IEEE TRANSACTIONS ON
EMERGING TOPICS
IN COMPUTING

Received 1 December 2021; revised 20 June 2022; accepted 24 June 2022.
Date of publication 6 July 2022; date of current version 6 September 2022.

Digital Object Identifier 10.1109/TETC.2022.3187199

PERCIVAL: Open-Source Posit RISC-V Core With Quire Capability

DAVID MALLASÉN, RAUL MURILLO, ALBERTO A. DEL BARRIO, (Senior Member, IEEE), GUILLERMO BOTELLA, (Senior Member, IEEE), LUIS PIÑUEL, AND MANUEL PRIETO-MATIAS,

David Mallasén and Alberto A. Del Barrio are with the Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain
Raul Murillo and Luis Piñuel are with the Facultad de Ciencias Físicas, Universidad Complutense de Madrid, 28040 Madrid, Spain
Guillermo Botella and Manuel Prieto-Matias are with the Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain, and also with the Instituto de Tecnología del Conocimiento, Universidad Complutense de Madrid, 28040 Madrid, Spain
CORRESPONDING AUTHOR: DAVID MALLASÉN

This work was supported in part by the 2020 Leonardo Grant for Researchers and Cultural Creators, from BBVA Foundation under Grant PR2003_20/01, and in part by the Spanish MINECO, the EU(FEDER), and Comunidad de Madrid under Grants RTI2018-093684-B-I00 and S2018/TCS-4423.

ABSTRACT The posit representation for real numbers is an alternative to the ubiquitous IEEE 754 floating-point standard. In this work, we present PERCIVAL, an application-level posit RISC-V core based on CVA6 that can execute all posit instructions, including the quire fused operations. This solves the obstacle encountered by previous works, which only included partial posit support or which had to emulate posits in software. In addition, Xposit, a RISC-V extension for posit instructions is incorporated into LLVM. Therefore, PERCIVAL is the first work that integrates the complete posit instruction set in hardware. These elements allow for the native execution of posit instructions as well as the standard floating-point ones, further permitting the comparison of these representations. FPGA and ASIC synthesis show the hardware cost of implementing 32-bit posits and highlight the significant overhead of including a quire accumulator. However, results show that the quire enables a more accurate execution of dot products. In general matrix multiplications, the accuracy error is reduced up to 4 orders of magnitude. Furthermore, performance comparisons show that these accuracy improvements do not hinder their execution, as posits run as fast as single-precision floats and exhibit better timing than double-precision floats, thus potentially providing an alternative representation.

INDEX TERMS Arithmetic, posit, IEEE-754, floating point, RISC-V, CPU, CVA6, LLVM, matrix multiplication

I. INTRODUCTION

Representing real numbers and executing arithmetic operations on them in a microprocessor presents unique challenges. When comparing with the simpler set of integers, working with reals introduces notions such as their precision. The representation of real numbers in virtually all computers for decades has followed the IEEE 754 standard for floating-point arithmetic [1]. However, this standard has some flaws such as rounding and reproducibility issues, signed zero, or excess of Not a Number (NaN) representations.

To face these challenges, alternative real number representations are proposed in the literature. Posits [2] are a promising substitute proposed in 2017 that provide compelling benefits. They deliver a good trade-off between dynamic range and accuracy, encounter fewer exceptions when operating, and have tapered precision. This means that numbers near ± 1 have more precision, while very big and very small numbers have less. The posit standard includes fused operations, which can be used to compute a series of multiplications and accumulations without intermediate rounding. Furthermore, posits are consistent between implementations, as they use a single rounding scheme and include only two special cases: single 0 and $\pm\infty$. Therefore, they potentially simplify the hardware implementation [3]. Nonetheless, posits are still under development, and it is still not clear whether they could completely replace IEEE floats [4].

Including Posit Arithmetic Units (PAUs) into cores in hardware is a crucial step to study the efficiency of this representation further. When designing such a core and its arithmetic operations, an important decision is which Instruction Set Architecture (ISA) to implement. RISC-V [5] is a promising open-source ISA that is getting significant attraction both in academia and in industry. Thanks to its openness and flexibility, multiple RISC-V

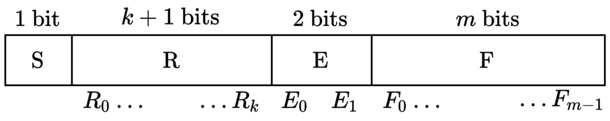


FIGURE 1. **Posit format with sign, regime, exponent and fraction fields.**

cores have been developed targeting diverse purposes in recent years. In the case of studying the performance of posits, a core that can run application-level software is needed.

Some works have studied the use of posits by emulating their execution in software [6]–[8]. However, this approach has the significant drawback of requiring excessive execution times, thus limiting the scalability of the applications.

To overcome these limitations, we propose to include native posit and quire support in hardware by leveraging a high-performance RISC-V core. A comparison of four of the leading open-source application-class RISC-V cores is studied in [9], CVA6 among them. In this work, we have extended the datapath of the CVA6 [10] RISC-V core with a 32-bit PAU with quire and a posit register file. Together with the Xposit compiler extension, this core allows the native hardware execution of high-level applications that leverage the posit number system.

Therefore, the main contributions of this paper are the following:

- We present PERCIVAL, an oPEn-souRCe[1] posIt risc-V core with quire cApabiLity based on the CVA6 that can execute all 32-bit posit instructions, including the quire fused operations.
- Compiler support for the Xposit RISC-V extension in LLVM. This allows to easily embed posit instructions into a C program that can be run natively on PERCIVAL or any other core that implements these opcodes.
- To the best of our knowledge, the PERCIVAL core together with the Xposit extension is the first work that integrates in hardware standard posit addition, subtraction, and multiplication together with quire fused operations. It also includes posit logarithmic-approximate hardware for division and square root operations. Furthermore, all comparison operations and conversions to and from integer numbers are also included in PERCIVAL.
- Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) synthesis results showcasing the resource-usage of posit arithmetic and quire capabilities on a RISC-V CPU. These results are compared with the native IEEE 754 Floating-Point Unit FPU available in the CVA6 and with previous works.
- Accuracy and timing performance of posit numbers and IEEE 754 floats are compared on PERCIVAL using General Matrix Multiplication (GEMM) and max-pooling benchmarks. Results show that 32-bit posits can be up to 4 orders of magnitude more accurate than 32-bit floats thanks to the quire register. Furthermore, this improvement does not imply a trade-off in execution time, as they can perform as fast as 32-bit floats, and thus execute faster than 64-bit floats.

[1]https://github.com/artecs-group/PERCIVAL

The rest of the paper is organized as follows: Section II introduces the necessary background about the posit format, the RISC-V ISA and the CVA6 RISC-V core. Related works from the literature are surveyed in Section III, both as standalone PAUs and at the core level. In Section IV the PERCIVAL posit core is described and in Section V the necessary compiler support for the Xposit RISC-V extension is introduced. The FPGA and ASIC synthesis results of the core are presented, as well as compared with other implementations, in Section VI. Subsequently, in Section VII posits and IEEE 754 floats are compared regarding accuracy and timing performance. Finally, Section VIII concludes this work.

II. BACKGROUND

A. POSIT FORMAT

Posit numbers [2] were introduced in 2017 as an alternative to the predominant IEEE 754 floating-point standard to represent and operate with real numbers. Posits provide reproducible results across platforms and few special cases. Furthermore, they do not support overflow or underflow, which reduces the complexity of exception handling.

A posit number configuration is defined using two parameters as Posit$\langle n, es\rangle$, where n is the total bit-width, and es is the maximum bit-width of the exponent. Although in literature [4], [6], [11] the most widespread posit formats have been Posit$\langle 8,0\rangle$, Posit$\langle 16,1\rangle$ and Posit$\langle 32,2\rangle$, in the latest Posit Standard 4.12 Draft [12], the value of es is fixed to 2. This has the advantage of simplifying the hardware design and facilitates the conversion between different posit sizes.

Posits only distinguish two special cases: zero and Not-a-Real (NaR), which are represented as $0 \cdots 0$ and $10 \cdots 0$ respectively. The rest of the representations are composed of four fields as shown in Figure 1:

- The sign bit S;
- The variable-length regime field R, consisting of k bits equal to R_0 followed by $\overline{R_0}$ or the end of the posit. This field encodes a scaling factor r given by Equation (1);
- The exponent E, consisting of at most es bits, which encodes an integer unbiased value e. If any of its bits are located after the least significant bit of the posit, that bit will have value 0;
- The variable-length fraction field F, formed by the remaining m bits. Its value $0 \leq f < 1$ is given by dividing the unsigned integer F by 2^m.

$$r = \begin{cases} -k & \text{if } R_0 = 0 \\ k-1 & \text{if } R_0 = 1 \end{cases} \tag{1}$$

The real value p of a generic posit is given by Equation (2). The main differences with the IEEE 754 floating-point format are the existence of the regime field, the use of an unbiased exponent, and the value of the fraction hidden bit. Usually, in floating-point arithmetic, the hidden bit is considered to be 1. However, in the latest representation of posits, it is considered to be 1 if the number is positive, or -2 if the number is negative. This simplifies the decoding stage of the posit representation [3], [13].

$$p = ((1 - 3\ s) + f) \times 2^{(1-2\ s)\times(4r+e+s)} \qquad (2)$$

In posit arithmetic, NaR has a unique representation that maps to the most negative 2's complement signed integer. Consequently, if used in comparison operations, it results in less than all other posits and equal to itself. Moreover, the rest of the posit values follow the same ordering as their corresponding bit representations. These characteristics allow posit numbers to be compared as if they were 2's complement signed integers, eliminating additional hardware for posit comparison operations.

The variable-length regime field acts as a long-range dynamic exponent, as can be seen in Equation (2), where it is multiplied by 4 or, equivalently, shifted left by the two exponent bits. Since it is a dynamic field, it can occupy more bits to represent larger numbers or leave more bits to the fraction field when looking for accuracy in the neighborhoods of ± 1. However, detecting these variable-sized fields adds some hardware overhead.

As an example, let `11101010` be the binary encoding of a Posit8, i.e. a Posit$\langle 8, 2\rangle$ according to the latest Posit Standard 4.12 Draft [12]. The first bit $s = 1$ indicates a negative number. The regime field `110` gives $k = 2$ and therefore $r = 1$. The next two bits `10` represent the exponent $e = 2$. Finally, the remaining $m = 2$ bits, `10`, encode a fraction value of $f = 2/2^2 = 0.5$. Hence, from (2) we conclude that $\texttt{11101010} \equiv (-2 + 0.5) \times 2^{-(4+2+1)} = -0.01171875$.

In addition to the standard representation, posits include fused operations using the *quire*, a $16n$-bit fixed-point 2's complement register, where n is the posit bit-width. This allows to execute up to $2^{31} - 1$ Multiply-Accumulate (MAC) operations without intermediate rounding or accuracy loss. The quire can represent either NaR, similarly to regular posits, or the value given by 2^{16-8n} times the 2's complement signed integer represented by the $16n$ concatenated bits.

B. RISC-V ISA

The open-source RISC-V ISA [5] emanates from the ideas of RISC. It is structured as a base integer ISA plus a set of optional standard and non-standard extensions to customize and specialize the final set of instructions. There are two main base integer ISA, RV32I and RV64I, that establish the user address spaces as 32-bit or 64-bit respectively.

The RISC-V general standard extensions include, among others, functionality for integer multiply/divide (M), atomic memory operations (A) and single- (F) and double-precision (D) floating-point arithmetic following the IEEE 754 standard. This set of general-purpose standard extensions IMAFD, together with the instruction-fetch fence (Zifencei), and the control and status register (Zicsr), form the general-purpose G abbreviation. In general, following the Reduced Instruction Set Computer (RISC) principles, all extensions have fixed-length 32-bit instructions. However, there is also a compressed instruction extension (C) that provides 16-bit instructions.

Expanding the RISC-V ISA with specialized extensions is supported by the standard to allow for customized accelerators. Non-standard extensions can be added to the encoding space leveraging the four major opcodes reserved for custom extensions. A proposal of the changes that should be made to the F standard extension in order to have a 32-bit posit RISC-V extension is described in [14].

C. CVA6

The CVA6 [10] (formerly known as Ariane) is a 6-stage, in-order, single-issue CPU which implements the RV64GC RISC-V standard. The core implements three privilege levels and can run a Linux operating system. The primary goal of its micro-architecture is to reduce the critical path length. It was developed initially as part of the PULP ecosystem, but it is currently maintained by the OpenHW Group, which is developing a complete, industrial-grade pre-silicon verification. CVA6 is written in SystemVerilog and is licensed under an open-source Solderpad Hardware License.

As execution units in the datapath it includes an integer ALU, a multiply/divide unit and an IEEE 754 FPU [15]. This FPU claims to be IEEE 754-2008 compliant, except for some issues in the division and square root operations. For the sake of comparison, it is important that the FPU is IEEE 754 compliant instead of being limited to normal floats only, since in theory, posit hardware is slightly more expensive than floating-point hardware that does not take into account subnormal numbers [3].

III. RELATED WORK

There has been a great deal of interest in hardware implementations of posit arithmetic since its first appearance. Standalone PAUs with different degrees of capabilities or basic posit functional units have been described in the literature [11], [16]–[18]. These units provide the building blocks to execute posit arithmetic. However, they do not allow themselves to execute whole posit algorithms.

Recently, some works adding partial posit support to RISC-V cores have been presented. CLARINET [19] incorporates the quire into a RV64GC 5-stage in-order core. However, not all posit capabilities are included in this work. Most operations are performed in IEEE floating-point format, and the values are converted to posit when using the quire. The only posit functionalities added to the core are fused MAC with quire, fused divide and accumulate with quire and conversion instructions.

PERC [20] integrates a PAU into the Rocket Chip generator, replacing the 32 and 64-bit FPU. However, this work does not include quire support, as it is constrained by the F and D RISC-V extensions for IEEE-754 floating-point

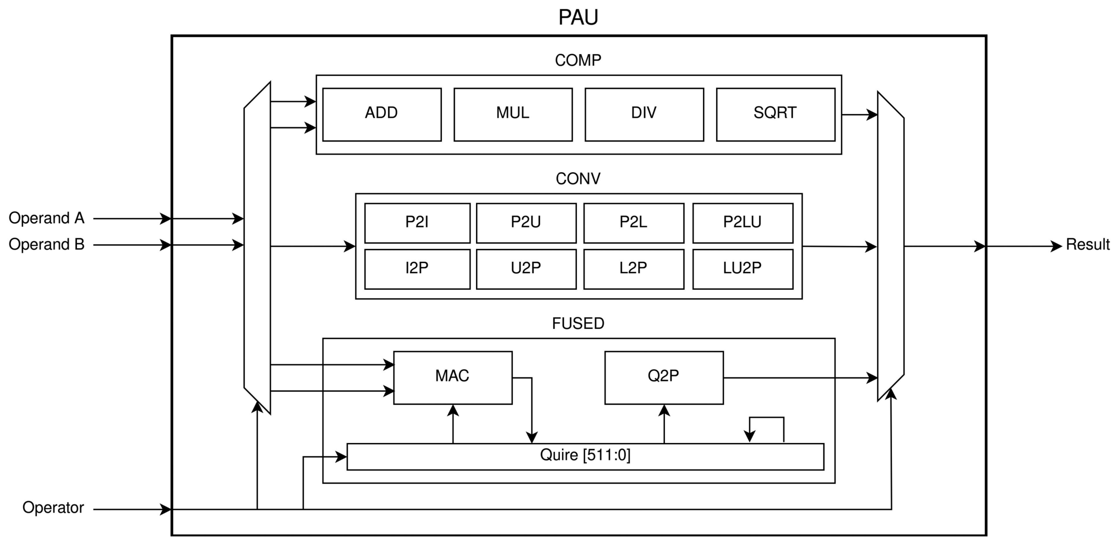


FIGURE 2. **Internal structure of the proposed Posit Arithmetic Unit (PAU).**

numbers. More recently, PERI [21] added a tightly coupled PAU into the SHAKTI C-class core, a 5-stage in-order RV32IMAFC core. This proposal also does not include quire support as it reuses the F extension instructions. Nonetheless, it allows dynamic switching between es=2 and es=3 posits. In [22] authors include a PAU named POSAR into a RISC-V Rocket Chip core. Again, this proposal does not include quire support and replaces the FPU present in Rocket Chip to reuse the floating-point instructions.

A different approach is taken in [23], where authors use the posit representation as a way to store IEEE floats in memory with a lower bit-width while performing the computations using the IEEE FPU. For this purpose they include a light posit processing unit into the CVA6 core that converts between 8 or 16-bit posits and 32-bit IEEE floats. They also develop an extension of the RISC-V ISA to include these conversion instructions.

IV. PERCIVAL POSIT CORE

In this work, we have integrated full posit capabilities, including quire and fused operations, into an application-level RISC-V core. In addition to the design of the functional units that execute the posit and quire operations, the novelty of our design is that it is fully compatible both at the software and hardware level with the F and D RISC-V extensions. Therefore, both posit and IEEE floating-point numbers can be used simultaneously on the same core. This is the first work that integrates practically all of the posit and quire operations specified in the posit standard into a core, to the best of our knowledge.

A. PAU DESIGN

The Posit Arithmetic Unit (PAU) is in charge of executing most posit operations and also contains the quire register, as shown in Figure 2. Posit comparisons are executed in the integer Arithmetic Logic Unit (ALU). As mentioned above, this is one of the benefits of the posit representation.

When designing the micro-architecture of the PAU, our objective was to achieve a similar latency and throughput as the FPU operations, to obtain fair comparisons. The throughput is limited, as there is no pipeline in the FPU nor the PAU. Nevertheless, all of the operations are multi-cycle. The latency of the PAU units is the following:

- PADD, PSUB, QMADD, and QMSUB: 2 cycles.
- PMUL, PDIV, PSQRT, and QROUND: 1 cycle.

All other operations have no latency, i.e. they output their result at the next clock cycle after receiving the inputs.

As a comparison, the 32-bit FADD, FSUB, FMADD, FMSUB, and FMUL instructions in the FPU have a latency of 2 clock cycles, but the 64-bit analogous instructions have a latency of 3 cycles. It is noteworthy that the comparisons in the FPU have a latency of 1, while the posit comparisons that reuse the integer hardware have no latency. Conversions to and from integer values also take an extra clock cycle in the FPU.

Depending on the operation, the input operands are directed to the corresponding posit unit and the result is forwarded as an output of the PAU. There are three main blocks: computational operations (COMP), conversion operations (CONV), and operations that make use of the quire register (FUSED) (Figure 2).

Regarding COMP, the ADD unit is used both for addition and subtraction, calculating the two's complement of the second operand when subtracting. In this group, all the modules use both operands except the square root, which uses only operand A. In addition, the operands and the result correspond to the posit register file.

It must be noted that the posit division and square root units are approximate, as this type of arithmetic simplifies the designs and thus reduces the hardware cost of the system. They are logarithm-approximate units based on Mitchell's Approximate Log Multipliers and our previous work [11]. These units have been demonstrated to have a maximum relative error of 11.11%, and have less impact on area/performance than the exact hardware operators. On the other hand,

```
Require: Instruction to decode instr.
Ensure: Scoreboard entry sc_instr which contains the
  operation op and the destination functional unit fu.
  switch (instr.opcode)
  ...
  case POSIT:
    switch (instr.func3)
    case 000: {Computational posit instruction}
      switch (instr.func5)
      case 00000: {PAU instruction}
        sc_instr.fu = PAU
        sc_instr.op = PADD
      ...
      case 00100: {ALU instruction}
        sc_instr.fu = ALU
        sc_instr.op = PMIN
      ...
      end switch
    case 001: {Posit load instruction}
      sc_instr.fu = LOAD
      sc_instr.op = PLW
    case 011: {Posit store instruction}
      sc_instr.fu = STORE
      sc_instr.op = PSW
    end switch
  ...
  default: {Instruction not decoded in any switch/case}
    illegal_instr = true
  end switch
```

FIGURE 3. Pseudocode describing the decoding of posit instructions.

exact division and square root algorithms could be implemented in software leveraging the MAC unit, thus eliminating the need for dedicated hardware. However, this is out of the scope of this work.

In the CONV group, only operand A is used for conversions. Depending on the operation, the input data and the result belong to the posit or the integer register file.

The quire register is the most singular addition to this number format. According to the posit standard, it must be an architectural register accessible by the programmer that is also allowed to be dumped into memory. However, being so wide, the cost of including this functionality into the core's datapath could be too high for the benefits it would add. In the vast majority of cases, the quire is used as an accumulator to avoid overflows in the MAC operations, and this does not require quire load and store operations. Instead, we can initialize the quire to zero (QCLR.S), negate it if needed (QNEG.S), accumulate the partial products in it without rounding or storing in memory (QMADD.S and QMSUB.S), and, when the whole operation is finished, round and output the result (QROUND.S). The necessary support for all of these operations related to the quire is included in our proposal (see Table II below). The hardware cost of including the quire as an internal register in the PAU is studied in Section VI.

B. CORE INTEGRATION

The proposed PAU has been integrated into the CVA6 RV64GC core while maintaining the compatibility with all existing extensions, including single- and double-precision floating point. Moreover, since we work with Posit32 numbers, i.e. Posit$\langle 32, 2\rangle$, the core adds a 32-bit posit register file in addition to the integer and floating-point registers.

The instruction decoder has been extended to support posit instructions. The inner workings of the decoder are described in Figure 3. As part of the decoding process, each posit instruction selects from which register file it must obtain its operands and to which register file it must forward its result.

The CVA6 core uses scoreboarding for dynamically scheduled instructions and allows out-of-order write-back of each functional unit. The scoreboard tracks which instructions are issued, their functional unit and in which register they will write back to. Our design has enlarged the scoreboard to include posit registers and instructions. In this manner, we can discern whether the input data of posit operations are retrieved from a register or forwarded directly as a result of a previous operation.

As mentioned in Section II-A, posit numbers have the benefit of being able to reuse the comparison hardware of 2's complement signed integers. Therefore, the integer ALU has also been extended to accept posit operands and to be able to forward the result of these instructions with minimal hardware overhead. Furthermore, the PAU has been integrated into the execution phase of the processor in parallel to the ALU and the FPU, connecting the issue module with the aforementioned scoreboard. Finally, the complete datapath has been adapted to include the posit signals and all necessary additional interconnections.

V. COMPILER SUPPORT: XPOSIT EXTENSION

The assembly output of a RISC-V compiler when processing programs that use floating-point arithmetic includes instructions from the corresponding F and D extensions. To produce a similar output but targeting posit numbers, a new extension must be introduced that translates posit instructions and posit operators to binary code. Therefore, in this section, the Xposit RISC-V extension targeting posit arithmetic is presented. As part of this work, Xposit has been integrated into LLVM 12 backend [24] to allow the compilation of high-level applications.

This modified version of LLVM can compile C code. However, posit instructions must be written from the assembly level, as there is currently no support for writing posit or quire operations directly in C. Therefore, previous codes can be reused in PERCIVAL, and only the computational kernels have to be manually written in assembly. An example of this is shown in Section VII.

The posit instruction set follows the structure of the F RISC-V standard extension for single-precision floating point [25]. This Xposit extension mostly follows the adaptation to the posit

TABLE I. RISC-V Base Opcode Map + POSIT Extension; inst[1:0]=11.

inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								$(> 32b)$
00	LOAD	LOAD-FP	POSIT	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	reserved	*custom-3/rv128*	$\geq 80b$

format proposed in [14]. The differences with this proposal are the following:

- We include 32 posit registers `p0-31` as in the F standard extension.
- Similarly to the integer operations in CVA6, there is no flag signaling division by zero.
- We do not include the possibility of loading and storing the quire in memory.

The Xposit extension uses the 0001011 opcode (*custom-0*), occupying the space indicated in Table I as POSIT. If more operations were needed in the future, especially posit load and store instructions of other word lengths, the 0101011, 1011011, and 1111011 opcodes (*custom-1,2,3*) could be leveraged. In this way, a similar approach as the F and D RISC-V extensions could be followed, which utilize the OP-FP, LOAD-FP and STORE-FP opcodes.

The format and fields of the Xposit instructions are described in Figure 4. Posit load and store use the same base+offset addressing as the corresponding floating-point instructions, with the base address in register *rs1* and a signed 12-bit byte offset. Thus, the PLW instruction loads a posit value from memory to the *rd* posit register and the PSW instruction stores a posit value from the *rs2* posit register to memory. The rest of the Xposit operations keep the POSIT opcode and differ from the previous instructions by the *funct3* field. Finally, it must be noted that the *fmt* field is fixed to `01` indicating that the instructions are for single-precision (32-bit) posits. The complete instruction set of the proposed Xposit RISC-V extension is detailed in Table II.

An important addition of the Xposit extension are the quire instructions. Since the quire is a single internal register of the PAU, the instructions that operate with it do not have to specify a quire register number. For example, the quire clear instruction does not have any parameters. It is decoded and then executed internally by the PAU, which simply sets the quire register to 0. The quire fused operations only have to specify the posit registers of the two values that will be multiplied. Then, the accumulation is performed implicitly on the quire.

VI. SYNTHESIS RESULTS

In this section, we present the FPGA and ASIC synthesis results of PERCIVAL. The details of its PAU and the IEEE 754 FPU using 32 and 64-bit formats are also included. In this manner, the hardware cost of posit numbers and the quire are highlighted and compared with other implementations.

A. FPGA SYNTHESIS

The FPGA synthesis was performed using Vivado v.2020.2 targeting a Genesys II (Xilinx Kintex-7 XC7K325T-2FFG900C) FPGA. Different configurations of FPU and PAU were tested, the results of which are shown in Table III. Since the critical path does not traverse the arithmetic units of the core, in all of the cases the timing constraint of 20 ns was met and the timing slack was +0.177 ns.

The bare CVA6 without a FPU or PAU requires 28950 Lookup Tables (LUTs) and 19579 Flip-flops (FFs). Including support for 32-bit floating-point numbers increases the number of LUTs and FFs by 6452 and 2039 respectively. This difference grows to 12310 LUTs and 4366 FFs when using also the double-precision D extension. Note that these values are larger than simply the FPU area, since they also include other elements such as the floating-point register file, instruction decoding and interconnections. These other non-FPU elements require 2406 LUTs and 1066 FFs in the 32-bit case and 4147 LUTs and 2122 FFs in the 64-bit case.

Comparing the overall cost of including posit support with the cost of including IEEE floating-point support, a significant difference can be seen. Adding 32-bit posit operations and quire support to the CVA6 requires 15743 LUTs and 4057 FFs, which is comparable to the FD floating-point configuration. Out of this area, 3864 LUTs and 1072 FFs are occupied by the non-PAU elements mentioned in the previous floating-point analysis.

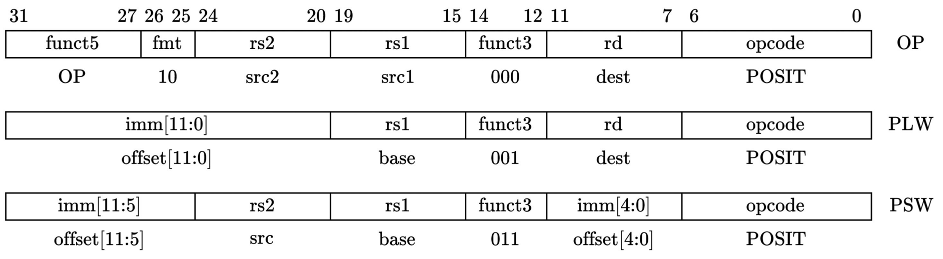


FIGURE 4. Internal structure and fields of Xposit instructions.

TABLE II. **Instruction Set of the Proposed XPosit RISC-V Extension.**

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	001	rd	00001011	PLW
imm[11:5]		rs2	rs1	011	imm[4:0]	00001011	PSW
00000	10	rs2	rs1	000	rd	00001011	PADD.S
00001	10	rs2	rs1	000	rd	00001011	PSUB.S
00010	10	rs2	rs1	000	rd	00001011	PMUL.S
00011	10	rs2	rs1	000	rd	00001011	PDIV.S
00100	10	rs2	rs1	000	rd	00001011	PMIN.S
00101	10	rs2	rs1	000	rd	00001011	PMAX.S
00110	10	00000	rs1	000	rd	00001011	PSQRT.S
00111	10	rs2	rs1	000	00000	00001011	QMADD.S
01000	10	rs2	rs1	000	00000	00001011	QMSUB.S
01001	10	00000	00000	000	00000	00001011	QCLR.S
01010	10	00000	00000	000	00000	00001011	QNEG.S
01011	10	00000	00000	000	rd	00001011	QROUND.S
01100	10	00000	rs1	000	rd	00001011	PCVT.W.S
01101	10	00000	rs1	000	rd	00001011	PCVT.WU.S
01110	10	00000	rs1	000	rd	00001011	PCVT.L.S
01111	10	00000	rs1	000	rd	00001011	PCVT.LU.S
10000	10	00000	rs1	000	rd	00001011	PCVT.S.W
10001	10	00000	rs1	000	rd	00001011	PCVT.S.WU
10010	10	00000	rs1	000	rd	00001011	PCVT.S.L
10011	10	00000	rs1	000	rd	00001011	PCVT.S.LU
10100	10	rs2	rs1	000	rd	00001011	PSGNJ.S
10101	10	rs2	rs1	000	rd	00001011	PSGNJN.S
10110	10	rs2	rs1	000	rd	00001011	PSGNJX.S
10111	10	00000	rs1	000	rd	00001011	PMV.X.W
11000	10	00000	rs1	000	rd	00001011	PMV.W.X
11001	10	rs2	rs1	000	rd	00001011	PEQ.S
11010	10	rs2	rs1	000	rd	00001011	PLT.S
11011	10	rs2	rs1	000	rd	00001011	PLE.S

The synthesis results reveal that the PAU requires significantly more resources than the FPU available in the CVA6. In particular, the 32-bit PAU with quire occupies 2.94 times as many LUTs and 3.07 times as many FFs as the 32-bit FPU. To better understand these results, in Table IV the area requirements of the different modules inside the PAU are presented. The most interesting value shown in this table is the area occupied by the posit MAC unit, which corresponds to almost half of the total area of the PAU.

When compared with the floating-point units, which do not include an accumulation register, the area requirements of the quire could be separated. Thus, the posit MAC and the quire rounding to posit can be subtracted from the total PAU area to obtain a value of 5326 LUTs and 1312 FFs. This outcome is now much closer to the synthesis results of the FPU, as the PAU without quire occupies 1.32 times as many LUTs and 1.35 times as many FFs. These results match previous works [22], where authors also report an increase of around 30% in FPGA resources when comparing their 32-bit PAU without quire with a 32-bit FPU.

In our case, the actual value of not including a quire would be even smaller, as the cost of allocating the 512-bit quire in the PAU and computing its 2's complement, which are included in the PAU top, should also be subtracted. However, the synthesis tool does not include these details.

B. ASIC SYNTHESIS

The 32-bit PAU with quire and the 32-bit FPU configuration present in PERCIVAL were synthesized targeting TSMC's 45 nm standard-cell library to further study their hardware cost in ASIC. The synthesis was performed using Synopsys Design Compiler with a timing constraint of 5 ns, which was met in both cases, and a toggle rate of 0.1.

The 32-bit FPU within CVA6 requires an area of 30691 μm^2 and consumes 27.26 mW of power. On the other hand, the 32-bit PAU with quire requires an area of 76970 μm^2 and

TABLE III. **Comparison of FPGA Synthesis Results With Different Configurations of FPU, Marked as F and D for 32 and 64-Bit Numbers Respectively, and 32-Bit PAU With Quire.**

	PAU				No PAU			
	F	D	FD	-	F	D	FD	-
Total core (LUT, FF)	(50318, 25727)	(55900, 27652)	(57129, 27996)	(44693, 23636)	(35402, 21618)	(40740, 23599)	(41260, 23945)	(28950, 19579)
FPU area (LUT, FF)	(3726, 1008)	(6352, 1905)	(7612, 2245)	-	(4046, 973)	(6626, 1905)	(8163, 2244)	-
PAU area (LUT, FF)	(11796, 2979)	(11810, 2979)	(11803, 2979)	(11879, 2985)	-	-	-	-

TABLE IV. FPGA Synthesis Area Results of the PAU Desegregated Into Its Individual Components.

Name	LUTs	FFs
PAU top	593	1063
Posit Add	784	106
Posit Mult	736	73
Posit ADiv	413	43
Posit ASqrt	426	33
Posit MAC	5644	1541
Quire to Posit	889	126
Int to Posit	176	0
Long to Posit	331	0
ULong to Posit	425	0
Posit to Int	499	0
Posit to Long	379	0
Posit to UInt	228	0
Posit to ULong	358	0
PAU total	11879	2985
PAU w/o quire	5346	1318

TABLE V. ASIC Synthesis Area and Power Results of the 32-Bit PAU With Quire Desegregated Into Its Individual Components.

Name	Area (μm^2)	Power (mW)
PAU top	13462.15	12.69
Posit Add	4075.31	3.59
Posit Mult	8635.37	9.98
Posit ADiv	2540.87	2.41
Posit ASqrt	1722.84	1.61
Posit MAC	30419.12	26.07
Quire to Posit	6026.76	4.04
Int to Posit	905.99	0.68
Long to Posit	1423.43	0.96
UInt to Posit	869.77	0.66
ULong to Posit	1353.11	0.94
Posit to Int	966.67	0.71
Posit to Long	1810.33	1.38
Posit to UInt	958.44	0.68
Posit to ULong	1800.22	1.33
PAU total	76970.38	67.73
PAU w/o quire	40524.62	37.62
CLARINET PAU	69920.02	68.31

consumes 67.73 mW of power. This follows the same trend shown in the FPGA synthesis, as the PAU with quire is significantly larger, 2.51x, and consumes more power, 2.48x, than the FPU.

In addition, to better assess these values in comparison with other proposals, the PAU available in CLARINET [19] was also synthesized with the same parameters. We have chosen to evaluate this work because it integrates, to the best of our knowledge, the only other PAU that contains a quire. In this case, the 32-bit PAU with quire requires an area of 69920 μm^2 and consumes 68.31 mW of power. This is a decrease of around 10% in area and a slight increase in power compared to our proposal, although ours implements a much larger set of posit functionality.

Similarly as in Section VI-A, the area and power results of the different elements inside the PAU are presented in Table V. As can be seen, when subtracting the cost of the quire in the PAU, the outcome is still higher than the 32-bit FPU, but it becomes much closer. The 32-bit PAU occupies 1.32 times as much area and consumes 1.38 times as much power as the 32-bit IEEE FPU FPNew [15]. However, it is noteworthy that some aspects of posit arithmetic are not yet fully studied. For example, most of the works presenting posit units have tackled the decoding and encoding phases using sign-magnitude. Nonetheless, more recent studies show that a 2's complement approach is more efficient [13].

VII. POSIT VS IEEE-754 BENCHMARKS

One of the benefits of PERCIVAL is that an accurate and fair comparison can be made between posit and IEEE floating point. The main advantage of having support for native posit and IEEE floating point simultaneously on the same core is that identical benchmarks can be run on both number representations to compare them. In this work, we have chosen to benchmark the GEMM and the max-pooling layer, used to down-sample the representation of neural networks. These examples showcase the use of the quire and posits both in the PAU and in the ALU, loading and storing from memory and leveraging the posit register file.

The GEMM and max-pooling codes for posits and IEEE floats have been written in C, including inline assembly for the required posit and float instructions. The floating-point code has also been written in inline assembly to provide exactly the same sequence of instructions to the core. The GEMM code for floats is shown in Figure 5 and the analogous version for posits using the quire is shown in Figure 6. These codes have been compiled using the modified version of LLVM with the Xposit RISC-V extension as specified in Section V, and serve as an example of how this extension can be leveraged. Therefore, the final target architecture is RV64GCXposit. The `-O2` optimization flag has been used to obtain an optimized code in every case.

A. ACCURACY

The accuracy differences between posits and floats are studied for the GEMM benchmark. Furthermore, each arithmetic is executed with and without using fused MAC operations, which in posit arithmetic include the quire. In the cases without quire or `FMADD`, each fused operation is substituted by a multiplication and an addition. The results obtained using the 64-bit IEEE 754 format are considered the golden solution and used to compute the Mean Squared Error (MSE) of the 32-bit posit and the 32-bit IEEE 754 floating point. In all cases, the inputs are square matrices with the same random values. These input values are generated from a uniform distribution in intervals of the form $[-10^i, 10^i],\ i \in \{-1, 0, 1, 2, 3\}$. This results in 5 different sets of inputs. These intervals allow for a study of the impact of the input data range on the GEMM. These random values are generated as 64-bit IEEE 754 numbers and then converted to the two other formats with the aid of the SoftPosit [26] library.

```
Require: Float matrices a and b of size n×n.
Ensure: Float matrix c = ab.
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      asm("fmv.w.x ft0,zero":::); {Set ft0 to 0}
      for k = 0 to n-1 do
        asm
          "flw      ft1,0(%0)" {Load float a and b}
          "flw      ft2,0(%1)"
          "fmadd.s ft0,ft1,ft2,ft0" {Accumulate on ft0}
          :: "r" (&a[i * n + k]), "r" (&b[k * n + j]):
        end asm
      end for
      asm
        "fsw ft0,0(%1)" {Store the result in c}
        : "=rm" (c[i * n + j]) : "r" (&c[i * n + j]):
      end asm
    end for
  end for
```

FIGURE 5. 32-bit floating-point GEMM using the F RISC-V extension.

```
Require: Posit matrices a and b of size n×n.
Ensure: Posit matrix c = ab.
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      asm("qclr.s":::); {Clear the quire}
      for k = 0 to n-1 do
        asm
          "plw      pt0,0(%0)" {Load posit a and b}
          "plw      pt1,0(%1)"
          "qmadd.s pt0,pt1" {Accumulate on the quire}
          :: "r" (&a[i * n + k]), "r" (&b[k * n + j]):
        end asm
      end for
      asm
        "qround.s pt2" {Round the quire to a posit}
        "psw      pt2,0(%1)" {Store the result in c}
        : "=rm" (c[i * n + j]) : "r" (&c[i * n + j]) :
      end asm
    end for
  end for
```

FIGURE 6. Posit GEMM using the Xposit RISC-V extension with the quire accumulator.

The MSE results are shown in Table VI for different matrix sizes and input ranges. Additionally, Figure 7 shows the MSE in the $[-1, 1]$ case. We decided to give slightly more attention to this case since many applications normalize their values. As can be seen, for 256 × 256 matrices, the difference between MSE is around four orders of magnitude when using fused operations. This is reduced to two orders of magnitude if the quire is not used. Note that when using floats, the accuracy difference between employing fused `FMADD` operations or not is minimal.

If we compare how the MSE scales when increasing the matrix size, it can be seen that posit numbers present a better behavior thanks to the quire register. This is true in all ranges of input values. Overall, the impact of the quire is significant among all test cases, and its extra cost is justified by the results.

These results go in line with our previous work [27], where a similar benchmark was performed using hardware simulations with an input interval of $[-2, 2]$. The MSE results on 32-bit floats and posits follow the same trends given in Table VI.

TABLE VI. GEMM MSE Comparison Between IEEE 754 Floating-Point and Posit Numbers.

Input values	Matrix size	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256
[-0.1, 0.1]	IEEE 754	1.385×10^{-18}	4.429×10^{-18}	1.523×10^{-17}	6.347×10^{-17}	2.407×10^{-16}
	Posit32	3.157×10^{-21}	6.110×10^{-21}	1.158×10^{-20}	2.014×10^{-20}	3.497×10^{-20}
	IEEE 754 no FMADD	1.515×10^{-18}	4.752×10^{-18}	1.566×10^{-17}	6.524×10^{-17}	2.432×10^{-16}
	Posit32 no quire	2.146×10^{-20}	6.726×10^{-20}	2.371×10^{-19}	7.805×10^{-19}	2.203×10^{-18}
[-1, 1]	IEEE 754	1.490×10^{-14}	4.251×10^{-14}	1.602×10^{-13}	6.019×10^{-13}	2.361×10^{-12}
	Posit32	1.138×10^{-17}	2.355×10^{-17}	4.729×10^{-17}	9.430×10^{-17}	1.937×10^{-16}
	IEEE 754 no FMADD	1.324×10^{-14}	4.637×10^{-14}	1.686×10^{-13}	6.246×10^{-13}	2.416×10^{-12}
	Posit32 no quire	5.028×10^{-17}	1.727×10^{-16}	6.457×10^{-16}	2.447×10^{-15}	9.870×10^{-15}
[-10, 10]	IEEE 754	1.371×10^{-10}	3.998×10^{-10}	1.581×10^{-9}	5.922×10^{-9}	2.378×10^{-8}
	Posit32	8.549×10^{-13}	1.475×10^{-12}	3.055×10^{-12}	6.355×10^{-12}	1.295×10^{-11}
	IEEE 754 no FMADD	1.300×10^{-10}	4.304×10^{-10}	1.708×10^{-9}	6.026×10^{-9}	2.447×10^{-8}
	Posit32 no quire	3.878×10^{-12}	1.341×10^{-11}	7.500×10^{-11}	3.282×10^{-10}	1.41×10^{-9}
[-100, 100]	IEEE 754	1.412×10^{-6}	4.206×10^{-6}	1.544×10^{-5}	6.402×10^{-5}	2.405×10^{-4}
	Posit32	4.819×10^{-8}	8.266×10^{-8}	1.760×10^{-7}	6.150×10^{-7}	1.506×10^{-6}
	IEEE 754 no FMADD	1.293×10^{-6}	5.052×10^{-6}	1.595×10^{-5}	6.503×10^{-5}	2.440×10^{-4}
	Posit32 no quire	3.077×10^{-7}	1.230×10^{-6}	4.295×10^{-6}	2.804×10^{-5}	1.569×10^{-4}
[-1000, 1000]	IEEE 754	1.503×10^{-2}	3.936×10^{-2}	1.509×10^{-1}	6.069×10^{-1}	2.391
	Posit32	5.293×10^{-3}	8.573×10^{-3}	1.900×10^{-2}	3.746×10^{-2}	8.265×10^{-2}
	IEEE 754 no FMADD	1.675×10^{-2}	4.815×10^{-2}	1.644×10^{-1}	6.323×10^{-1}	2.433
	Posit32 no quire	4.168×10^{-2}	1.570×10^{-1}	5.669×10^{-1}	2.365	9.586

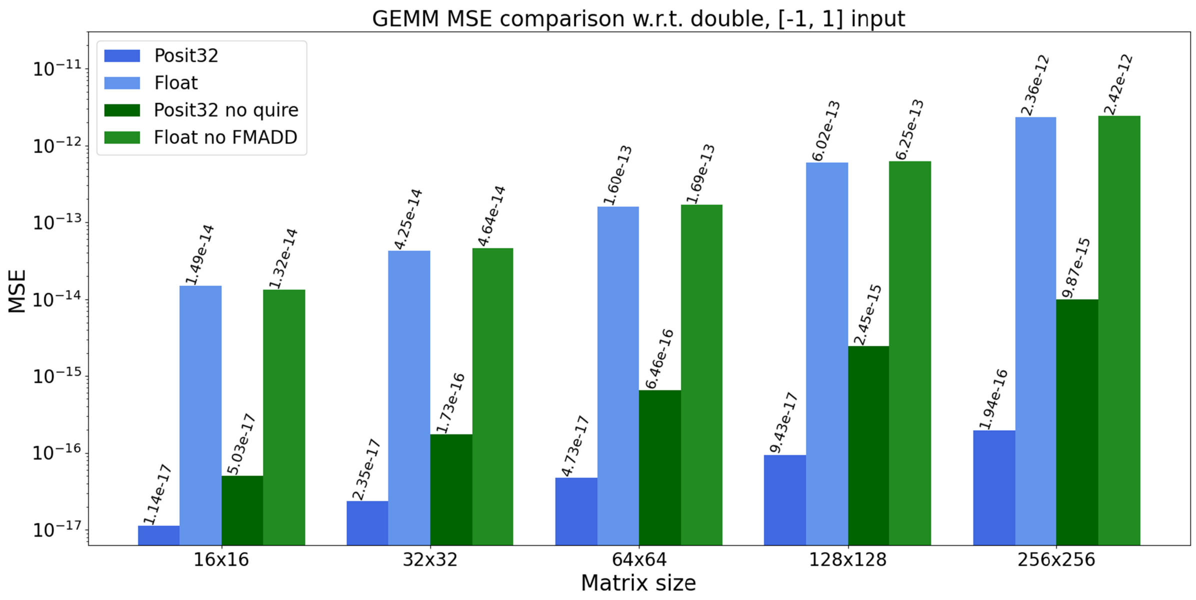


FIGURE 7. MSE results of posits and floats with respect to doubles in the GEMM test with input values in $[-1, 1]$. Note the logarithmic Y-axis. Blue (green) bars show the results with (without) fused MAC and quire operations.

When removing the quire, posits still have a lower MSE than floats except in the $[-1000, 1000]$ case. This can be explained by posit's tapered precision. When the numbers' exponents are closer to 0, they end up in the so-called "golden zone" of posits [4]. This is the area where posits have more accuracy bits than floats thanks to their variable-length fields. However, when the accumulated values are large or very small, IEEE floats gain an advantage over posits without quire.

Particularly, this "golden zone" comprises values roughly in the interval $[10^{-6}, 10^{6}]$. In the test with input values in $[-1000, 1000]$, the absolute value of the final outputs averages 1.2×10^6 in the 16×16 matrix and 4.3×10^6 in the 256×256 case. As a comparison, even in the 256×256 multiplication, the $[-100, 100]$ input range only averages 4.3×10^4.

B. PERFORMANCE

Besides the synthesis data presented in Section VI, the execution time is a critical metric to study the hardware performance of posits and floats. The test has been performed executing the same GEMM and max-pooling described previously on PERCIVAL, avoiding cold misses and averaging over 10 executions to obtain more accurate measurements.

The range of the input values does not affect performance. Thus, the values shown in Table VII for GEMM are an average of the timings obtained in the 5 cases described previously. This gives a total of 50 executions in the GEMM operation. In this case, when using fused MAC operations and the quire, the execution time of 32-bit posits is practically the same as that of single-precision floats for the larger matrix sizes, where the overhead execution of the extra qround.s instruction becomes negligible (see Figure 6). This instruction is executed in the order of $\mathcal{O}(n^2)$ times, compared with the $\mathcal{O}(n^3)$ running time of the algorithm. This cost is noticeable for smaller values of n, when 32-bit posits are slightly slower than 32-bit and 64-bit floats. However, for larger matrix sizes, which are common in scientific applications and Deep Neural Networks (DNNs), 32-bit posits perform equally as 32-bit floats and outperform 64-bit floats, since these instructions require more clock cycles to compute. Furthermore, as seen in the previous accuracy benchmark, 32-bit posits are orders of magnitude more accurate than 32-bit floats when performing this calculation. Therefore, they provide an alternative solution for the execution of kernels that make use of the dot product.

TABLE VII. GEMM Timing Comparison Between IEEE 754 Floating-Point and Posit Numbers.

Matrix size	16×16	32×32	64×64	128×128	256×256
32-bit float	0.978 ms	6.58 ms	52.1 ms	1.48 s	13.9 s
64-bit float	0.920 ms	6.64 ms	69.4 ms	1.74 s	15.0 s
Posit32	0.949 ms	7.30 ms	57.7 ms	1.48 s	13.9 s
32-bit float no FMADD	1.16 ms	8.69 ms	68.6 ms	1.61 s	15.0 s
64-bit float no FMADD	1.26 ms	9.36 ms	92.6 ms	1.92 s	16.7 s
Posit32 no quire	1.27 ms	9.63 ms	69.1 ms	1.61 s	15.0 s
VividSparks Posit32 no quire	7.95 ms	48.9 ms	345 ms	2.63 s	21.1 s

TABLE VIII. Max-Pooling Timing Comparison Between IEEE 754 Floating-Point and Posit Numbers.

Max-pooling layer	32-bit float	64-bit float	Posit32
LeNet-5 (28x28x6)	0.715 ms	1.211 ms	0.688 ms
AlexNet (54x54x96)	0.115 ms	0.160 ms	0.116 ms
ResNet-50 (112x112x64)	0.337 ms	0.470 ms	0.340 ms

The quire and fused MAC operations have a positive impact on timing performance. This is true in all test cases. Again, this performance increase stems from the extra clock cycles needed for a multiplication + an addition in comparison to only one fused operation.

Additionally, for the sake of completeness, we have performed the same GEMM timing test on a commercial core with support for posit arithmetic. RacEr is a GPGPU FPGA provided by VividSparks that supports computation with Posit32 but does not include quire support, so its accuracy results are the same as the Posit32 no quire case. It has 512 CPUs running at 300 MHz with 32 GB of DDR4 RAM. Table VII also includes the results of the GEMM benchmark on this platform. As can be seen, our proposal provides significantly faster results than this commercial accelerator.

Regarding the max-pooling layers, three different configurations have been tested following common DNNs. In LeNet-5, the input of this layer is 28x28x6, the pooling kernel is 2x2 and is applied with a stride of 2, creating a 14x14x6 output representation. In AlexNet, the input size is 54x54x96, the kernel size is 3x3 and is applied with a stride of 2, generating an output of size 26x26x96. Finally, ResNet-50 is the largest configuration we have tested, as its input is 112x112x64, the pooling kernel is 3x3 and again is applied with a stride of 2, creating a 55x55x64 output representation.

The results of executing these layers on PERCIVAL using the 32 and 64-bit IEEE floating-point and Posit32 representations are shown in Table VIII. Results show that 32-bit posits perform as fast as 32-bit floats but without the need for extra hardware, as the posit maximum operation is carried out reusing the integer ALU. Double-precision floats are slower than 32-bit posits and floats by a factor of 1.4-1.7× due to the latency difference in the units as seen in the GEMM benchmark.

VIII. CONCLUSION

This paper has presented PERCIVAL, an extension of the application-level CVA6 RISC-V core, including all 32-bit posit instructions as well as the quire fused operations. These capabilities, integrated into a Posit Arithmetic Unit together with a posit register file, are natively incorporated while preserving IEEE 754 single- and double-precision floats.

Furthermore, the RISC-V ISA has been extended with Xposit, which includes support for all posit and quire instructions. This allows the compilation and execution on PERCIVAL of application-level programs that make use of posits and floats simultaneously. To the best of our knowledge, this is the first work that enables complete posit and quire capabilities in hardware.

Synthesis results show that half the area dedicated to the PAU is occupied by the quire and its operations. When comparing with the only previous work which includes quire capabilities [19], our proposal consumes slightly less power and only 10% more area, while also providing full posit operations support. When focusing on the 32-bit PAU without the quire, our proposal requires 32% more area and 38% more power than the 32-bit FPU. This goes in line with the results of recent works which reuse the F RISC-V extension [22], where authors obtain a 30% increase in FPGA resources when comparing their PAU to the FPU.

The Posit vs IEEE-754 comparison benchmark results show that 32-bit posits are up to 4 orders of magnitude more accurate than 32-bit floats when calculating the GEMM due to the quire. Moreover, they do not show a performance degradation compared with floats, thus providing a potential alternative when operating with real numbers. In addition, our proposal performs significantly better than available commercial solutions, obtaining up to 8× speedup when multiplying small matrices.

Some known limitations occur in the use of the quire. As it is a single internal register in the PAU, PERCIVAL cannot support parallel accumulation into different independent accumulators. This also prevents safe automatic context switches, as the value of the quire cannot be loaded or stored in memory. Therefore, when developing programs for PERCIVAL this must be taken into account to not overwrite the value of the quire.

As future work, we plan to implement and evaluate on PERCIVAL large-scale scientific applications which make use of dot products, leveraging the accuracy gains of fused operations.

REFERENCES

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754–2019 (Revision of IEEE 754-2008), pp. 1–84, Jul. 2019.

[2] J. L. Gustafson and I. T. Yonemoto, “Beating floating point at its own game: Posit arithmetic,” *Supercomput. Front. Innov.*, vol. 4, no. 2, pp. 71–86, Apr. 2017.

[3] A. Guntoro *et al.*, “Next generation arithmetic for edge computing,” in *Proc. Des. Automat. Test Eur. Conf. Exhib.*, 2020, pp. 1357–1365.

[4] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, “Posits: The good, the bad and the ugly,” in *Proc. Conf. Next Gener. Arithmetic*, 2019, pp. 1–10.

[5] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, “The RISC-V instruction set manual, volume I: User-level ISA, version 2.0,” EECS Dept., Univ. California, Berkeley, Tech. Rep. UCB/EECS-2014–54, May 2014.

[6] R. Murillo, A. A. Del Barrio, and G. Botella, “Deep PeNSieve: A deep learning framework based on the posit number system,” *Digit. Signal Process.*, vol. 102, Jul. 2020, Art. no. 102762.

[7] G. Raposo, P. Tomás, and N. Roma, “PositNN: Training deep neural networks with mixed low-precision posit,” in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2021, pp. 7908–7912.

[8] H. F. Langroudi *et al.*, “ALPS: Adaptive quantization of deep neural networks with GeneraLized PositS,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops*, 2021, pp. 3094–3103.

[9] A. Dörflinger *et al.*, “A comparative survey of open-source application-class RISC-V processor implementations,” in *Proc. 18th ACM Int. Conf. Comput. Front.*, 2021, pp. 12–20.

[10] F. Zaruba and L. Benini, “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 Trans. Very Large Scale Integration Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

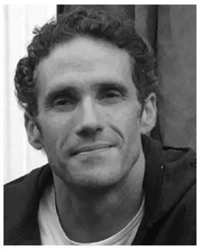
[11] R. Murillo *et al.*, "PLAM: A. posit logarithm-approximate multiplier," *IEEE Trans. Emerg. Topics Comput.*, early access, Sep. 6, 2021, doi: 10.1109/TETC.2021.3109127.
[12] Posit Working Group, "Posit Standard Documentation Release 4.12-draft," *Standard Posit Arithmetic*, Jul. 2021. [Online]. Available: https://posithub.org/posit_standard4.12.pdf
[13] R. Murillo, D. Mallasén, A. A. Del Barrio, and G. Botella, "Comparing different decodings for posit arithmetic," in *Proc. Conf. Next Gener. Arithmetic*, 2022, pp. 71–86.
[14] J. L. Gustafson, "RISC-V proposed extension for 32-bit posits," Jun. 2018. [Online]. Available: https://posithub.org/docs/RISC-V/RISC-V.htm
[15] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "FPnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 29, no. 4, pp. 774–787, Apr. 2021.
[16] R. Chaurasiya *et al.*, "Parameterized posit arithmetic hardware generator," in *Proc. IEEE 36th Int. Conf. Comput. Des.*, 2018, pp. 334–341.
[17] M. K. Jaiswal and H. K.-H. So, "PACoGen: A. hardware posit arithmetic core generator," *IEEE Access*, vol. 7, pp. 74586–74601, 2019.
[18] R. Murillo, A. A. Del Barrio, and G. Botella, "Customized posit adders and multipliers using the FloPoCo core generator," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2020, pp. 1–5.
[19] N. Sharma *et al.*, "CLARINET: A RISC-V Based Framework for Posit Arithmetic Empiricism," 2021, *arXiv: 2006.00364*.
[20] M. V. Arunkumar, S. G. Bhairathi, and H. G. Hayatnagarkar, "PERC: Posit enhanced rocket chip," in *Proc. 4th Workshop Comput. Archit. Res. RISC-V*, 2020, Art. no. 8.
[21] S. Tiwari, N. Gala, C. Rebeiro, and V. Kamakoti, "PERI: A. configurable posit enabled RISC-V core," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, pp. 1–26, Jun. 2021.
[22] S. D. Ciocirlan, D. Loghin, L. Ramapantulu, N. Tapus, and Y. M. Teo, "The Accuracy and Efficiency of Posit Arithmetic," 2021, *arXiv:2109.08225*.
[23] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "A lightweight posit processing unit for RISC-V processors in deep neural network applications," *IEEE Trans. Emerg. Topics Comput.*, early access, Oct. 21, 2021, doi: 10.1109/TETC.2021.3120538.
[24] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis amp; transformation," in *Proc. Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.
[25] A. Waterman and K. Asanović, "The RISC-V instruction set manual, Volume I: User-Level ISA, Document Version 20191213," A. Waterman and K. Asanović, Eds. RISC-V Found., Dec. 2019. [Online]. Available: https://riscv.org/technical/specifications/
[26] S. H. Leong, "SoftPosit," Mar. 2020. [Online]. Available: https://gitlab.com/cerlane/SoftPosit
[27] R. Murillo, D. Mallasén, A. A. Del Barrio, and G. Botella, "Energy-efficient MAC units for fused posit arithmetic," in *Proc. IEEE 39th Int. Conf. Comput. Des.*, 2021, pp. 138–145.

DAVID MALLASÉN received the BSc degree in computer science and the BSc degree in mathematics from the Complutense University of Madrid (UCM), in 2020, and the MSc degree in embedded systems from the KTH Royal Institute of Technology, specializing in embedded platforms, from 2020 to 2022. Currently, he is working toward the PhD degree in computer engineering at UCM. His main research areas include computer arithmetic, computer architecture, embedded systems, and high-performance computing.

RAUL MURILLO received the BSc degree in computer science and the BSc degree in mathematics from the Complutense University of Madrid (UCM), Spain, in 2019, where he also received the MSc degree in computer science in 2021. His main research interests include approximate computing, new computer arithmetic, and deep neural networks (DNNs). He is currently working toward the PhD degree with UCM related to the previously mentioned areas.

ALBERTO A. DEL BARRIO (Senior Member, IEEE) received the PhD degree in computer science from the Complutense University of Madrid (UCM), Madrid, Spain, in 2011. He has performed stays with Northwestern University, University of California at Irvine and University of California at Los Angeles. Since 2021, he is an Associate Professor (tenure-track, civil-servant) of Computer Science with the Department of Computer Architecture and System Engineering, UCM. His main research interests include design automation, arithmetic and their application to the field of artificial intelligence. He is leading the PARNASO project, funded by the Leonardo Grants program by Fundación BBVA. The main objective is to natively integrate the posit format in a hardware/software platform. Since December 2020 he is an ACM senior member, too.

GUILLERMO BOTELLA (Senior Member, IEEE) received the MASc degree in physics (fundamental) in 1998, the MASc degree in electronic engineering in 2001, and the PhD degree (computer engineering) in 2007, all from the University of Granada, Spain. He was a research fellow funded by EU working with the University of Granada, Spain and the Vision Research Laboratory, University College London, U.K.. After that, he joined as Assistant Professor with the Department of Computer Architecture and Automation of Complutense University of Madrid, Spain, where he is currently Associate Professor. He has performed research stays acting also as visiting professor from 2008 to 2012 with the Department of Electrical and Computer Engineering, Florida State University, Tallahassee, USA. His current research interests include image and video processing for VLSI, FPGAs, GPGPUs, Embedded Systems, and novel computing paradigms such as analog and quantum computing.


LUIS PIÑUEL received the MSc and PhD degrees in computer science from the Universidad Complutense de Madrid (UCM) in 1996 and 2003, respectively. He is an Associate Professor with the Department of Computer Architecture and Systems Engineering, Universidad Complutense de Madrid, Spain. His research interests include computer architecture, high-performance computing, embedded systems, and resource management for emerging computing systems. In these fields, he is coauthor of more than 70 publications in prestigious journals and international conferences, several book chapters and he has advised or coadvised five PhD dissertations. Worried about improving knowledge transfer between research institutions and industry, he has directed more than 12 research contracts with different companies (Texas Instruments, Imagination Technologies, Indra,...). He has also served as evaluator for several national agencies and has also been member of the Board of Directors of the Spanish Computer Architecture Society (SARTECO).

MANUEL PRIETO-MATIAS received the PhD degree from the Complutense University of Madrid (UCM) in 2000. Since 2002, he has been a professor with the Department of Computer Architecture, UCM, being a full professor since 2019. His research interests include high-performance computing, non-volatile memory technologies, accelerators, and code generation and optimization. His current focus is on effectively managing resources on emerging computing platforms, emphasizing the interaction between the system software and the underlying architecture. Manuel has coauthored more than 100 scientific publications in journals and conferences in parallel computing and computer architecture. He is a member of the ACM.