A Lightweight Posit Processing Unit for RISC-V Processors in Deep Neural Network Applications

Marco Cococcioni, *Senior Member, IEEE,* Federico Rossi, Emanuele Ruffaldi, *Senior Member, IEEE,* Saponara Sergio, *Senior Member, IEEE*

***Abstract*—Nowadays, two groundbreaking factors are emerging in neural networks. Firstly, there is the RISC-V open instruction set architecture (ISA) that allows a seamless implementation of custom instruction sets. Secondly, there are several novel formats for real number arithmetic. In this work, we combined these two key aspects using the very promising posit format, developing a light Posit Processing Unit (PPU-light). We present an extension of the base RISC-V ISA that allows the conversion between 8 or 16-bit posits and 32-bit IEEE Floats or fixed point formats in order to offer a compressed representation of real numbers with little-to-none accuracy degradation. Then we elaborate on the hardware and software toolchain integration of our PPU-light inside the Ariane RISC-V core and its toolchain, showing how little it impacts in terms of circuit complexity and power consumption. Indeed, only 0.36% of the circuit is devoted to the PPU-light while the full RISC-V core occupies the 33% of the overall circuit complexity. Finally we present the impact of our PPU-light on a deep neural network task, reporting speedups up to 10 on sample inference processing time.**

***Index Terms*—alternative representations of real numbers, posit arithmetic, hardware synthesis, RISC-V processors, instruction set architecture extension, scalar operations**

I. Introduction

Recently, RISC-V rose as an open-source alternative CPU architecture [1]–[3].

It quickly became an important competitor of Intel, AMD and ARM CPUs (both for 32 and 64-bit variants) for being royalty free. Several companies in industry have already supported and funded the project. Among them we can find star companies like Intel, Microsoft and ST Microelectronics [4]. The most important and crucial feature of RISC-V is its open-source instruction set architecture (ISA). This means that anyone can modify it by extending the ISA with his very own instructions and functionalities: this feature is fundamental, since it allows the design of very low-latency co-processors, functional units and accelerators without the need to consider them as external devices that require memory mapping and interrupts.

In the latest years, several representations for real number operations have been proposed by industry and research such as Intel with Flexpoint [5, 6], Google with BFLOAT16 [7], IBM with DLFloat [8], NVIDIA with TensorFloat32 [9] and Facebook with logarithmic numbers [10].

M. Cococcioni, F. Rossi and S. Saponara are with the Department of Information Engineering, University of Pisa, 56122 Pisa – Italy, e-mail: {marco.cococcioni, federico.rossi, sergio.saponara}@unipi.it

E. Ruffaldi is with MMI spa, e-mail: emanuele.ruffaldi@mmimicro.com

One of the most promising alternative to IEEE 32-bit Floating-point standard is the posit™ format [11]. Posits proved to be able to match single precision (i.e. IEEE 32-bit floats) accuracy (in machine learning and neural network tasks) performance with only 16 bits used for the representation both in our previous works and in independent research [12]–[14]. Moreover with just 8 bits, the overall performances did not degrade critically, as shown in [15, 16].

In this work, we envision the adoption of these two disruptive innovations (RISC-V open architecture and posit arithmetic) to enable posit support inside a RISC-V core in a transparent way, without changing other component behaviour.

Our goal is to seamlessly extend RISC-V with a Posit Processing Unit (PPU) without interfering with the pre-existent architecture (e.g. without modifying or removing already existent floating point support).

In this paper we aimed to add the fewest instructions possible to the RISC-V architecture, such that we could perform compression and decompression of weights using the posit format for storage. It is clear that, since we did not want to entirely replace floats, we benefit from having an interchange and compressed format for moving information. Furthermore, we analyze what could happen when we employ this compression in a "real-time" approach, and not only on the storage. Of course, adding additional instructions impacted the performance, since the effort of a full-posit processing cannot be stand without a full-hardware support for posits (i.e. full PPU with all the arithmetic operations). However, we believe that the PPU^{light} is a milestone in the road towards full posit computing, since it enables posit-support with the most lightweight approach.

The paper is structured as follows: in Section 2 we briefly introduce posit numbers and some of its interesting properties. In Section 3 we present our C++ posit library and its overall software architecture. In Section 4 we summarize the key aspects of the RISC-V instruction set architecture (ISA) and in Section 5 we elaborate on our posit extension for the RISC-V ISA. In Section 6 we provide the design strategies behind the logic circuit implementation of selected posit operations and metrics of the implementation of said circuit on a Digilent Genesys 2 FPGA board. In Section 7 we present validation results on the official RISC-V instruction emulator and outcomes of the integration of our PPU^{light} core inside the Ariane open-source RISC-V core. Furthermore we present benchmarks on common deep

learning scenarios. Section 8 contains a summary and the conclusions of this work.

II. Related works and contributions

A. Related Works

In [17] the authors present a fully functional posit floating point unit and RISC-V posit extension exploiting and overloading the already existent RISC-V IEEE 32-bit float instructions. The authors introduce a posit unit with 32 32-bit posit registers with an additional status register. The final design is a 32-bit posit co-processor that is decoupled from the RISC-V core execution pipeline. The proposed unit reportedly occupies 3507 slice LUTs and 1294 slice registers on an Artix-7-100T Xilinx FPGA running at 100 MHz.

In [18] the authors present a benchmark platform for alternative real number arithmetic, including posits. They introduce two components: i) Melodica, a complete posit unit implementing several arithmetic, quires and fused multiply-add operations; ii) Clarinet, a RISC-V core with Melodica support. The authors leveraged the custom op-code space in RISC-V to add custom instructions, as well as a custom C compiler toolchain. Furthermore they added a new set of posit registers with parametric posit size.

In both works there is the trend of altering the pre-existent RISC-V architecture by adding new registers to the instruction set architecture. As we describe in the section hereafter, our aim is to provide a seamless integration of our PPU component with minimal architectural modifications as well as a complete software tool-chain that do not need a custom compiler to leverage the new instruction set.

B. Our Contribution

Similarly to previous works, we leveraged the RISC-V custom opcode space to introduce new posit instructions inside a RISC-V processor core. However, we decided to diverge from previous works in two different ways, one at hardware level and the other at software level:

- At hardware level we decided to keep our PPU^{light} as light as possible, in order to adhere to RISC-V minimalism, without bringing too much additional complexity to the RISC-V core. Indeed we did not introduce new posit registers but we decided to re-use the integer ALU registers also for posit operands. This simplified extremely the integration of our PPU^{light} design inside the RISC-V core. On the other hand we only implemented conversion instructions between posits, IEEE 32-bit floats and fixed-points, enabling quire support at a software level. Note that, once converted to a fixed-point, the sum of two posits is simply the sum of two integers: this means that we can perform true posit floating-point-like computations without involving the IEEE FP32 floating point unit at all. As a trade-off, this is, of course, not a lossless conversion between posits and fixed-point format.
  Furthermore, we decided to support only 8 and 16 bit posits. This is because 16-bit posits proved to be as good as IEEE floats in deep learning applications [12]–[14]. Furthermore, having support for 8-bit posits allows very fast arithmetic without having significant accuracy degradation ([15, 16]).
- At software level we decided not to modify any element of the C compiler toolchain, making the overall software library completely portable on any modern RISC-V C compiler. We indeed make use of inline assembly instruction emission directly from C code, then wrapped in a high-level intrinsic interface. Everything is finally self-contained inside a single header file.

As a consequence of these two points we consider our light PPU can be used in two different ways (see Figure 1):

- If the RISC-V processor embeds an FPU the PPU^{light} can be used as a wrapper, providing a data compression by a factor up to 4, with little accuracy degradation. The cost of compression and decompression is the cost of converting a posit to a float and vice-versa.
- If the RISC-V processors does not embed an FPU or we want to exploit only the ALU, the PPU^{light} can function as a wrapper of fixed-point representation. Indeed, once we have converted between posit and fixed-point, the basic arithmetic operations can be computed just with the ALU. Also note that, for half of the posit domain, that is the $[-1, 1]$ range, the conversion between posit and fixed point is a simple left shift of 2 positions followed to 0 padding on the most significant bits to reach desired fixed-point size.

Figure 1 shows an example of the two possible approaches: in the top one we employ the PPU^{light} alongside the FPU and the ALU, supporting both floating point and fixed point as a back-end of our operation. In the bottom one, we put the PPU^{light} in a scenario where only the ALU is present, thus enabling posit computation with a pure fixed-point approach. Note that in both cases, our approach is non-disruptive. This means that the existent architecture remains untouched, with just the addition of a new module. From a software level perspective this offers the highest transparency possible.

A summary of the different possible operation implementations is shown in Table I. More operations that do not need encoding and decoding are summarised in Table II.

TABLE I: Basic arithmetic operations for posit numbers in our approach.

Operation	Functional Unit (back-end)	Conversion needed
Sum	FPU or ALU (fixed point)	yes
Product	FPU or ALU (fixed point)	yes
Multiplication	FPU or ALU (fixed point)	yes
Division	FPU or ALU (fixed point)	yes
Comparison	ALU (integer comparison)	no

III. Posit numbers and cppPosit library

Posit™ numbers have been presented for the first time by John L. Gustafson in [11]. This novel format is fixed length

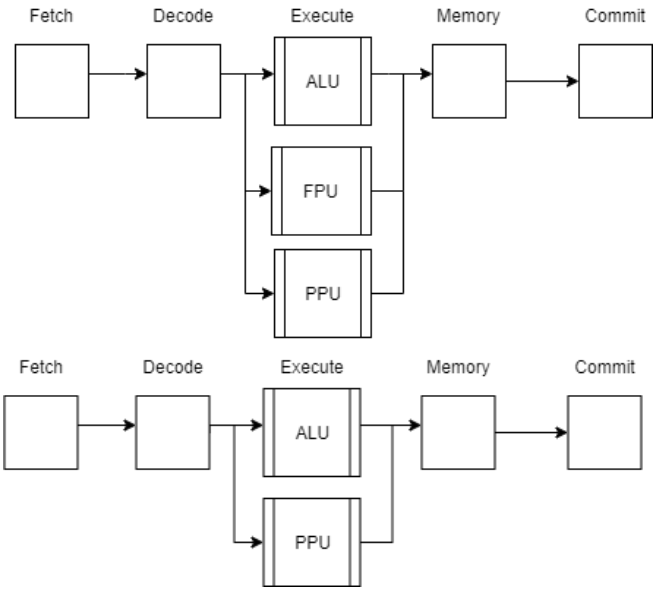

Fig. 1: A visualization of the two possible use cases of the PPU^{light}.

one (length and exponent length can be configured), with up to 4 fields as reported in Figures 2 and 3:

- Sign field: 1-bit
- Regime field: variable length, composed by a string of bits equal to 1 or 0 ended, respectively by a 0 or 1 bit.
- Exponent field: at most *es* bits
- Fraction field: variable length mantissa

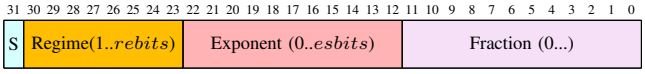

Fig. 2: Illustration of a posit$\langle 32, 11\rangle$ data type.

Given $posit\,\langle nbits, esbits\rangle$, represented by the signed integer X and let e and f be respectively the exponent and fraction values, the real number r represented by X encoding is:

$$r = \begin{cases} 0, \text{ if } X = 0 \\ \text{NaN, if } X = -2^{(nbits-1)} \\ sign(X) \cdot useed^k \cdot 2^e \cdot (1+f), \text{ otherwise} \end{cases} \quad (1)$$

where $useed = 2^{2^{esbits}}$ and k is the value of the regime. The regime field is run-length encoded. This means that the value represented by this field depends on its length. In particular the regime length is the number of identical subsequent bits stopped by the opposite valued bit. For example, given a regime 00001, its length will be 4. The value k depends on the regime length l and the regime identical bit value b:

$$k = \begin{cases} -l, \text{ if } b = 0 \\ l - 1, \text{ otherwise} \end{cases} \quad (2)$$

Figure 3 shows an example of posit decoding. Given the sequence on top of the figure, after detecting it starts with one 1, we have to compute the 2's complement of all the remaining bits (passing from 001-110-111011001 to 110-001-000100111). Then we can proceed to decode the posit. The regime bit-string is 110, therefore the regime length l is 2 (two consecutive 1 and a 0, $b = 1$). Thus, the k value is 1, following Equation (2). The exponent bits are 001 that translates into an exponent value of 1. Finally, the mantissa bits are 000100111, representing the integer 39 and the mantissa length is $9 \rightarrow 2^9 = 512$, thus the fraction value is $\frac{39}{512}$. The associated real value is therefore: $-256^1 \cdot 2^1 \cdot (1 + 39/512)$. The final value is therefore $-512 \cdot (1 + 39/512) = -551$ (exact value, i.e., no rounding, for this case).

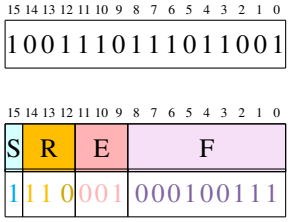

Fig. 3: An example of a 16-bit Posit with 3 bits for the exponent ($esbits$=3).

As reported in [16] by the authors, this novel format introduces new interesting properties with $esbits = 0$. In this case it is possible to implement fast versions of common operations (possibly with slight approximation); these particular versions can be computed just by using the arithmetic-logic unit (ALU) of the CPU, since they only employ bit manipulation and basic integer arithmetic. Among these operations we can accelerate, we can find the double and half operators ($2x$ and $x/2$), the inverse operator ($1/x$) and the one's complement ($1 - x$). As also seen in our past work this kind of implementation allows the vectorization of several posit operations by reusing the integer vector registers and functional units.

Software support for posits is enabled by our cppPosit library [19], developed in Pisa and maintained by the authors of this work. The library uses templatization to define different posit configurations during compilation. The posit operations are put into four different levels ($\mathcal{L}1$-$\mathcal{L}4$) with increasing computational complexity [16]. The first level $\mathcal{L}1$ is the simplest and fastest and comprises all the operators in Table II. In this table, approximated column states whether the operation is an exact or an approximated result and reporting the requirements to be fulfilled. For instance notice how $1 - x$ can be computed using fast bit manipulations only when $x \in [-1, 1]$. We use three different back-ends to execute posit operations that cannot be emulated directly via ALU:

- Floating point back-end, using the FPU;
- Fixed point back-end, exploiting big-integer support (64 or 128 bits) for operations;
- Tabulated back-end, generating lookup tables for most of the operations (suitable for Posit$\langle [8, 12], * \rangle$ due to table sizes).

TABLE II: Most interesting $\mathcal{L}1$ operations implemented in cppPosit using only the ALU.

DNN Operation	Approximated	Requirements
$2 \cdot x$	no	*esbits=0*
$x/2$	no	*esbits=0*
$1 - x$	no	*esbits=0,* $x \in [-1, 1]$
$1/x$	yes	*esbits=0*
FastSigmoid	yes	*esbits=0*
FastTanh [16]	yes	*esbits=0*
FastELU	yes	*esbits=0*

IV. RISC-V ISA Architecture

The RISC-V [1] architecture is a modular, open-source and royalty-free instruction set architecture (ISA) and comprises both 32 and 64-bit architectures. The ISA is built out of small sub-ISAs. The base subsets are referred as *base integer instruction sets* and identified by the letter `I`. Furthermore, a RISC-V based architecture has additional extensions; some extension are *frozen*, since their encoding and behaviour has been already ratified and cannot change during the current revision of the ISA. These extensions are integer multiplication/division operations (`M`), single (`F`), double (`D`) precision floating point operations (following the IEEE 754 Float standard) and atomic instructions (`A`). In order to access the RISC-V architecture for customization of the ISA and program execution we have the following two choices, on which we also elaborate more in the next sections:

- The RISC-V ISA simulator (also known as Spike [20]). This simulator fully emulate the instruction set of a 64-bit RISC-V with all the extensions said above, but also with vectorization support. It brings a high-level interface to customize the instruction set, simply adding the opcodes and the behaviour of the instructions, all in C++. In order to execute compiled binaries on the Spike simulator, we must pass through the RISC-V proxy kernel, that embeds the *Berkeley Boot Loader* and allows us to execute statically linked RISC-V binaries. This also comes with a customizable high-level interface where we can define the instruction opcodes. The combination of Spike and the proxy kernel allows us to execute any RISC-V binary on any other architecture for which it is compiled (e.g. x86) machine. To compile and link RISC-V binaries we used the official GCC cross compiler from the RISC-V organization repository. This allowed us to produce RISC-V binaries on a x86 host.
- The Ariane RISC-V FPGA core. This core is a 6-stage (2-stage speculative frontend, instruction decoding, issuing, executing and commit), single issue, in-order CPU. It embeds a 64-bit RISC-V instruction set with I, M, A and C subsets. It also support M, S and U privilege levels, allowing the possibility to execute any Unix-like operative system on it. The core is completely open-source and extensible using SystemVerilog.

V. RISC-V posit ISA extension design

We extended the RISC-V ISA to support posit operations keeping in mind the minimalism of RISC-V. The core idea is to implement basic posit operations (addition, multiplication and others) using other wider types as backend. Therefore, we did not introduce new ad-hoc registers for posits. Instead, we aimed to re-use existing floating point and integer registers, providing conversion instructions from/to floating point (on float registers) and fixed point (on integer registers) numbers.

Note that, unlike floating point numbers, when converting between posit and fixed point, the size of the latter depends on the posit characteristics. From now on we will refer to a fixed point with N overall bits and $N/2$ bits for the mantissa as fx$\langle N \rangle$. Then, a posit$\langle 8, 0 \rangle$ will be converted to fx$\langle 16 \rangle$, a posit$\langle 16, 0 \rangle$ to fx$\langle 32 \rangle$ and a posit$\langle 16, 1 \rangle$ to fx$\langle 64 \rangle$.

Since we employ posits with an overall size of 16 or 8 bits, we are performing a lossy data compression by a factor 2 or 4 if we start from an IEEE FP32 format (maintaining a similar inference accuracy, in machine learning and neural network tasks, as demonstrated in our previous works [15, 16]. Moreover, even if we use a wider backend to perform computations, the data expansion is performed only within the posit processing unit. Therefore, we are transferring compressed data from memory to CPU integer/float registers. This means that just by using posits as a lossy compressed information storage can reduce the amount of data transferred up to a factor 4.

The instruction encoding uses the suffix `'b0001011` (6 least significant bits) that is reserved for custom ISA extension.

As listed in Table III, we added the following instructions:

- Floating point and posit conversions
  - `FCVT.S.P8/FCVT.P8.S`: Float to/from posit$\langle 8, 0 \rangle$ conversion
  - `FCVT.S.P16.0/FCVT.P16.0.S`: Float to/from posit$\langle 16, 0 \rangle$ conversion
  - `FCVT.S.P16.1/FCVT.P16.1.S`: Float to/from posit$\langle 16, 1 \rangle$
- Fixed point and posit conversions
  - `FXCVT.H.P8/FXCVT.P8.H`: fx$\langle 16 \rangle$ to/from posit$\langle 8, 0 \rangle$ conversion
  - `FXCVT.W.P16.0/FXCVT.P16.0.W`: fx$\langle 32 \rangle$ to/from posit$\langle 16, 0 \rangle$ conversion
  - `FXCVT.L.P16.1/FXCVT.P16.1.L`: fx$\langle 64 \rangle$ to/from posit$\langle 16, 1 \rangle$
- Posit to posit conversions
  - `FCVT.P8.P16.0/FCVT.P16.0.P8`: posit$\langle 8, 0 \rangle$ to/from posit$\langle 16, 0 \rangle$ conversion
  - `FCVT.P16.1.P16.0/FCVT.P16.0.P16.1`: posit$\langle 16, 1 \rangle$ to/from posit$\langle 16, 0 \rangle$ conversion
  - `FCVT.P8.P16.1/FCVT.P16.1.P8`: posit$\langle 16, 1 \rangle$ to/from posit$\langle 8, 0 \rangle$ conversion

Once the instructions have been encoded we need to provide a high-level interface to use them. The idea is to

TABLE III: Instruction listing for RISC-V RVXposit extension

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
1100000		00010	rs1	000	rd	0001011	FCVT.S.P8
1100000		00011	rs1	000	rd	0001011	FCVT.S.P16.0
1100000		00011	rs1	010	rd	0001011	FCVT.S.P16.1
1101000		00010	rs1	000	rd	0001011	FCVT.P8.S
1101000		00011	rs1	000	rd	0001011	FCVT.P16.0.S
1101000		00011	rs1	010	rd	0001011	FCVT.P16.1.S
1100000		00010	rs1	001	rd	0001011	FXCVT.H.P8
1100000		00011	rs1	001	rd	0001011	FXCVT.W.P16.0
1100000		00011	rs1	011	rd	0001011	FXCVT.L.P16.1
1101000		00010	rs1	001	rd	0001011	FXCVT.P8.H
1101000		00011	rs1	001	rd	0001011	FXCVT.P16.0.W
1101000		00011	rs1	011	rd	0001011	FXCVT.P16.1.L
1101000		00010	rs1	001	rd	0001011	FXCVT.P8.H
1101000		00011	rs1	001	rd	0001011	FXCVT.P16.0.W
1101000		00011	rs1	011	rd	0001011	FXCVT.P16.1.L
1100000		00010	rs1	100	rd	0001011	FCVT.P8.P16.0
1100000		00011	rs1	100	rd	0001011	FCVT.P16.0.P8
1101000		00011	rs1	111	rd	0001011	FCVT.P16.1.P16.0
1101000		00010	rs1	101	rd	0001011	FCVT.P16.1.P8
1100000		00011	rs1	110	rd	0001011	FCVT.P8.P16.1
1101000		00011	rs1	101	rd	0001011	FCVT.P16.0.P16.1

implement a C intrinsic for each instruction exploiting the inline assembly `__asm__` operator to emit the byte-code associated to the specific instruction. This approach avoids us to implement the code generation inside the compiler. Instead, we let the compiler choose the proper registers for the intrinsic exploiting C/C++ register allocation with the keyword `register`.

Listing 1 shows an intrinsic example for the float to posit$\langle 8, 0\rangle$ conversion. The many `.set` directives are used to set RISC-V register identifiers. The `.byte` directive is used to emit the four bytes that compose the instruction. Comparing the four bytes of the instruction with the encoding in Table III we can see that both `rs1` and `rd` (source and destination register) are not being explicitly set in the intrinsic. Finally, the two register allocations in the function header use the RISC-V standard register names for input and output passing.

We instrumented the cppPosit library to be compiled with specific flags to directly use said hardware instructions instead of using software emulation. For example, without hardware support the conversion between float and posit needs a series of bit manipulation, done in software. If we provide PPU support the same conversion results in a single call to the `FCVT.S.*/FCVT.*.S` instruction.

This approach has three key aspects:

- We implemented some core posit operations that can not be implemented as $\mathcal{L}1$ operations
- We discarded other slow instructions that require activation and withdrawal like in an external execution unit.
- We may seamlessly run in a super-scalar environment with multiple parallel execution units since we only used native integer and floating point registers.

Listing 1: Intrinsic example for FCVT.S.P8

```
int __fcvt_f32_p8 (float a) {
 register float p1 asm (``fa0") = a;
 register int result asm (``a1");
 __asm__ volatile(
  ``"
  ``.set rfs0,8\n"
  ``.set rfs1,9\n"
  ...
  ``.set op,0xb\n"
  ``.set opf1,0x0\n"
  ``.set opf2,0x2\n"
  ``.set opf3,0x60\n"
  ``.byte op|((r%[result]&1) <<7),
((r%[result]>>1)&0xF)|(opf1<<4)|((r%1&1)<<7),
((opf2&0xF) << 4) | ((r%1>>1)&0xF),
((opf2>>4)&0x1)|(opf3<<1)"
  : [result] ``=r"(result)
  :``f"(p1), ``[result]"(result));
 return result;
}
```

Regarding the real hardware implementation, we used the Ariane RISC-V core [21] as development base, as described in next section.

VI. RISC-V posit ISA extension implementation

A. Circuit design

In order to provide a circuit design for our PPU^{light} we considered several key points to simplify the final logic design:

- IEEE floating point values are encoded in a module and sign like representation while posits are encoded using 2's complement representation. Therefore, when

converting from IEEE floats we just ignore the sign and build the positive posit. Then we use the sign to apply the 2's complement to the result if negative.

- Given a Posit$\langle 16, 0\rangle$ the size of the regime spans from a minimum of 2 to a maximum of 15 bits. As a result the mantissa size spans from a minimum of 0 to a maximum of 12 bits. This means that, given a 23-bit mantissa IEEE Float, the 8 least significant bits of the float are set to 0. The same concepts hold for Posit$\langle 8, 0\rangle$.
- We can build the Posit$\langle X, 0\rangle$ regime arithmetically shifting an appropriate value by the $\log_2 X$ least significant bits of the FP32 normalized exponent. For Posit$\langle 16, 0\rangle$ we shift the signed integer represented by 2^{15} (`8000` in hexadecimal notation, as in Figure 7). For Posit$\langle 8, 0\rangle$ we shift the signed integer represented by 2^7. Furthermore, we always build the regime starting from the absolute value of the normalized FP32 exponent. It can be then transformed to the "negative" regime in case of negative exponent values as in Figure 7. In this circuit we take the floating point exponent on 8 bits and produce the regime bits corresponding to that exponent. In order to generate the regime we arithmetically shift the signed integer represented by 2^{15} (`'h8000`) by the amount specified by the 4 last significant digits of the floating point exponent. The same procedure holds for both negative and positive exponents, with just a negation at the end to restore the correct sign.
- Decoding the regime is particularly interesting since we need to employ a *find first set* module (or *find first unset*) to evaluate the regime length. The output of the *find first set* module is the index i of the highest set bit (discarding the sign if present). Therefore, as reported in Figure 5, the regime length is actually computed as $l = 14 - i$. At the end, the k-value (which is the non-normalized floating point exponent) is obtained from the regime length as $-l$ or $l + 1$, depending on the regime "sign". In the circuit of Figure 5, we take the regime field and output the corresponding value of k, that is used in Equation (1) to compute the posit value. The core part is represented by the two *find high* modules that help to compute the number of subsequent bit set (or unset). If we subtract this number at the maximum length of the regime (that is 14, or `'he` in hexadecimal notation) we get the actual regime length l. Finally if we follow Equation (1), we can retrieve k from l.

In Figure 4 we show a simplified circuit for conversion from posit$\langle 16, 0\rangle$ to FP32. The 16-bit regime decoder module is implemented by the simplified circuit shown in Figure 5. Note how when converting to IEEE floats we firstly compute the absolute value of the posit number and then convert it to a floating point one. At the end of the circuit, we just replicate the bit sign of the posit in the bit sign of the floating point, being it represented in sign and module.

In Figure 6 we show a simplified circuit for conversion

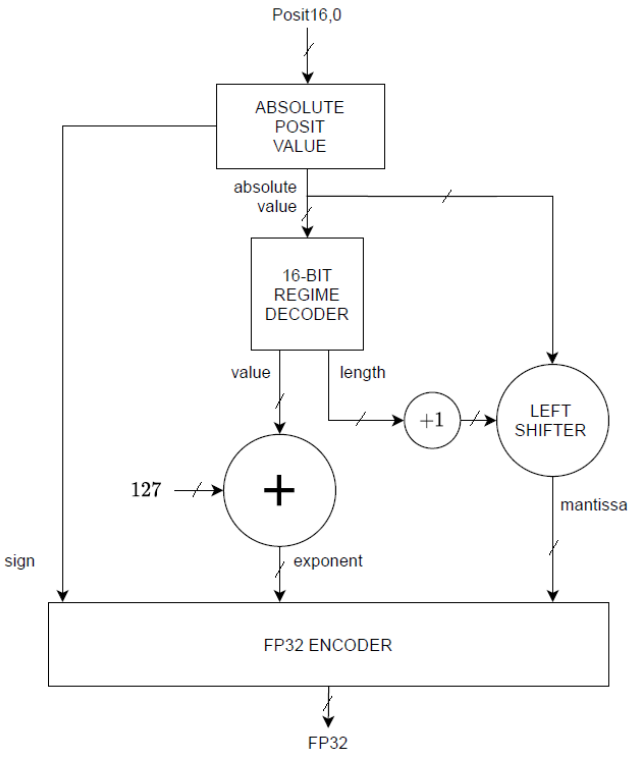

Fig. 4: Logical circuit for the posit$\langle 16, 0\rangle$ to 32-bit floating point converter.

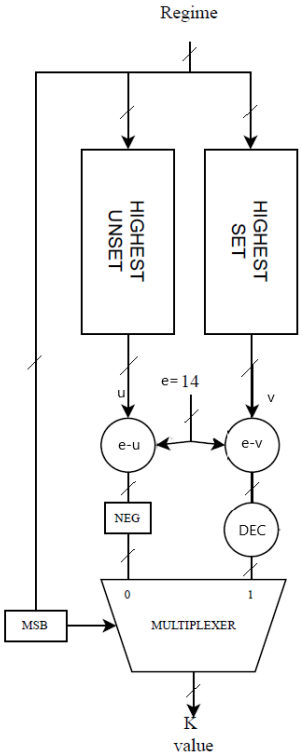

Fig. 5: Logical circuit for the posit$\langle 16, 0\rangle$ regime **decoder**. K value is the regime value as in (2).

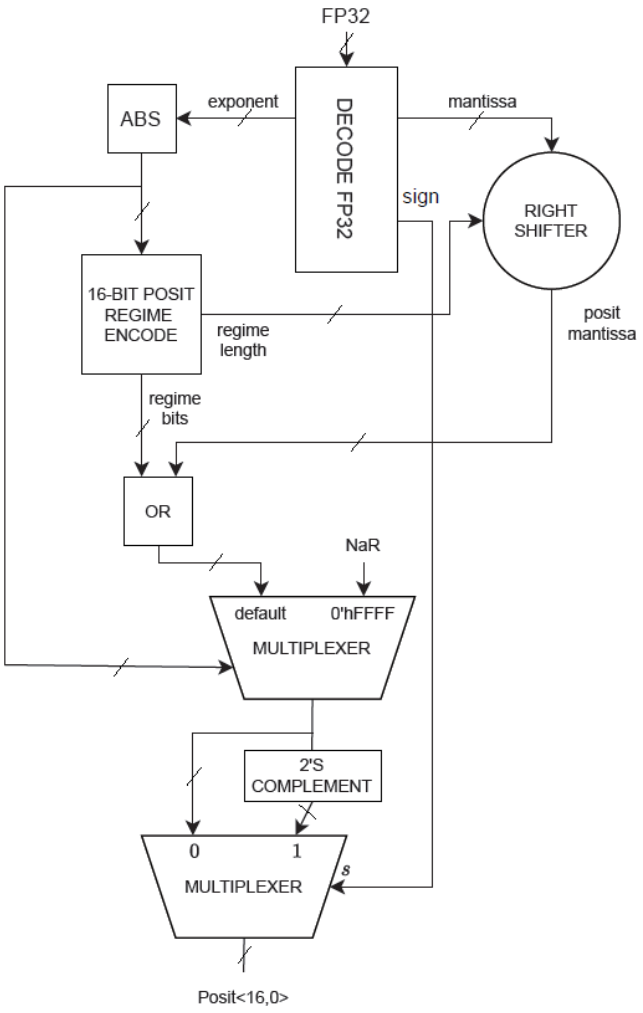

Fig. 6: Logical circuit for the 32-bit floating point to posit$\langle 16, 0\rangle$ converter. *NaR* is *Not A Real*.

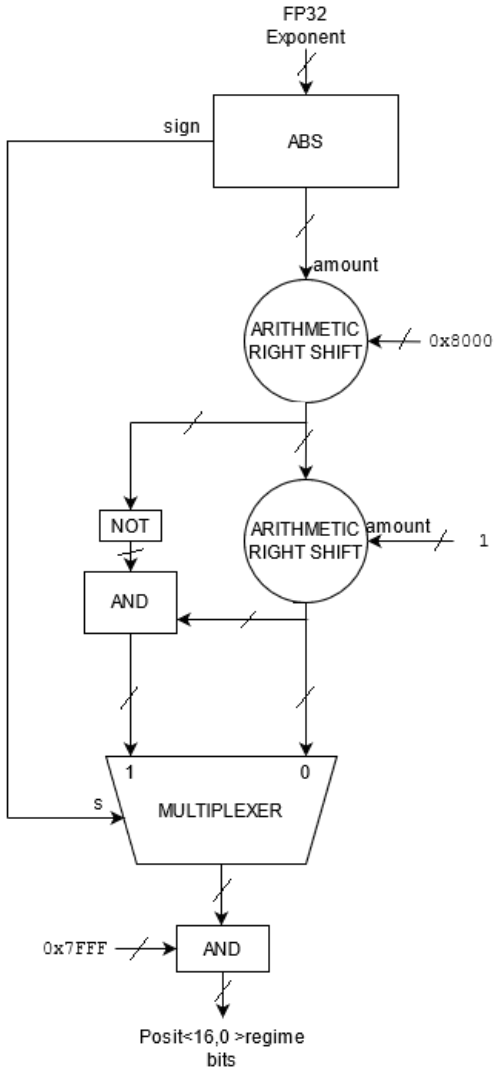

Fig. 7: Logical circuit for the posit$\langle 16, 0\rangle$ regime **encoder**. *Amount* is the number of position to shift the other input in the *Arithmetic Right Shift* module.

from FP32 to posit$\langle 16, 0\rangle$. The first multiplexer in the cascade of two multiplexers takes the exponent value as input; this input acts as a mask to detect *Not A Real (NaR)* values. The 16-bit regime encoder module is implemented by the simplified circuit shown in Figure 7. Note how when converting from IEEE floats we just ignore the sign and build the positive posit and then we use the sign to apply the 2's complement to the result if negative. Furthermore we build the different posit fields considering their maximum possible length without computing any length but the regime one, making the circuit simpler.

VII. Results

A. Hardware Results (Synthesis Outcomes)

For the hardware implementation we chose to use the Xilinx Genesys 2 board (equipped with a Kintex 7 `XC7K325T-2FFG900C` FPGA component). We chose this board to minimize the implementation effort of our PPU inside a RISC-V core. Indeed, we used the ARIANE RISC-V core that was initially designed for this specific board.

Once we designed the logic circuits for the posit conversions we wrapped them in a single functional core. A simplified example is shown in Figure 8.

Note that all the inputs and the output are connected to registers to break-down combinatory-only logic chains. It is crucial how we can reuse the same hardware module for posit-to-posit conversion when using either 8-bit posit or 16-bit posit. Indeed, if we convert a posit$\langle 16, 1\rangle$ to a posit$\langle 16, 0\rangle$ and then we shift the latter of 8 bits right we obtain the correspondent posit$\langle 8, 0\rangle$. Similarly, if we want to convert a posit$\langle 8, 0\rangle$ to a posit$\langle 16, 1\rangle$, we can convert the former to posit$\langle 16, 0\rangle$ beforehand and then finally convert the latter to posit$\langle 16, 1\rangle$. This is useful, since conversion between 0 exponent posits is only a shift and conversion between posit$\langle 16, 0\rangle$ and posit$\langle 16, 1\rangle$ is straightforward.

Synthesised design was then implemented into the same board. We performed power, circuit complexity and propagation delay (worst case combinatorial propagation delay of

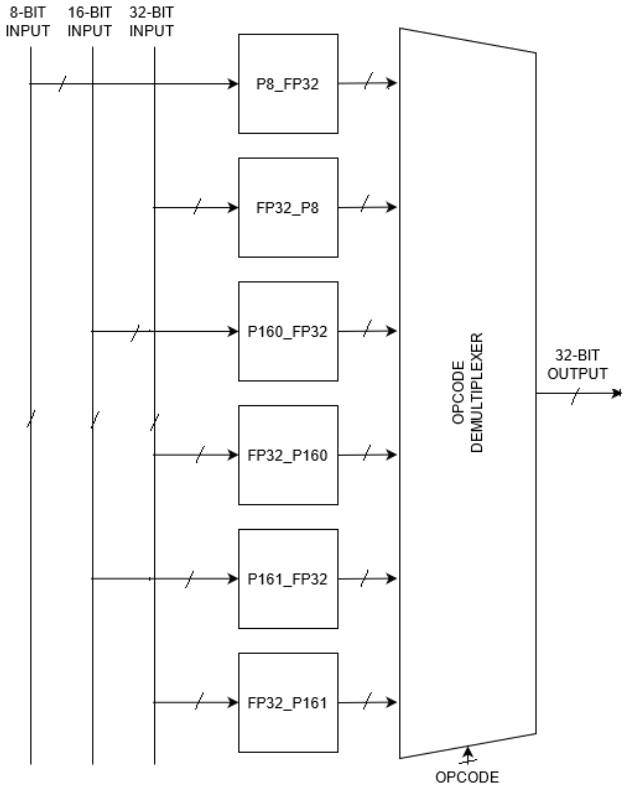

Fig. 8: Simplified design for the PPU^{light} core.

TABLE IV: Summary of the impact of the PPU addition in the synthesized hardware.

Metric	Change
Worst propagation delay (worst critical path)	unchanged
Total power on FPGA component (Kintex 7)	$< 0.02W$ increase
LUT utilisation (w.r.t original design w/o PPU)	1% increase

the PPU^{light}) report for the PPU^{light} component:

- Look-up table (LUT) utilisation: $747/203800$ (0.36%) LUTs used.
- Component latency: $6.332ns$ (worst propagation delay).

Finally, we integrated the new instruction set architecture inside the Ariane RISC-V core and we synthesized it for the Xilinx Genesys 2. We obtained the following quality parameters:

- Clock frequency: $125MHz$
- Total power on FPGA component (Kintex 7): $2.056W$
- Look-up table (LUT) utilisation: $63805/203800$ (31.54%) LUTs used.

Table IV shows the impact of the addition of the PPU^{light} inside the ARIANE core. As reported the impact of the PPU^{light} component is minimal inside the overall architecture.

The PPU^{light} core is connected to the `fu_data_i` instruction data "bus" that is connected to all the main functional unit (e.g. ALU, FPU) inside Ariane. The 7-bit operator lane of the said lane controls the operation selected inside the PPU^{light}, while the single 64-bit operand data is multiplexed and converted to the 3 different lengths for computation. The output is then connected to the output line that goes into the scoreboard module.

TABLE V: Accuracy degradation on inference task when using compressed format on different datasets and network models.

	LeNet		EfficientNetB0
	MNIST	GTRSB	CIFAR100 (top-1)
FP32	98.83%	91.8%	82.2%
posit$\langle 16,1\rangle$	98.83%	91.8%	82.2%
posit$\langle 16,0\rangle$	98.50%	90.5%	82.2%
posit$\langle 8,0\rangle$	98.34%	90.4%	82.1%

B. Software Results on DNN Benchmarks

1) ISA Simulation and validation: We validated the new ISA assuring the coherency between the two version of cppPosit, compiled with or without the PPU^{light} support. In the former case, the ISA instruction were emulated in the Spike simulator and the operation results were compared to the latter case in which posit operations were implemented completely in cppPosit.

Secondly we compared the two version of cppPosit inside the tinyDNN C++ DNN library (our custom version is available at [22]), with a synthetic dataset in a 10-layer deep neural network. This was aimed to provide timing performances for the emulated instruction set and to compare them with the default implementation of posit operations.

For simulation and emulation purposes we employed the Spike RISC-V official Instruction Set Architecture emulator. This emulator was configured to run the RISC-V $RV64GC$ stack with the newly introduced posit instruction set. The Spike simulator ran on a `Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz` processor under a Linux 5.4 environment.

2) DNN Accuracy results: To assess the variation of accuracy with the compressed formats, we tested two simple datasets (the MNIST dataset [23] and the GTSRB datasets [24]) on the networks shown in Figure 9, pre-trained using 32-bit floats. Furthermore, to test the posit compression in a complex scenario, we used the 237-layer deep EfficientNetB0 model described in [25]. In particular we used the pretrained weights from the EfficientNet authors to extract features from the CIFAR100 dataset and perform a transfer learning task on the fully-connected top layer of the network (using 32-bit floats), obtaining similar results to the state-of-art described in the EfficientNet paper. We then tested the model using the posit compression to assess accuracy variations. Table V shows the outcomes of said benchmark.

3) Emulation Results: Table VI shows the results of image inference times on tinyDNN using a 10-layer convolutional neural network (see Figure 9), on 32×32 synthetic images.

Timing performance is referred to the inference time of a single 32×32 image, measured instrumenting the code with the C++ `chrono` directives.

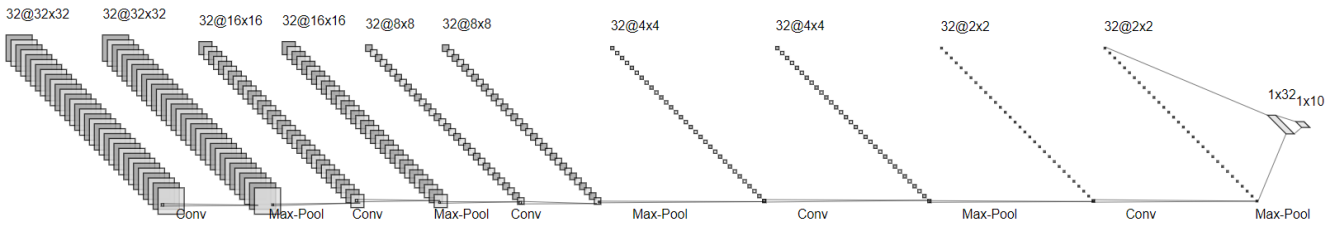

Fig. 9: Schematic of the LeNet-like DNN used for some of the timing and accuracy benchmarks. Note that input layer is a $32 \times 32 \times 1$ grayscale synthetic image and that there is a $\tanh$ activation layer after each convolution. The output prediction is passed through a softmax layer.

TABLE VI: Emulated instruction timing performance on a 10-layer convolutional neural network (Figure 9) with and without the emulated PPU^{light} (ePPU) support for the cppPosit library.

	w/ ePPU (ms)	wo/ ePPU (ms)	Speedup
posit$\langle 8,0\rangle$	92	381	**4.14**
posit$\langle 16,0\rangle$	105	416	**3.96**

The measurements reported are the mean value of multiple executions. We did not report the standard deviation, being it too small.

We choose image size of 32×32 since a frequent application scenario is when the neural network is fed with regions of interests coming from *regional neural networks* that extract sub-images from bigger ones, focusing on particular and smaller regions than the initial one.

As reported in Table VI, the image processing benefits even from the emulated PPU^{light}. This is because the emulation of posit instructions allows a more compact solution, executing less simulated instructions.

4) PPU^{light} unit testing results: We developed a testbench using the same HDL language used for the PPU^{light} unit to test the functionality of the newly introduced components. In particular, we tried every combination for the inputs and tested the outputs of conversion against our cppPosit software library [19]. This means that we have also verified the correct handling of all the corner cases, such as the redundant negative zero and the redundant representations for the Infinity and Not-A-Number in IEEE 32-bit floats.

5) Real hardware results: To assess the performance of the customized core we ran the same benchmarks used in the simulation phase on the ARIANE RISC-V core, equipped with the OpenPiton 12 Linux distribution (based on the ARIANE Linux 4.2). As before, timing performance were measured using C++ internal software `chrono` directives.

Table VII shows the results of the same image processing task on the synthesized Ariane RISC-V core, enabled with the PPU^{light} support. As reported, the relative speedup (computed as $time_{noppu}/time_{ppu}$) shows how much we can benefit from accelerated format compression and decompression using our PPU^{light}.

As described in previous sections, this approach involves the conversion between posits and floats at each operation

TABLE VII: Real HW (FPGA) timing performance on a 10-layer convolutional neural network (Figure 9) with and without the synthesized hardware PPU^{light} support for the cppPosit library.

	w/ PPU (s)	wo/ PPU (s)	Speedup
posit$\langle 8,0\rangle$	5.4	58.87	**10.90**
posit$\langle 16,0\rangle$	11.6	64.54	**5.56**

TABLE VIII: Trade-off between processing time of the network in Figure 9. Processing time was obtained from the real hardware implementation like in Table VII.

	Time (s)	DNN size (bytes)	Compression
IEEE FP32	2,1	224894	-
posit$\langle 16,0\rangle$	11.6	112874	**1.99**
posit$\langle 8,0\rangle$	5.4	56864	**3.95**

(e.g. sum or multiplication). As a result, for each operation, we are performing two more instructions for type conversion. Table VIII summarizes the results obtained with the evaluation of this trade-off. Note that the value obtained with IEEE FP32 is totally independent from the presence of the PPU^{light}.

We may think instead to take an entire posit network and convert it beforehand to IEEE FP32, in order to exploit compression as much as possible without slowing down actual image processing.

This use case is relevant if we think about resource constrained environments where volatile memory is scarce (e.g. embedded or automotive systems) or when frequent transfer of network models are done (e.g. smartphones with recognition software). Moreover, these systems often use an adaptive approach where, depending on the surrounding environment (e.g. snow, night-time, off-road etc.), different machine learning models need to be loaded. This means that having multiple model on volatile storage can highly benefits from compression even when they are not actually loaded into main memory.

In this case, we need to decompress the network into IEEE FP32 format only one time at the beginning of execution. This will lead to a much slower start but a faster computation time (that is the same as IEEE FP32 in Table VIII). Table IX summarizes the results of this evaluation.

6) Discussion: From Tables IV and VII we can see how we introduced our PPU^{light} component with a minimal

TABLE IX: Trade-off between load&decompression time of the network in Figure 9. Decompression time was obtained from the real hardware implementation (Table VII).

	Time (s)	DNN size (bytes)	Compression
IEEE FP32	-	224894	-
posit$\langle 16, 0\rangle$	51.5s	112874	**1.99**
posit$\langle 8, 0\rangle$	49.8s	56864	**3.95**

overhead in terms of power consumption (on the FPGA component), utilisation of FPGA resources and no impact on operating frequency. Moreover, the seamless integration did not impact at all the performance of IEEE FP32 computations. We reported an overhead in posit compression and decompression at computation time (theoretically adding two instructions for each sum or multiplication operation). As shown in Table VIII the trade-off between compression and processing time can offer a compression up to a factor 4 with a slow down of 2.5 in terms of processing time.

This kind of trade-off leads us to propose a different approach: we decided to decompress the entire neural network at the very beginning of the execution and then using only IEEE FP32 numbers.
Doing so, we could get the processing time of pure FP32 approach (as in Table VIII) without the overhead of runtime compression by spending around $50s$ at the beginning of the execution for a single network decompression.

VIII. Conclusions

In this paper we described an extension for the RISC-V ISA that implements the conversion between 8 or 16-bit posits and 32-bit IEEE Floats or fixed point formats. Instead of altering the already existent floating point units in the ARIANE core we proposed a seamless integration of the PPU^{light} unit in the core pipeline. We obtained a minimal overhead in terms of power consumption, utilisation and circuit latency. Therefore, performance of pure IEEE FP32 were not impacted. The use of the PPU^{light} leads to a compression of data up to a factor 4 with a very little degradation in computation accuracy when using posit$\langle 8, 0\rangle$ (as proven in previous articles [15, 16]), but with an overhead in terms of processing time. Finally we proposed a flexible approach to using posit compression, being able to decompress the entire neural network only at the beginning without suffering from the runtime compression/decompression overhead. As we reported, there is a clear overhead when employing conversions during computations. Future works will assess this aspect, bringing more arithmetical operations to the PPU, combining the light approach to a more complete one.

Acknowledgment

Work partially supported by H2020 projects (EPI grant no. 826647, https://www.european-processor-initiative.eu/ and TEXTAROSSA grant no. 956831, https://textarossa.eu/) and partially by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

References

[1] "RISC-V ISA Specification," https://riscv.org/specifications/isa-spec-pdf/, (Accessed 2020-03-11).

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: Base user-level ISA," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

[3] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[4] "RISC-V History," https://riscv.org/risc-v-history/, Accessed May 28th, 2020.

[5] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *In Proc. of teh 31st Conference on Neural Information Processing Systems (NIPS'17)*, 2017, pp. 1742–1752.

[6] V. Popescu, M. Nassar, X. Wang, E. Tumer, and T. Webb, "Flexpoint: Predictive numerics for deep learning," in *In Proc. of the 25th IEEE Symp. on Computer Arithmetic (ARITH'18)*, 2018, pp. 1–4.

[7] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks," in *2019 IEEE 26th Symp. on Computer Arithmetic (ARITH'19)*, 2019, pp. 88–91.

[8] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan, "DLFloat: A 16-b floating point format designed for deep learning training and inference," in *2019 IEEE 26th Symp. on Computer Arithmetic (ARITH'19)*, 2019, pp. 92–95.

[9] "Nvidia TensorFloat19." [Online]. Available: https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/

[10] J. Johnson, "Rethinking floating point for deep learning," *CoRR*, vol. abs/1811.01721, 2018. [Online]. Available: http://arxiv.org/abs/1811.01721

[11] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[12] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep positron: A deep neural network using the posit number system," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1421–1426.

[13] S. H. Fatemi Langroudi, Z. Carmichael, J. Gustafson, and D. Kudithipudi, "Positnn framework: Tapered precision deep learning inference for the edge," in *2019 IEEE Space Computing Conference (SCC)*, 07 2019, pp. 53–59.

[14] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[15] M. Cococcioni, F. Rossi, E. Ruffaldi, S. Saponara, and B. Dupont de Dinechin, "Novel arithmetics in deep neural networks signal processing for autonomous driving: Challenges and opportunities," *IEEE Signal Processing Magazine*, vol. 38, no. 1, pp. 97–110, 2021. [Online]. Available: 10.1109/MSP.2020.2988436

[16] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Fast approximations of activation functions in deep neural networks when using posit arithmetic," *Sensors*, vol. 20, no. 5, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/5/1515

[17] S. Tiwari, N. Gala, C. Rebeiro, and V. Kamakoti, "PERI: A posit enabled RISC-V core," *CoRR*, vol. abs/1908.01466, 2019. [Online]. Available: http://arxiv.org/abs/1908.01466

[18] R. Jain, N. Sharma, F. Merchant, S. Patkar, and R. Leupers, "CLARINET: A RISC-V based framework for posit arithmetic empiricism," *CoRR*, vol. abs/2006.00364, 2020. [Online]. Available: https://arxiv.org/abs/2006.00364

[19] E. Ruffaldi, "The cppPosit library," https://github.com/eruffaldi/cppPosit.

[20] "Spike, a RISC-V ISA Simulator," https://github.com/riscv/riscv-isa-sim, Accessed July 7th, 2020.

[21] 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. on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[22] "European Processor Initiative: Posit-based tinydnn," https://gitlab.version.fz-juelich.de/epi-wp1-public/tinyDNN.

[23] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[24] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The German Traffic Sign Recognition Benchmark: A multi-class classification competition," in *In Proc. of the IEEE International Joint Conference on Neural Networks (IJCNN'11)*, 2011, pp. 1453–1460.
[25] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *CoRR*, vol. abs/1905.11946, 2019. [Online]. Available: http://arxiv.org/abs/1905.11946

Marco Cococcioni (SM'12) received the Laurea degree in 2000 and the Diploma degree in 2001 in Computer Engineering from University of Pisa and Scuola Superiore S. Anna, respectively, both with magna cum laude. In 2004 he earned the Ph.D. degree in Computer Engineering at the University of Pisa. After working as a post-doc in the same department, in 2010-2011 he spent two years as Senior Visiting Scientist at the NATO Undersea Research Centre (now CMRE) in La Spezia, Italy. For his collaboration with CMRE he obtained the NATO Scientific Achievement Award in 2014. Since 2016 he is an Associate Professor at the Department of Information Engineering of the University of Pisa. He is in the editorial board of several journals indexed by Scopus. He is member of three IEEE task forces: Genetic Fuzzy Systems, Computational Intelligence in Security and Defense, and Intelligent System Application. Prof. Cococcioni has co-authored more than 100 contributions to international journals and conferences and he is a Senior Member of both IEEE and ACM (Association for Computing Machinery). He has been involved in two H2020 European Projects (EPI and TEXTAROSSA).

Federico Rossi is a PhD student of the Information Engineering Department at University of Pisa. In 2019 he received his Master Degree in Computer Engineering *magna cum laude*. He is currently involved in the European Processor Initiative (EPI) project. His research topics include alternative real number representations and their applications to Deep Neural Networks for the automotive environment.

Emanuele Ruffaldi (SM'18) is senior software engineer at MMI S.p.A. (IT) working on robotic assisted microsurgery. Formerly he has been Assistant Professor at Scuola Superiore Sant'Anna in the Perceptual Robotics laboratory, Pisa, Italy. His research interests are in the field of machine learning for HRI and embedded artificial intelligence. He is Senior IEEE Member and has served IEEE as Publicity Chair for the Haptics TC.

Sergio Saponara (SM'13) is Full Professor of Electronics at University of Pisa, where he got Master degree cum laude and Ph.D. degree. In 2012 he was a Marie Curie Research Fellow in IMEC. He is an IEEE Distinguished Lecturer and co-founder of special interest group on IoT of both IEEE CAS and SP societies. He is the director of I-CAS lab, of Crosslab Industrial IoT, of the Summer School Enabling Technologies for IoT. He is associate editor of several IEEE and Springer Journals. He co-authored more than 300 scientific publications and 18 patents. He is the leader of many funded projects by EU and by companies like Intel, Magneti Marelli, Ericsson and PPC.