using a system that minimizes the number of unusable representations and introduces higher dynamic range, resulting in superior accuracy compared to IEEE-754 floats [7][19].

The equation (1) shows the formal posit format. The format is a dynamic floating point representation which has two fixed parameters: posit size(ps) and exponent size(es).

$$\underbrace{\overbrace{s}^{Sign}\ \overbrace{r\ r\ r\ \cdots\ \bar{r}}^{Regime}\ \overbrace{e_1\ e_2\ e_3\ \cdots\ e_{es}}^{Exponent,\ if\ any}\ \overbrace{f_1\ f_2\ f_3\ \cdots}^{Fraction,\ if\ any}}_{posit\ size} \tag{1}$$

Similar to floats, posits also contain the sign, exponent and fraction bits with an additional field called regime. Negative numbers are represented using 2's complement in posits. Regime bits come after the sign bit and are a set of similar bits terminated by a complementary bit. The number of similar bits in the regime field rc gives the regime value k (2). es bits occurring after the regime gives the posit exponent e. The actual exponent value exp is derived from the regime value k and posit exponent bits e (3). The remaining bits trailing after exponent bits are the fraction bits f. The posit value x is obtained according to (4) after adding the hidden bit to the fraction which is always 1 in the case of posits.

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

$$exp = (k \ll es)\ +\ e \tag{3}$$

$$x = \begin{cases} 0 & P = 000\cdots000 \\ NaR & P = 100\cdots000 \\ (-1)^s \times 2^{exp} \times 1.f & \text{otherwise} \end{cases} \tag{4}$$

As a consequence of the run length encoding of the regime the exponent and fraction bits are optional, and come after regime bits if bit positions are left. This also means that the higher precision numbers can be represented with much smaller es values. The dynamic range of a posit number is dependent on the es value as larger es can be used to represent exponent values. This form of scaled dynamic representation also minimizes the number of unusable representations contrary to the IEEE-754 2008 floating point standard which features fixed width bit fields. It has a single representation for 0, another one for $+\infty$ or $-\infty$ called *'Not A Real'*, and does not support subnormal representations. All these factors taken together facilitate a hardware friendly number system that can serve as a viable replacement for the IEEE-754 floating point standard.

2.2 RISC-V and Rocket Chip SoC generator

RISC-V is an open standard instruction set architecture (ISA) which features a minimal base integer ISA(I instructions) with optional standard extensions [26]. The standard extensions include support for real number computations and provisions for custom instructions. This modular and extensible structure has enabled open source processor implementations around RISC-V such as *SHAKTI*[10], *Rocket Chip/BOOM* [1], *Ariane* [27].

RISC-V ISA supplies the F and D standard extensions to support single and double precision floating-point computations using the IEEE-754 2008 arithmetic standard. It also has reserved two major opcode spaces for custom instructions meaning that they can be utilized with the assurance that no future standard extensions will utilize these opcodes. In a later section, the paper proposes how these ISA extensions can be leveraged to provide support for posit arithmetic.

Rocket Chip is a flexible and parameterized system-on-chip (SoC) generator which emits synthesizable register-transfer level (RTL). Designed using the Chisel hardware construction language [2], it features an extensive library of generators for cores, caches and interconnects required for the integrated SoC. Rocket chip also supports integration of custom co-processors (or accelerators) by adding them as tightly coupled units or standalone processors [1]. These chip building generator libraries can be used to generate different SoC variants. This is enabled by the use of standard interfaces to interconnect different units. This "plug-n-play" approach to hardware generation allows different designs to be generated by just changing the configuration files, keeping the actual hardware description source code untouched. Thus hardware designers can easily add extensions without breaking the original design.

The solution proposed in this paper also leverages this feature to add posit support to the Rocket Chip.

2.3 Chisel and FIRRTL

Chisel (Constructing Hardware In a Scala Embedded Language) is an open source hardware construction DSL embedded in the Scala programming language[2]. By adding hardware construction primitives to the Scala language, it enables designers to write complex, parameterizable circuit generators. It highly resembles a traditional hardware-description language (HDL) such as Verilog rather than a high-level synthesis language. In fact, Chisel when compiled, emits

synthesizable Verilog through an intermediate representation called *FIRRTL*[15]. It brings features of modern software languages like object-oriented and functional programming to circuit description.

Chisel also comes with a variety of testers and simulators including *Chisel IO Testers* and *Verilator*[5]. Verilator can generate a fast, cycle-accurate RTL simulator implemented in C++ from the Verilog emitted from Chisel, which is functionally equivalent to but significantly faster than other Verilog simulators.

Based on this understanding of the Rocket chip and its ecosystem including Chisel, we have developed a floating point compatible Posit Processing Unit (PPU), discussed in the rest of the paper.

3 THE POSIT ARITHMETIC MODULES

In this section we discuss the arithmetic modules that make up the PPU. These modules have been implemented in Chisel which allows for parameterizable circuit generation of modules for any *ps* and *es* value. Being a Chisel library, this modularity enables reusability of these modules in alternate implementations such as SoC units and accelerators.

3.1 Posit Extractor

The posit extractor extracts the sign exponent and fraction from an input posit number. The unit also detects whether the input posit number is 2 special values 0 or *NaR* contrary to the IEEE-754 representation which requires checking for 5 different special values: *qNaN*, *sNaN*, infinity, zero and subnormal numbers. The exponent in a posit number is scaled using a run length encoded regime. Regime extraction involves counting similar adjacent bits after the sign bit. Algorithm 1 gives the algorithm implemented by the extractor.

Algorithm 1: Algorithm for Posit extraction

Input : P: Posit number of *ps* bits
Output : rs: Sign of P, re: Exponent of P, rf: Fraction of P, isZero: Indicate if P is zero, isNaR: Indicate if P is *NaR*

```
isZero ←~| P
isNaR ← P[ps - 1]& ~| P[ps - 2 : 0]
rs ← P[ps - 1]
if rs then
    P ←~ P + 1
k, rc ← extarctRegime(P)
P ← P ≪ (rc + 2)
e ← P[ps - 1 : ps - es]
re ← e + k ≪ es
rf ← 1 || P ≪ es
```

3.2 Posit Generator

The posit generator is responsible for generating a posit encoded number from the real number fields supplied to it. The regime is extracted from the input exponent value and added to the final posit number along with the remaining *es* exponent bits and fraction bits. A sticky bit is also supplied to the generator which is the logical OR of all the bits shifted out from previous circuit levels. Proper rounding is carried out using the truncated fraction bits and the input sticky bit. Posit supports only one rounding mode that is round to nearest even while IEEE-754 supports 5 rounding modes. Rounding is carried out in such a way that if the posit will not be rounded up to *NaR* (in case of maxpos) or rounded down to zero. Algorithm 2 gives the algorithm implemented by the generator.

Algorithm 2: Algorithm for Posit Generation

Input : s: Sign for P, e: Exponent for P, f: Fraction for P, isZero: Indicate if P is zero, isNaR: Indicate if P is *NaR*, sb: Sticky bit
Output : P: Generated posit number of *ps* bits

```
e, f ← checkIfNormalized(e, f)
k ← e ≫ es
c ← 3
if e[MSB] then
    k ← −k
    c ← 2
exp ← e[es - 1 : 0]
pshft ← k + es + c
rb ← generateRegime(k)
P ← rb || exp || f
sb ← sb | (| P[pshft- 2 : 0])
ab ← P[pshft- 1]
lb ← P[pshft]
rb ← (lb & ab) | (ab & sb)
P ← P ≫ pshft
if (~ P[ps + 1] & ( & P[ps - 2 : 0])) then
    rb ← 0
if P = 0 then
    P ← 1
P ← P + rb
if s then
    P ←~ P − 1
if isZero then
    P ← 0
if isNaR then
    P ← NaR
```

3.3 Posit Fused Multiply-Add

The Fused Multiply-Add module performs the FMA operation which takes in 3 posit numbers, multiplies the first 2 numbers and adds the product with the third number, implemented using Algorithm 3. It has 2 additional boolean inputs, *neg* to negate the result and *sub* to perform subtraction instead of addition. The FMA module can also be used to perform normal addition, subtraction and multiplication to reduce/reuse resources. The module also checks if the result is zero or *NaR* but does not need any exception flags as these are silent exceptions. In contrast to IEEE-754 floating point FMA which has to check for overflow and underflow of the product exponent

and normalization of subnormal product fractions, the posit FMA module is only burdened with overflow checking for the product fraction.

Algorithm 3: Algorithm for Posit Fused Multiply-Add

Input : num_1: First posit operand, num_2: Second posit operand, num_3: Third posit operand
Output : P: FMA result posit number of *ps* bits

$s_1, e_1, f_1, isNaR_1, isZero_1 \leftarrow$ `PositExtractor(`num_1`)`
$s_2, e_2, f_2, isNaR_2, isZero_2 \leftarrow$ `PositExtractor(`num_2`)`
$s_3, e_3, f_3, isNaR_3, isZero_3 \leftarrow$ `PositExtractor(`num_3`)`
$rIsNaR \leftarrow isNaR_1 \mid isNaR_2 \mid isNaR_3$
$rIsZero \leftarrow (isZero_1 \mid isZero_2)\ \&\ isZero_3$
$Ls \leftarrow s_1 \oplus s_2 \oplus neg$
$Ss \leftarrow s_3 \oplus neg \oplus sub$
$Se, Sf \leftarrow e_3, f_3$
$Le \leftarrow e_1 + e_2$
$Lf \leftarrow f_1 \times f_2$
$Le, Lf \leftarrow$ `checkOverflow(Lf)`
if $(Se > Le) \mid (Se = Le\ \&\ (Sf > Lf))$ **then**
  `swap(Ls, Ss), swap (Le, Se), swap(Lf, Sf)`
$rs \leftarrow Ls$
$re \leftarrow Le$
$expDiff \leftarrow Le - Se$
$sb \leftarrow \mid Sf[expDiff\text{-}\ \boldsymbol{1:0}]$
$Sf \leftarrow Sf \gg expDiff$
if $Ls \oplus Ss$ **then**
  $Sf \leftarrow -Sf$
$rf \leftarrow Lf + Sf$
$re, rf, sb \leftarrow$ `checkOverflowAndNormalize(re, rf, sb)`
$P \leftarrow$ `PositGenerator(rs, re, rs, rIsZero, rIsNaR, sb)`

3.4 Posit Division and Square Root

The Division square root module takes in 2 posit numbers and performs division of the 2 numbers or the square root of the first number based on a boolean input *op*. Division and square root are performed using a sequential non restoring algorithm. Algorithm 4 is implemented by the module. The module uses a FIFO (ready/valid) interface for accepting inputs and 2 valid signals to indicate when the output is valid for each case(division or square root).

For IEEE-754 division 5 exceptions can occur while for posits the only possible exception is divide by zero which is detected by the module. In the case of square root for posits no exceptions can occur.

3.5 Posit Comparison

Comparison of posits is exactly the same as that of integer comparison. This is a consequence of the dynamic representation and the fact that negative posits are represented using 2's complement similar to integers. This simplifies comparison to a huge extent compared to IEEE-754 which must detect multiple exceptional cases like

Algorithm 4: Algorithm for Posit Division Square Root

Input : num_1: First posit operand, num_2: Second posit operand(ignored if sqrtOp is high), sqrtOp: Indicates whether division or square root is to be performed
Output : P: Division or Square Root result posit number of *ps* bits, exception: Exceptions occurred

$s_1, e_1, f_1, isNaR_1, isZero_1 \leftarrow$ `PositExtractor(`num_1`)`
$s_2, e_2, f_2, isNaR_2, isZero_2 \leftarrow$ `PositExtractor(`num_2`)`
$rIsNaR \leftarrow isNaR_1 \mid (sqrtOp\ \&\ s_1) \mid (\sim sqrtOp\ \&\ isNaR_2)$
$rIsZero \leftarrow isZero_1$
$exception \leftarrow \sim sqrtOp\ \&\ isZero_2$
$rs \leftarrow \sim sqrtOp\ \&\ (s_1 \oplus s_2)$
if sqrtOp **then**
  $re \leftarrow e_1 \gg 1$
else
  $re \leftarrow e_1 - e_2$
if $sqrtOp\ \&\ e_1[\boldsymbol{0}]$ **then**
  $f_1 \leftarrow f_1 \ll 1$
$rf, sb \leftarrow$ `nonRestoringDivSqrt(`f_1, f_2, sqrtOp`)`
$rf, re \leftarrow$ `normalize(rf)`
$P \leftarrow$ `PositGenerator(rs, re, rf, rIsZero, rIsNaR, sb)`

comparison between +0 and −0 or between *NaN* representations while posits face no such exceptions.

The compare module also utilizes signed integer comparison to compare 2 input numbers and outputs 3 boolean signals that indicate if the first number is less than, equal to or greater than the second number.

3.6 Posit-to-Integer Converter

The Posit-to-Integer converter takes in a posit number of *ps* bits and outputs a signed or unsigned integer of *iw* bits. A boolean input *us* indicates whether the output is signed or unsigned. The module is parameterized by *iw* to support different output integer widths. The integer is computed by offsetting the fraction by the exponent size and extracting the most significant bits as depicted in Algorithm 5. As posits do not support inexact representations no checks are performed in contrast to IEEE-754.

3.7 Integer-to-Posit Converter

The Integer-to-Posit converter takes in an integer of *iw* bits and outputs a posit number of *ps* bits. A boolean input *us* indicates whether the input integer is signed or unsigned. Similar to the posit to integer converter the module is parameterized for different input integer widths. The integer is converted to posit by counting the most significant zeros to derive the exponent and fraction bits as captured in Algorithm 6. Compared to IEEE-754 which has to check for inexactness, integer to posit conversion does not require any such checks.

Algorithm 5: Algorithm for Posit-to-Integer Conversion

Input : P: Posit number of *ps* bits, us: Indicate whether output is signed or unsigned
Output : I: Integer result of *iw* bits

```
s, e, f, isNaR, isZero ← PositExtractor(P)
Is ← s & ~ us
f ← f ≪ e
If ← f[iw + ps − 1 : ps]
if us & (e ≥ iw) then
    If ← 2^ps − 1
if ~ us & (e ≥ (iw − 1)) then
    If ← 2^(ps−1) − 1
rb ← f[ps − 1]
I ← round(If, rb)
if Is then
    I ←~ I + 1
```

Algorithm 6: Algorithm for Integer to Posit Conversion

Input : I: Integer *iw* bits, us: Indicate whether input is signed or unsigned
Output: P: Posit result of *ps* bits

```
rIsZero ← I = 0
rIsNaR ← 0
rs ← I[iw - 1] & ~ us
if rs then
    I ←~ I + 1
zc ← countMSBZeros(I)
I ← I ≪ zc
re ← (iw − 1) − zc
rf ← (1 ‖ I[iw - 2: 0]) ≪ ps − iw + 1
sb ← 0
if iw > ps + 1 then
    sb ← (| I[iw − ps − 2 : 0])
P ← PositGenerator(rs, re, rf, rIsZero, rIsNaR, sb)
```

3.8 Posit-to-Posit Converter

The Posit-to-Posit Converter take in a posit of total size *ips* and exponent size *ies* and converts it to a posit of *ops* and exponent size *oes*. The module extracts the fields from the input posit, shifts the fraction if the conversion is widening or truncates it for a narrowing conversion, and generates the new posit. This conversion does not produce any exceptions.

4 RISC-V INSTRUCTION SUPPORT

The RISC-V ISA standard does not inherently support posit arithmetic operations. The 2 possible approaches to add support for posit arithmetic are overloading the predefined extensions for floating-point computations or utilizing the custom opcode space that the ISA provides. Here we discuss the benefits and caveats of each approach.

4.1 Overloading the F and D extensions

The single precision (F) and double precision (D) RISC-V ISA extensions supply floating-point computational instructions compliant with the IEEE-754 2008 arithmetic standard for 32 bit and 64 bit floats respectively [26].

It is notable that most features of posits map perfectly to the F and D instructions [11]. To provide support for both 32-bit and 64-bit posits using the F and D extensions we propose a few deviations from the standard.

The same 32 register architecture as that proposed by the ISA can be utilized although with one small deviation. The D extension proposes *NaN* boxing to support writing narrow floats to the register by setting the remaining most significant bits to 1. This is not possible in posits as all representations other than *NaR* are valid. There is no method to differentiate between boxed and unboxed values. Thus, the writes to the register can be the unchanged posits itself with narrow writes padded with zeros, assuming that the contents of the registers will not be used by higher level software interchangeably. Another remedy for this problem would be to maintain a separate 32-bit register to store the width of the last write into (0 for 32-bit register write and 1 for 64-bit register write) and read from the register file accordingly.

The floating point control and status register is a 32-bit register which contains 5 bits for exception flags, 3 bits to indicate rounding mode and remaining bits are reserved by the ISA. For posit compatibility given that the occurrence of *NaR* is a silent exception we only require one exception flag for DZ (divide by zero). Also posit only supports one rounding mode that is round to nearest even, so the rounding mode field also loses its significance, although rounding mode support for posit to integer conversion may be of significance for certain applications[23].

4.2 Utilizing Custom Instructions

The RISC-V ISA has reserved certain opcode spaces for custom extension to provide support for user defined functionalities. This adds the possibility of a posit arithmetic unit being added to Rocket chip as a co-processor leveraging the RoCC interface and coexists with the IEEE-745 2008 FPU rather than replacing it. We discuss such a possibility in a later section of this paper.

Our approach in this paper leverages the F and D standard extensions by adding a new Posit Processing Unit(PPU) to the Rocket core pipeline. This approach requires the minimal modification of the software toolchain, whereas utilizing custom instructions, as we understand, would involve modification of the RISC-V toolchain including compilers and debuggers [3].

We were well aware that for providing complete support for posits which includes quire register functionalities, an accelerator/co-processor model would be necessary as the F and D extensions do not provide such instructions. Though this is the case, a FPU-compatible PPU in the core pipeline seems the right way to go to verify if posits can be a viable replacement for the IEEE-754 2008 floating point standard and whether a RISC-V extension for posits should be considered for a future addition.

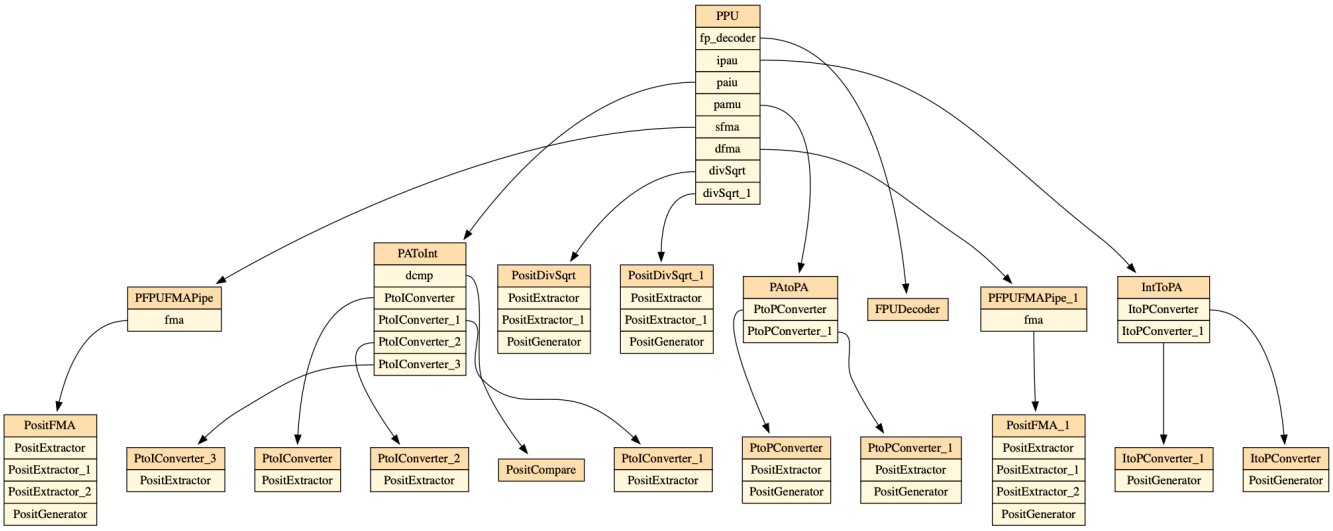

Figure 1: Circuit Hierarchy of the PPU generated using diagrammer[8]

5 THE POSIT PROCESSING UNIT

In this section we delve into how a FPU-compatible Posit Processing Unit (PPU) has been implemented using the posit arithmetic modules mentioned previously. The PPU is modelled in a fashion similar to the already existing IEEE-754 2008 FPU inside Rocket Chip which internally uses the Berkeley hardfloat implementation. The standard posit configuration used for single precision instructions is ps = 32, es = 2 and for double precision instructions is ps = 64, es = 3. The PPU consists of multiple inner modules which together perform operations as mandated by the input instruction. Figure 1 captures the top level circuit hierarchy of the PPU.

5.1 Floating Point Instruction Decoder

The floating-point instruction decoder is the same decoder used inside the IEEE-754 2008 FPU. When the decoder is given the instruction as input it outputs a set of FPU control signals which indicate the different operations to be performed by the PPU.

5.2 Posit to Integer Unit

This module takes care of all instructions which take in posits from the PPU register file and produce an integer result to be written into the integer register file. These instructions include the posit-to-integer conversion instructions, classify instructions, compare instructions, move posit to integer register file instructions and load/store instructions.

The module contains 1 to 4 posit-to-integer converters to support integer conversion instructions. The unit also contains a posit compare module for the compare instructions `FEQ, FLT` and `FLE`. The classify instruction `FCLASS` for IEEE-754 requires the transfer of a 10 bit mask to classify the number while for posits only 3 bits are required, first bit for sign, second to indicate if the number is *NaR* and third to indicate if the number is zero. The move instructions `FMV.X.W` and `FMV.X.D` move a value from the PPU register file to an integer register. Posits are sign extended if a narrower posit is moved to a wider integer registers.

5.3 Integer to Posit Unit

This module supports all instructions involving data taken from the integer register file and produces results to be written into the PPU register file. These instructions include integer-to-posit conversion instructions and move from integer register file instructions.

The module contains 1 to 2 integer-to-posit converters to convert the input integer to posit. The move instructions `FMV.W.X` and `FMV.D.X` move a posit encoded value from the integer register file to a PPU register. If a narrower integer is written into a wider posit register it is also sign extended.

5.4 Posit to Posit Unit

This module is responsible for all operations involving instructions with source and destination registers in the PPU register file excluding fused multiply-add and division square root operations. These instructions include posit-to-posit conversion instructions, sign injection instructions and min/max instructions.

The module contains 0 or 2 posit to posit converters based on whether both F and D extensions are supported to convert between 32-bit and 64-bit posits. The module uses the output from the compare module inside the posit to integer unit to compute the result of the min/max instructions `FMIN` and `FMAX` that computes which of 2 input numbers is the minimum or maximum.

The sign injection instructions `FSGNJ, FSGNJN` and `FSGNJX` change the sign of a posit number based on the sign of another posit number. As negative posits are represented using 2's complement, sign change also requires 2's complementing the number while IEEE-754 only requires toggling the sign bit.

5.5 Posit FMA Pipe Unit

This module interfaces a posit fused multiply-add module with the PPU through a pipe interface which adds a latency to for proper FMA operation. It takes care of all the fused multiply add instructions `FMADD, FMSUB, FNMSUB, FNMADD, FADD, FMUL` and `FSUB`. The PPU contains 1 to 2 of these units based on whether both F and D extensions are supported.

5.6 Posit DivSqrt Unit

The PPU also contains 1 to 2 posit division square root modules to support the `FDIV` and `FSQRT` instructions. As division and square root take multiple clock cycles to be performed, the FIFO interface of the division square root module is used to indicate the completion of the operation in the pipeline. This is the only unit that will set an exception flag in the `fcsr` register which happens if the divisor is zero.

The variable number of units inside the PPU and converters in the inner modules are determined by the base integer ISA variant used (RV32I or RV64I), and whether both F and D extensions are supported. These are all parameterized, and can be configured for circuit generation.

Currently the decoding/extraction and encoding/generation of the posit happens in the arithmetic pipeline and not just during load and store instructions. This enables us to avoid a separate internal representation which could lead to rounding errors but introduces a toll on the pipeline and resource usage[7]. Although this is a foreseeable problem it can be ignored in a complete posit implementation which uses a quire register where values are accumulated and extraction/generation is not performed for every computational instruction.

The Rocket Chip project consists of a collection of parameterized chip-building libraries that one can use to generate different SoC variants. As mentioned earlier different designs can be generated by changing the configuration files, keeping the hardware source files untouched. Rocket Chip's advanced configurability has also been utilized in adding the PPU to the Rocket chip. This was done by adding a new configuration which, when invoked during circuit generation, replaces the Rocket Chip's in house FPU with the PPU.

6 VERIFICATION AND OBSERVATIONS

6.1 Testing and Verification

The posit arithmetic modules were tested initially using unit tests which were written using Chisel IO testers. The modules have also been tested using a random test generator created utilizing the universal numbers C++ template library[22]. After integrating the modules into the PPU and adding it to Rocket chip, integration tests were performed using Rocket Chip's emulator enabled using Verilator[5]. For the test cases we modified the RISC-V ISA assembly test suite and added tests for posit arithmetic numbers using F and D instructions.

Currently we are working towards verifying the design on hardware and generating sufficient benchmarks.

6.2 Observations

6.2.1 Posit Software Support. As posit arithmetic is a fairly recent introduction to the arithmetic computing space, no compiler in RISC-V or any other architecture that currently supports posits, to the best of our knowledge. One approach that is suggested by recent works[4, 16] to provide ISA support for posits is to accept the IEEE-754 floating-point number from software and convert it to posit in hardware to maintain compatibility with the existing toolchain. This however will reduce the potential of the posit format. As the maximum precision of posit is much greater than that of IEEE-745 floats for certain ranges of numbers, converting them to and from IEEE-754 floating-point will lead to loss of this precision and introduces unnecessary overhead[23].

The lack of support for posits and the additional effort for adding instruction to the existing toolchain is one of the main reasons why we decided to leverage the F and D extensions. To achieve this the value of the float is replaced with the value of the posit number in the program itself. All subsequent floating-point instructions will operate upon the new posit values.

6.2.2 RISC-V Posit Arithmetic ISA Extension. Other than the two approaches proposed earlier which were overloading the F and D extensions and utilizing the ISA custom opcode space, another method to provide posit support in RISC-V would be a standard ISA extension for posit arithmetic. Such an extension has been proposed for 32-bit posit support[11] which is similar to the F extension but adds some instructions and removes some unnecessary ones.

There are more than one benefits of a standard posit ISA extension. The extension would be able to provide some instruction that are posit specific like quire manipulation operations. This extension should guarantee support for floating-point and posit instructions in the same architecture to support applications relying on floating-point arithmetic. In addition to this, the custom op-code space that RISC-V would stay free for other purposes. Lastly, developer toolchains may not be interested in custom op-codes, but in a standard extension. We trust enabling support for a posit ISA extension in the compilers and debuggers would be worth those efforts. Overall, the extension could benefit domain-specific applications especially related to high performance computing, data-intensive computing, and machine learning.

6.2.3 Chisel. As mentioned earlier, we used Chisel to describe our design, which provides circuit-design specific constructs in Scala language. For example, Chisel circuits are Scala classes.

So, even basic object-orientation helps to capture structural aspects of a circuit design, such that encapsulation, inheritance, and polymorphism bring in scoping, reuse, and flexibility to handle diversity of implementations. Generics, traits, and other features help to capture a meta-structure. Other features like a rich type-system with support for structured data and bundled interfaces, width inference for wires, high-level descriptions of state machines, and bulk wiring operations help make a lot of the design generic and reusable. For example, one can represent a circuit for adders, irrespective of its bit-width, which is made possible by combining features like generics and traits. Even further, parameterization of generics offers composition of meta-structures, so that the designers describe a circuit generator rather than a circuit itself. Functional programming

features such as higher order functions allow importing decisions and behavior from outside of a structure.

Scala ecosystem benefits via integrated development environments, build tools and libraries. Static code analysis can flag potential and actual defects in the source code namely 'Code Smells'. In addition, the issues that are not caught there, developers can write unit tests as supported by Chisel testers via Scala. The test-driven development enables regression testing to discover breaking changes sooner[9]. This gives an ability to create separately packaged hardware libraries coupled unit tests. Overall, the cumulative productivity boost helps to practice agile hardware development.

7 CONCLUSION AND FUTURE WORK

In this paper, we design a parameterized posit processing unit using a library of posit arithmetic modules and integrate it with the Rocket Chip SoC generator. We present how these modules were constructed, followed by a discussion of using the RISC-V ISA 'F' and 'D' extensions for posit arithmetic, or utilizing the custom opcode space. We also mention the deviations from the 'F' and 'D' extensions to accommodate posits. From this we were able to conclude that posits decrease the complexity involved in floating point arithmetic compared to IEEE-754 by minimizing unusable representations and exceptional cases. Later we document the design approach and decisions taken for the PPU. We also discuss how the designs were tested and note the progress in the work. We also present some observations that were made which played a role in the design decisions. We believe this work will be extended in future. Rest of the section discusses possible extensions based on current understanding.

7.1 Posit Arithmetic Accelerator

As mentioned in earlier sections one method to add RISC-V support for posits is to utilize the custom opcode space. This method can be used to create an accelerator/co-processor system where when a certain custom instruction is encountered the processor asks the accelerator to perform the required operation. Rocket chip provides such an interface called Rocket Custom Co-processor (or RoCC) interface [20, 21] with its own custom instruction format.

This interface can be utilized to create a Posit Arithmetic accelerator which can provide complete posit functionality. As the posit arithmetic modules have been packaged separately these can be included in the design of the accelerator as a Chisel library. This is the most feasible approach which will allow for IEEE-754 floats and posits to co-exist in the same architecture.

7.2 RISC-V Toolchain Support

Open source compiler toolchains such as GCC and LLVM have support for extensions and plugins to add features without altering the core components [14, 25]. We are interested in adding posit support to these toolchains as extensions.

ACKNOWLEDGEMENTS

The authors acknowledge the contributions by **Bhimsen Padalkar** in this work.

REFERENCES

[1] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. The rocket chip generator. Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012* (2012), IEEE, pp. 1212–1221.

[3] Bandara, S., Ehret, A., Kava, D., and Kinsy, M. A. Brisc-v: An open-source architecture design space exploration toolbox. *Proceedings of the 2019 ACM/SIGDA International Symposium on F ield-Programmable Gate Arrays* (2019).

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

[5] Campbell, K. A., He, L., Yang, L., Gurumani, S. T., Rupnow, K., and Chen, D. Debugging and verifying soc designs through effective cross-layer hardware-software co-simulation. In *DAC '16* (2016).

[6] Chaurasiya, R., Gustafson, J., Shrestha, R., Neudorfer, J., Nambiar, S., Niyogi, K., Merchant, F., and Leupers, R. Parameterized posit arithmetic hardware generator. *2018 IEEE 36th International Conference on Computer Design (ICCD)* (2018), 334–341.

[7] Dinechin, F. D., Forget, L., Muller, J.-M., and Uguen, Y. Posits: the good, the bad and the ugly. In *CoNGA'19* (2019).

[8] FREECHIPSPROJECT. "Chisel / FIRRTL Diagramming Project". https://github.com/freechipsproject/diagrammer.

[9] Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., and Juzgado, N. J. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering 43* (2017), 597–614.

[10] Gala, N., Menon, A., Bodduna, R., Madhusudan, G., and Kamakoti, V. Shakti processors: An open-source hardware initiative. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)* (2016), IEEE, pp. 7–8.

[11] Gustafson, J. L. "RISC-V proposed extension for 32-bit Posits(unofficial)", Jun 2018. https://posithub.org/docs/RISC-V/RISC-V.htm.

[12] Gustafson, J. L., and Yonemoto, I. T. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations 4*, 2 (2017), 71–86.

[13] Hofmann, J., Fey, D., Riedmann, M., Eitzinger, J., Hager, G., and Wellein, G. Performance analysis of the kahan-enhanced scalar product on current multicore processors. In *PPAM* (2015).

[14] Huang, Y., Peng, L., Wu, C., Kashnikov, Y., Rennecke, J., and Fursin, G. Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In *2nd International Workshop on GCC Research Opportunities (GROW'10)* (Pisa, Italy, Jan. 2010).

[15] Izraelevitz, A., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., Kim, D., Schmidt, C., Markley, C., Lawson, J., and Bachrach, J. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (Nov 2017), pp. 209–216.

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

[17] Jaiswal, M. K., and So, H. K. . Pacogen: A hardware posit arithmetic core generator. *IEEE Access 7* (2019), 74586–74601.

[18] Leong, C. "Softposit C library". https://gitlab.com/cerlane/SoftPosit.

[19] Lindstrom, P., Lloyd, S., and Hittinger, J. Universal coding of the reals: alternatives to ieee floating point. In *CoNGA '18* (2018).

[20] Mao, H. "Building custom SoCs with RocketChip", 2017. https://aspire.eecs.berkeley.edu/wiki/_media/eop/2017/howard_mao_talk_slides.pdf.

[21] Panades, I. M. Design and programming of a coprocessor for a risc-v architecture. In *Collegio di Ingegneria Informatica, del Cinema e Meccatronica, Master degree course in Computer Engineering* (2017).

[22] Stillwater-Sc. "Universal: a C++ template library for universal number arithmetic", Apr 2020. https://github.com/stillwater-sc/universal.

[23] Tiwari, S., Gala, N., Rebeiro, C., and Kamakoti, V. PERI: A posit enabled risc-v core, 2019.

[24] Uguen, Y., Forget, L., and de Dinechin, F. Evaluating the hardware cost of the posit number system. In *FPL 2019 - 29th International Conference on Field-Programmable Logic and Applications (FPL)* (Barcelona, Spain, Sept. 2019), pp. 106 – 113.

[25] Vitovská, M., Chalupa, M., and Strejcek, J. Sbt-instrumentation: A tool for configurable instrumentation of llvm bitcode. *ArXiv abs/1810.12617* (2018).

[26] Waterman, A., Lee, Y., Patterson, D. A., and Asanovic, K. The risc-v instruction set manual, volume i: Base user-level isa version 2.0. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2014-54* (2014).

[27] Zaruba, F., and Benini, L. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 27*, 11 (Nov 2019), 2629–2640.