Generating Posit-Based Accelerators With High-Level Synthesis

Raul Murillo, Alberto A. Del Barrio, *Senior Member, IEEE*, Guillermo Botella, *Senior Member, IEEE*, and Christian Pilato, *Senior Member, IEEE*

***Abstract*— Recently, the posit number system has demonstrated a higher accuracy over standard floating-point arithmetic for many scientific applications. However, when it comes to implementing accelerators for these applications, the tool support for this arithmetic format is still missing, especially during the high-level synthesis (HLS) step. In this paper, we incorporate the posit data type into the high-level synthesis (HLS) design process, so that we can generate the register-transfer level (RTL) implementation directly from a given behavioral specification, but using posit numbers instead of the classical floating-point notations. Our evaluations show that, even if posit-based circuits require more area than their floating-point counterparts, they offer higher accuracy when using the same bitwidth. For example, using posit arithmetic can reduce computation errors by about two orders of magnitude when compared to using standard floating-point numbers. Our approach also includes an alternative to mitigate the high overheads of the posits and broadening the potential use of this format. We also propose a hybrid scheme that uses posit numbers only in the private local memory, while the accelerator operates in the classic floating-point notation. This solution is useful when the designers want to optimize local memories and data transfers, but still use legacy HLS tools that only support traditional floating-point notations.**

***Index Terms*— HLS, computer arithmetic, posit, floating-point.**

I. Introduction

In recent years, data-intensive applications have permeated many areas of computing due to the rise of deep learning and the increasing demand for resolution in physical simulations (e.g., molecular dynamics and weather forecasting). The exceptional performance achieved by these applications over the last few years has been possible mainly due to the large increase in available datasets and computational resources to analyze them. However, this trend of increasing computational models clashes with the end of Moore's law and Dennard scaling. Therefore, maintaining performance improvement nowadays to enable new software capabilities, such as physical simulations, is both important and challenging. According to recent studies [1], an interesting research direction in computer architecture is the use of domain-specific architectures (DSAs), a class of processors tailored to a specific domain or class of applications, or even more specialized processors such as GPUs, field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs) (also known as accelerators) [2]. Such accelerators are closely tailored to the needs of a given application and can therefore achieve higher performance and higher energy efficiency than general-purpose CPUs [3], [4], [5], [6].

The use of FPGA accelerator cards has become more common in the last years thanks to the progress and availability of integrated tool flows based upon HLS, which allow generating highly-optimized hardware starting from a source code with a higher level of abstraction [7], [8]. Nowadays, most of the algorithms are written in high-level programming languages like C/C++, where the functional execution of the program is much faster and simpler than the counterpart register-transfer level (RTL) simulation. Therefore, the HLS has boosted the hardware/software co-design, making it possible to automate the synthesis of new accelerators and to map complex workloads onto domain-specific architectures [9].

Alongside the development of DSAs, new arithmetic formats have also emerged recently in an attempt to mitigate the effects of the end of Moore's law and Dennard scaling [10], [11], [12], [13]. In the area of scientific and high-performance computing (HPC), the IEEE 754™ standard for floating-point arithmetic [14] has been for decades the format used for representing real numbers in this kind of applications. Nonetheless, the appearance of the disruptive posit™ arithmetic [15] in 2017 has shaken the board. This novel way of representing real numbers mitigates some of the shortcomings that the IEEE 754 standard presents (such as dealing with signed zero, the multiple-bit patterns wasted for indicating Not a Number (NaN) exceptions, or the fact that reproducibility of results is not guaranteed), but is also able to represent a wider range of values and provide more accurate computations than floating-point numbers using the same number of bits. However, these benefits come at a cost—when implemented

Manuscript received 10 April 2023; revised 5 July 2023; accepted 24 July 2023. Date of publication 2 August 2023; date of current version 29 September 2023. This work was supported in part by MCIN/AEI/10.13039/501100011033 under Grant PID2021-123041OB-I00; in part by the "ERDF—A Way of Making Europe;" in part by the 2020 Leonardo Grant for Researchers and Cultural Creators, BBVA Foundation, under Grant PR2003_20/01; in part by CM under Grant S2018/TCS-4423; in part by the EU Horizon 2020 Programme under Grant 957269; and in part by the HiPEAC6 Network funded by the EU Horizon 2020 Programme under Grant ICT-2019-871174. This article was recommended by Associate Editor X. S. Zhang. *(Corresponding author: Raul Murillo.)*

Raul Murillo is with the Department of Computer Architecture and Automation, Faculty of Physics, Complutense University of Madrid, 28040 Madrid, Spain (e-mail: ramuri01@ucm.es).

Alberto A. Del Barrio and Guillermo Botella are with the Department of Computer Architecture and Automation, Faculty of Computer Science, Complutense University of Madrid, 28040 Madrid, Spain (e-mail: abarriog@ucm.es; gbotella@ucm.es).

Christian Pilato is with Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20133 Milan, Italy (e-mail: christian.pilato@polimi.it).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2023.3299009.

Digital Object Identifier 10.1109/TCSI.2023.3299009

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

in hardware, posits require more hardware resources, time, and power than the corresponding floating-point units [16], [17]. The real advantage of using posit arithmetic comes from using fewer bits, thereby saving storage, memory bandwidth, and energy without sacrificing accuracy. Changing arithmetic, however, affects the entire hardware-software stack from chip design to the application.

Prior studies have extensively examined the hardware costs associated with posit arithmetic in the context of individual arithmetic operations; however, there remains a dearth of information regarding the interconnection and integration of these components as a whole. In this paper, we present a proof-of-concept implementation of an end-to-end methodology that leverages the properties of posit arithmetic to create FPGA-based systems and accelerate numerical kernels in scientific computations. The major contributions of our work can be summarized as follows:

- We design an **RTL library of posit operators** that adds support for all the basic operations required in HLS. The library is parameterizable for multiple bitwidths, including 32 and 64 bits, and target frequencies.
- We integrate the **proposed posit library into an open-source HLS flow**, so accelerators based on this arithmetic format can be automatically generated from the same source code as floating-point applications.
- We perform an **evaluation of the proposed solutions** at both single operator and application-level with PolyBench. Experimental results demonstrate that, under the same bitwidth, posit arithmetic reduces the error of computations around two orders of magnitude, with an overhead of about 75% more area and 50% more latency with respect to floating-point arithmetic. In terms of hardware resources, the proposed 32-bit posit designs require, on average, 1.46× more LUTs, 1.73× more FFs, and 1.30× more cycles than the corresponding floating-point designs. Similar figures are found for the 64-bit case.
- We implement a **hybrid scheme that uses posit arithmetic in memory**, while the accelerator logic remains in floating-point format. This allows legacy HLS tools without support for alternative formats to leverage posit arithmetic; data can be stored in memory using a lower bitwidth posit format while preserving the accuracy of computations. Evaluation results show that this approach allows for reducing the error of computations with a small area overhead and a negligible increase in latency.
- We **compare the proposed HLS flow with previous works**. FPGA synthesis results show that the proposed posit designs provide lower overhead than those proposed in previous works when compared to corresponding accelerators designed for floating-point arithmetic.

The remainder of this paper is organized as follows: Section II presents the basics of posit arithmetic and HLS. The proposed methodology and strategies for integration of posit operators into an HLS tool are presented in Section III. Section IV explains the integration of the posit RTL library into the tool, while the necessary changes in the design flow related to memory allocation and integration of new data formats are detailed in Section V. The application benchmarks, implementation results and analyses are presented in Section VII. Finally, Section VIII concludes this paper.

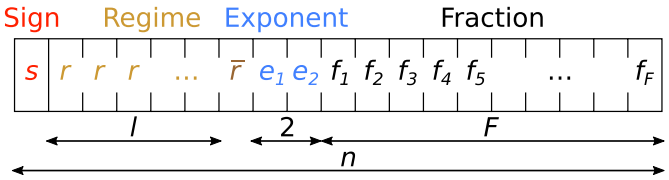


Fig. 1. Generic n-bit posit binary encoding.

II. Background

A. Posit Notation

Posit arithmetic [15] was introduced in 2017 as an alternative to the ubiquitous IEEE 754 floating-point standard to represent and operate with real numbers. Posit numbers (posits in short) provide reproducible results across platforms and very few special cases. Furthermore, they do not support overflow or underflow, which reduces the complexity of exception handling.

A posit number configuration is usually defined by two parameters $\langle n, es \rangle$—the total bitwidth n and the exponent size es, i.e., the number of bits reserved for the exponent field. Although in literature [15], [18], [19] 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 version of the Posit Standard [20], 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 [21]. Therefore, such a configuration will be used in the rest of the paper, denoting it as Posit$N$, with N being the total size and the length of the exponent field fixed to 2 bits.

Posit arithmetic only distinguishes two special cases: zero and Not a Real (NaR), which are represented as `0...0` and `10...0`, respectively. The rest of the bit patterns are used to represent different real values, which are composed of four fields as shown in Fig. 1: a sign bit (s), several bits that encode the regime value (k), up to $es = 2$ bits for the exponent (e), and the remaining bits for the normalized fraction (f). The regime is a sequence of l identical bits (r) finished with a negated bit ($\bar{r}$) that encodes an extra scaling factor k given by (1),

$$k = \begin{cases} -l & \text{if } r_0 = 0 \\ l - 1 & \text{if } r_0 = 1 \end{cases}. \tag{1}$$

As this field does not have a fixed length, it may cause the exponent to be encoded with less than 2 bits, even with no bits if the regime is wide enough. The same occurs with the fraction, which must be normalized with respect to the size of the fraction field (2^F). The variable length of the regime allows posit arithmetic to have more fraction bits for values close to ± 1 (which increases the accuracy within that range), or to have fewer fraction bits for the sake of more exponent bits for values with large or small magnitudes (increasing this way the range of representable values).

The real value p of a generic posit is given by (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 case of posits, it is considered to be 1 if the number is positive, or -2 if the number is negative [21], [22].

$$p = (1 - 3s + f) \times 2^{(1-2s)\times(4k+e+s)}. \quad (2)$$

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 [22]. Another interesting feature of posit arithmetic is that it includes fused operations. In operations of this kind, which take more than two operands, intermediate results are accumulated in a larger register called *quire*, avoiding intermediate roundings and thus providing even more accurate results [23], [24].

Although posit arithmetic was designed to have similar circuitry to the floating-point format, the variable length of the fields and the signed hidden bit of the fraction requires redesigning some of the logic when implementing posit operators. However, such an effort might be compensated by the benefits of using posit arithmetic—its higher accuracy, when compared with standard floating-point, can reduce the bitwidth of the data and operations of scientific computations without sacrificing the accuracy of the results, with all the benefits this entails at the hardware level [25].

B. High-Level Synthesis

HLS is an automated design process that, starting from the high-level description of an application, an RTL component library, and specific design constraints, finds an RTL structure that implements the given behavior [7], [8]. The main steps an HLS tool executes are the following:

1) **Compilation**. HLS always begins with the compilation of the functional specification. This first step transforms the high-level input description (typically ANSI C/C++) into a formal representation. Usually, it includes several code optimization, such as data dependency solving, dead-code elimination, or loop transformations.
2) **Allocation**. The type and the number of hardware resources (such as functional units (FUs), storage, or connectivity components) are defined according to the design constraints. Such components are selected from the RTL component libraries. The use of numerical representations with lower bitwidth has a direct impact on the hardware resources of the final design, since less memory and smaller FUs will be allocated instead.
3) **Scheduling**. All operations required in the specification model must be scheduled into control steps. This is done considering the functional components and the operation priorities and dependencies. Operations can be chained, or can be scheduled to execute in parallel provided there are no data dependencies and sufficient available resources. Again, different numerical representations might result in different scheduling results. Typically, reducing the bitwidth of the signals also reduces the datapath delay, yielding faster circuits.
4) **Binding**. Within the computed schedule, each variable that carries values across control steps must be bound to a storage unit or register. Variables with non-overlapping life intervals may share the same register. In a similar manner, operations must be bound to the capable FUs, preventing those that execute concurrently from sharing the same resource instance. Register and FU binding also depends on interconnection binding, which introduces the steering logic or connection units (such as buses or multiplexers) to perform transfers from component to component. As one may guess, the impact of numerical representation in the previous steps directly affects the binding stage. Smaller data might require smaller or fewer FUs, registers and hardware resources, in general.
5) **Netlist generation**. The final architecture obtained from the tasks of allocation, scheduling, and binding is translated in an RTL model of the synthesized design in a hardware description language (HDL) like Verilog or VHDL. This process accesses the resource library, which embeds the RTL implementation of each resource, and is target-dependent, so hardware descriptions may differ for different technologies.

C. Related Work

Our work is focused on the integration of posit arithmetic into the HLS flow. For that purpose, we start from the implementation of the required RTL components and then certain steps of the synthesis process are modified to accommodate such an arithmetic format. At the time of writing this paper, there are few published works exploring the use of posit arithmetic in HLS. Authors in [16] introduced MArTo, a C++ library for posit arithmetic compliant with Xilinx Vivado HLS. It is built on a custom internal representation, and supports addition, subtraction and multiplication of posit datatypes, as well as the exact accumulation of posit products. Although this library is designed from a higher abstraction level, it requires adapting the source codes to use it, and its usage is currently limited to the aforementioned operations. Moreover, experiments in [16] evaluated just the performance of standalone posit operators, but no results on posit-based accelerators generated from HLS are given. Similarly, previous works [17], [18], [26], [27] also compare standalone posit and floating-point operators. They all conclude that even posits provide more accurate results under the same bitwidth, posit units are almost twice as large and twice as slow as the corresponding IEEE units.

On the other hand, some works [28], [29], [30] have evaluated the effects of using posit arithmetic not for computation, but just for data storage in many different applications, including deep learning or climate modeling. Since posits can be as accurate as floats with a fewer number of bits, data can be compressed in a lower-precision posit format with negligible effect on the accuracy. This results in less memory storage required per operand, so higher computing bandwidths can be achieved, or hardware requirements can be reduced this way.

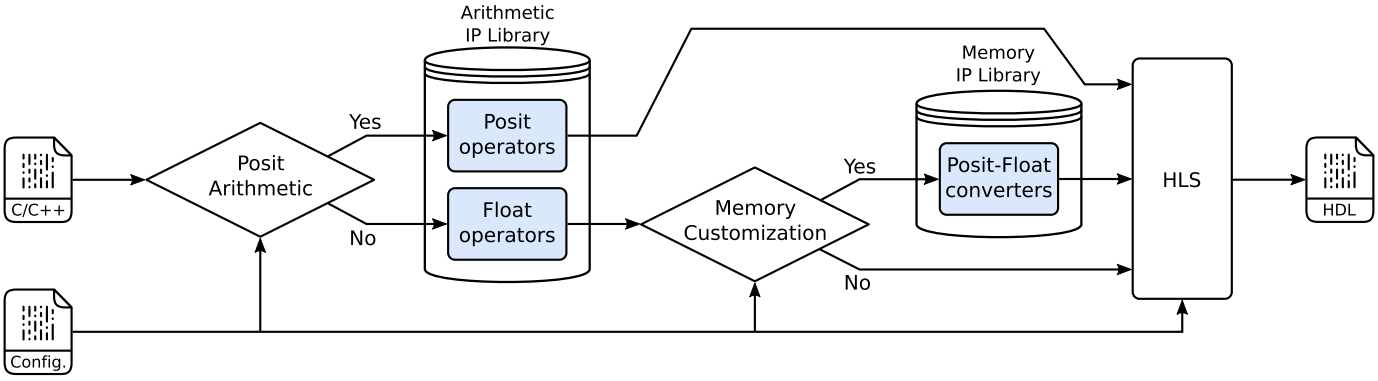


Fig. 2. HLS flow with support for posit arithmetic and memory customization.

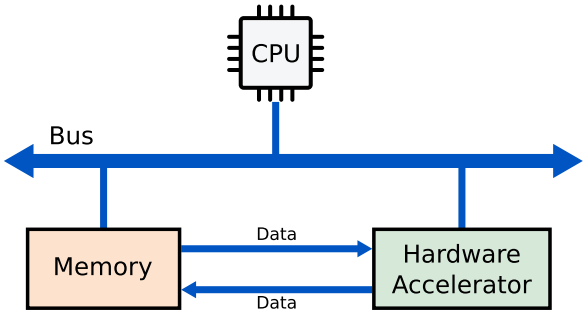
III. Proposed Approach

In this paper, we consider Bambu, an open-source HLS research framework [31], [32]. The tool receives as input a behavioral description of the specification, written in C/C++ language, and generates the HDL description of the corresponding RTL implementation as output, which is compatible with commercial RTL synthesis tools. In addition, it is designed in an extremely modular way and supports floating-point operations through FloPoCo, a generator of arithmetic floating-point cores [33]. The choice of Bambu as HLS tool for this work is motivated by its open-source philosophy and its integration with FloPoCo.

A scheme of the proposed tool flow is depicted in Fig. 2. To add support for posit arithmetic to the HLS flow, we designed an RTL library of posit operators based on FloPoCo, and integrated it within Bambu. The design flow of the tool was extended to handle such an additional library without the need of modifying the C/C++ source code. From the point of view of the programmer, the use of posit arithmetic for the computation of real numbers should be as transparent as selecting between single or double-precision floating-point.

Although certain implementation aspects of the proposed approach are closely tied to the aforementioned software technologies, the fundamental concepts of our approach can be extended and potentially applied to other HLS tools. This would require 1) providing a posit operators library that provides equivalent functionality to floating-point operations (either at the RTL or software level), and 2) implementing an arithmetic back-end selection/substitution mechanism that preserves the memory infrastructure of the device.

Furthermore, prior research has demonstrated that storing data in posit format can lead to reduced memory requirements while preserving accuracy [18], [28]. Such an approach is also useful when the designers want to customize communication by reaping the benefits of posits but they still want to use legacy HLS tools that do not support non-standard formats. In light of this observation, this study also aims to investigate the implications of adopting such a design alternative, specifically examining the impact of employing posit numbers in memory with accelerators that do not inherently support this arithmetic format. By doing so, we intend to gain insights into the potential benefits and challenges associated with incorporating posit-based memory in systems utilizing non-posit arithmetic accelerators. Fig. 3 illustrates the different strategies we evaluated in this work. Case #1 represents the classical



Device	Case #1	Case #2	Case #3
Memory	Float	Posit	Posit
Accelerator			Float

Fig. 3. Scheme of the different approaches that use float and/or posit arithmetic in memory and hardware accelerator.

floating-point approach, and it serves as a reference point. It is important to note that in case #3, where just floats are used in the accelerator, certain data conversion processes would be necessary to ensure compatibility with the data originally stored in posit format. For each case, we consider both 32 and 64-bit precisions.

IV. Library of Posit Operators

HLS tools transform a high-level specification into an RTL design. Realistic hardware implementation thus requires the conversion of floating-point and integer variables into bit-accurate data types of a specific length (not a standard byte or word size, as in software) with acceptable computation accuracy. This is done in the resource allocation step of the HLS process (see Fig. 4 below), and usually requires RTL libraries to map the variables and structures in higher abstraction level to specific hardware components. Bambu has support for floating-point operators through FloPoCo [33], [34], an open-source C++ framework that generates floating-point arithmetic datapaths in synthesizable VHDL from the operator specifications.

Using FloPoCo, we implemented a library of posit operators that includes the basic arithmetic operations—addition, subtraction, multiplication and division—, as well as units to perform the comparison of operands and conversion with integer arithmetic. More precisely, we rely on the posit arithmetic units designed in our previous works [17], [35], [36],

which have empirically demonstrated to be efficient in terms of performance and energy. The advantage of designing posit operators with FloPoCo is that this tool can generate such operators for any given bitwidth. In this work, we generate 32 and 64-bit accelerators, so the same design is used for the different bitwidths; FloPoCo automatically adjusts the size of internal wires and signals according to the specified data width. The tool also allows pipelining the FUs automatically according to the specified parameters and target frequency.

In order to verify that the proposed architectures are correct, exhaustive tests were generated with the reference software library Universal [37] for 8, 10 and 12-bit posits, as well as random/corner case tests for 16, 32 and 64-bit posits. All these tests were successful.

The implementation of parameterizable designs with FloPoCo makes the creation of the library of posit operators independent from the design of the final accelerator. After the compilation step, Bambu detects all the arithmetic operations that are required in the source code. When a floating-point operation is detected, Bambu calls FloPoCo with the corresponding parameters to generate the required operator according to the design constraints. At this point is were we can select a different implementation from the IP library, for example, a posit adder instead of a floating-point adder circuit. However, the HLS tool must be aware of the latency of such a component in order to properly generate the entire accelerator, as will be discussed in Section VI.

Previous works also used the reference tool FloPoCo to design basic adders and multipliers [17], approximate FUs [27], or even fused operations with accumulator [16], [23]. However, such works just focus on the design and performance of each individual component. In this work, we additionally developed comparison and conversion units. Thus, the HLS flow of a program involving basic arithmetic operations (such as addition, multiplication, division, and square root) can rely solely on the proposed posit RTL library, eliminating the need for additional floating-point operations within the HLS flow.

In addition, we made sure that each of the operators that constitute the proposed library is at least as efficient as those proposed by previous works in terms of performance and hardware resources.

V. Memory Customization

Bambu allows using a wide variety of memory allocation policies and memory accesses. The HLS tool automatically infers the memory infrastructure according to the constraints, commands and types of the operands (e.g., integer, float, etc.). However, the posit data type is neither available in high-level languages nor commercial hardware devices, so some extra effort is needed to customize the memory with posit format. In this work, we consider the 32-bit and 64-bit precisions of the different arithmetics. Therefore, float (32-bit) and double (64-bit) C types are used as replacements for Posit32 and Posit64, respectively, as they have the same bitwidth, and therefore memory accesses do not change. Using a different number of bits would require changing either the memory inside the accelerator or the compiler (so that it understands a different kind of floating-point operation), which is out of scope.

Algorithm 1 Posit to IEEE 754 float conversion

Require: $x \in \text{Posit}N$
Ensure: $y \in \text{Float}\langle E, M\rangle$
1: $sign \leftarrow x[N-1]$
2: $val \leftarrow x[N-2:0]$
3: **if** $val = 0$ **then**
4: **if** $sign = 0$ **then**
5: $y \leftarrow 0$
6: **else**
7: $y \leftarrow NaN$
8: **else**
9: **if** $sign = 0$ **then**
10: $abs\_val \leftarrow val$
11: **else**
12: $abs\_val \leftarrow -val$ ▷ Take 2's complement
13: $regime, exp, frac \leftarrow$ extract_fields(abs_val)
14: ▷IEEE float exponent $bias = 2^{E-1} - 1$ ◁
15: $biased\_exp \leftarrow \{regime, exp\} + bias$
16: $y \leftarrow \{sign, biased\_exp, frac\}$
17: ▷ Pad y with 0's to the right if necessary ◁
18: **return** y

Posit arithmetic claims to have higher accuracy using the same number of bits, or sufficient accuracy using fewer bits. When storing data in fewer bits, the total memory footprint is reduced, reaching lower power and energy consumption. But also, when transferring data from an external memory to devices such as FPGAs, using smaller bitwidths might allow transferring more data simultaneously under the same bandwidth, increasing the SIMD vectorization and achieving higher throughput.

In this paper, we compare the effects of using 32-bit posits in memory with regard to 64-bit floats. While this clearly halves the size of the external memory, it is important to evaluate its impact on the accuracy of the results.

In addition to this, we propose a hybrid scenario in which the memory is customized with posit format while the accelerator logic is kept unchanged in floating-point. The usefulness of this approach lies in storing the data in a posit format with a smaller bitwidth than the floating-point format in which the computations are performed. From the memory point of view, this is not different from the case when all data is in posit format. However, under this approach, the input data must be converted into floating-point format before performing computations in the accelerator, and the results must be stored in memory using posit format. The conversion between N-bit posits and floating-point numbers with E exponent bits and M fraction bits is depicted in Algorithm 1, and the reverse analogous process is done for float-to-posit conversion.

VI. Posit-Aware High-Level Synthesis

Performing HLS with a custom arithmetic format such as posit is not straightforward, since the data format and the RTL components interfere in multiple steps of the HLS design flow.

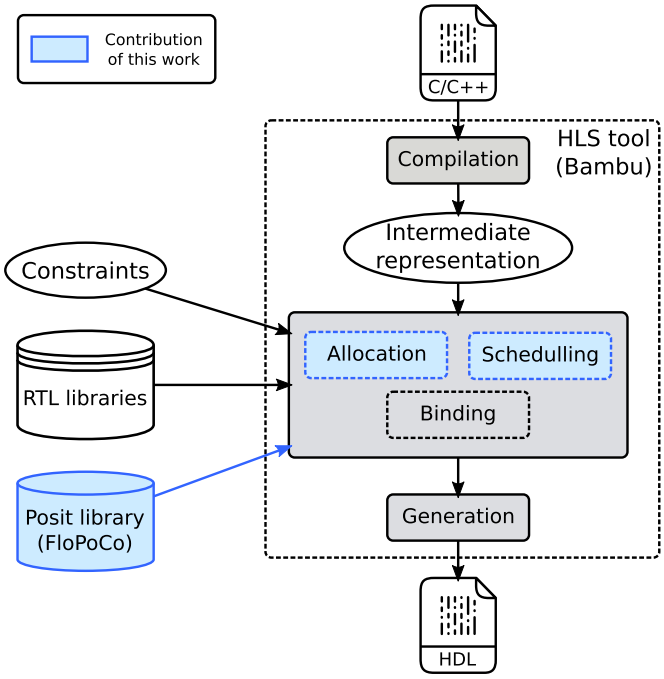


Fig. 4. HLS design flow.

As mentioned in Section II, the synthesis process starts with the compilation of the C/C++ source code. However, the posit data type is not supported in such programming languages or compilers, in contrast to the float or double types. For that reason, we decided to keep unchanged that part of the process and perform further modifications in the subsequent stages of the synthesis. More precisely, we included an option within Bambu that indicates whether floating-point data in the high-level specification must be considered as posit arithmetic. To accomplish this, it is necessary to modify the allocation and scheduling stages of the HLS flow, as indicated in Fig. 4.

Regarding the allocation stage, we made modifications to incorporate the posit RTL library. This involved ensuring that the appropriate library, whether it is the float or posit library, is called based on the specified configuration of the HLS. To facilitate the usage of different back-end arithmetic libraries for end users, we introduced the option `-flopoco=posit` within Bambu. When this posit flag is activated, floating-point operations in the source code are translated in the corresponding RTL for posit operations defined in FloPoCo. The allocation step maps them on the set of available FUs: their characterization includes information, such as latency, area, and the number of pipeline stages. The pipeline of the operators is automatically driven by FloPoCo according to the specified design constraints such as the clock period. In addition to FUs, also memory resources are allocated, in this case as detailed in Section V. Along the entire HLS process, especially during resource allocation and scheduling, it is necessary to have certain information about each component in order to perform synthesis optimizations. For this reason, Bambu adopts a pre-characterization approach. The latency and resource occupation of every posit FU are obtained by synthesizing them for multiple combinations of bitwidths, frequencies and target devices. This has a direct impact on the scheduling stage of the HLS design flow, and is a common approach adopted by other HLS tools such as LegUp [38]. It is worth noting that this pre-characterization needs to be performed only once, and allows performing aggressive optimizations. Then, such characterizations are used to select the most appropriate configuration for each posit FU and schedule the operations according to the design constraints. As a result of this work, from the user perspective, performing HLS in posit arithmetic with Bambu just requires activating the posit flag, without modifying the C/C++ source codes that use floating-point data.

As already mentioned, this work also proposes the use of posit arithmetic in memory (second case in Fig. 3), while keeping the logic of the accelerator in floating-point. Using posits as lossy compressed information storage (in 32-bit precision) can reduce the amount of data transferred up to a factor 2 with respect to the double-precision accelerator while maintaining decent accuracies. Also, such an approach is compatible with commercial HLS tools that do not support posit arithmetic. When the data in memory is in a different format or precision with respect to the logic of the accelerator, data conversion must be performed before and after performing computations in the accelerator. For that reason, posit-to-float and float-to-posit units are designed to convert input data and output results from the accelerator, respectively. Such FUs are implemented in FloPoCo (as well as the rest of the units of the proposed library), so the same parameterized design is used for different bitwidths. In Bambu, it is possible to map C functions to hand-written HDL modules, which makes this process straightforward. What is more, such modules could be also appended to accelerators generated using floating-point, which makes this a suitable option for those HLS tools without support for posit arithmetic. However, when using this hybrid approach, the corresponding conversions must be considered in the HLS scheduling stage, as data are converted before and after the real computation.

One may notice that 32-bit posits have higher accuracy (or precision bits) than 32-bit floats, but lower than 64-bit floats. Thus, conversion from a more accurate format to a less accurate one requires handling proper rounding to mitigate error. Reducing the number of bits in memory has clear benefits in the design of accelerators—higher computing bandwidths can be achieved while reaching lower power and energy consumptions—, but this comes at a cost—additional hardware is necessary for data conversion, and error might increase due to rounding.

VII. Hardware Evaluation

While previous works have already analyzed the hardware cost of posit arithmetic in the context of individual arithmetic operations, there is not much information about how all these pieces fit together. In this section, we analyzed the impact of the different arithmetic formats in the design of hardware accelerators for real applications.

A. Experimental Setup

The effects that each of the schemes depicted in Fig. 3 has in terms of hardware resources and latency when performing HLS, were evaluated. It is also important to have an understanding of the accuracy of each approach. For cases #1, #2

and #3, both 32 and 64-bit precision were considered. When the memories and accelerator logic have different representations, 32-bit posit format (Posit32) is used on the memory side.

To compare the performance of the different approaches, we extracted a series of numerical benchmarks from the PolyBench 4.2 suite,[1] which includes many common algorithms in fields such as linear algebra, data mining, and image processing. In particular, we have chosen the following representative benchmarks:

- 3mm: Linear algebra kernel that consists of three matrix multiplications $G = ((AB)(CD))$.
- cholesky: Cholesky decomposition of a positive-definite matrix A into a lower triangular matrix L such that $A = LL^T$.
- covariance: Computes the covariance of N data points, each with M attributes.
- fdtd-2d: Simplified finite-difference time-domain method for 2D data. It models electric and magnetic fields based on Maxwell's equations.
- gemm: General matrix-matrix product from BLAS, $C = \alpha AB + \beta C$.
- ludcmp: LU decomposition followed by forward and backward substitutions to solve a system of linear equations.

PolyBench implements each benchmark in a single file, with some header parameters and a series of compile-time directives, including the data format and dataset size. We configured PolyBench to use 32-bit and 64-bit precision for all our experiments. For numerical accuracy evaluation, the code structure was kept unchanged, including the initialization phase which populated the input data to the algorithms, just modifying the data format and test size for different experiments.

For hardware evaluation, the focus is on the HLS generated accelerators for main kernel computation. Thus, we eliminate the initialization phase when performing synthesis, while the kernel core remains unchanged. Instead, we add a wrapper around the kernel that creates a local copy of the data in the accelerator. This has two consequences. First, when working with arrays, it is faster to access the accelerator's local memory rather than the host machine's memory, which significantly reduces the latency of the accelerators. On the other hand, in case #3, where the memory data is assumed to be in posit format but the computation is performed in floating-point, it is necessary to convert the data to the latter format before operating, and back to the former at the end of the computation. Although this conversion could be applied each time an operation is performed, in cases such as gemm, where the same piece of data is used for several intermediate computations, this approach has a negative impact on operator latency. However, performing this conversion only once per single datum and storing it in local memory allows to reduce the number of conversions (and clock cycles), at the cost of higher hardware resource cost. To have a fair comparison across the proposed approaches, this approach is considered in all the experiments. Performing data conversions at every single operation might reduce the amount of memory required in the FPGA at the cost of increasing the data transfers between the accelerator and the host device, but such a study is out of the scope of this work.

We performed HLS of each application with Bambu targeting a Xilinx Artix-7 (XC7A100T-1CSG324C) FPGA device. In particular, for the HLS with Bambu we included the options `-no-iob` (so primary ports from the IOB are disconnected, and large arrays can be instantiated in the target device) and `-experimental-setup=VVD` (which provides similar settings for RTL synthesis as the commercial solution Vivado HLS). Under this approach, all objects and internal variables that need to be stored in memory are allocated on BRAMs rather than on external memory. To select a suitable target frequency for the HLS, we conducted detailed tests for individual arithmetic operators targeting different maximum clock frequencies, which allow us to obtain more details in this regard. Xilinx Vivado 2021.2 was used to perform the logic synthesis for the comparison of hardware resources.

To generate floating-point logic for the accelerators, the option `-flopoco=float` was used, so the floating-point FUs are the ones provided by FloPoCo. However, such units are non-compliant with the IEEE 754 standard: although the memory format is in IEEE 754 format, subnormals are flushed to zero to save resources. This could produce inaccurate results in applications that make use of such small-magnitude data. Also, exceptions are handled in a much simpler way as required by the standard, and just a single rounding mode is implemented (round to nearest, ties to even), rather than the five rounding rules defined in the standard. Therefore, it should be kept in mind that a fully IEEE 754-compliant implementation would incur a much higher overhead than the current one. On the other hand, we extended Bambu with the option `-flopoco=posit` to allocate posit FUs in the final accelerator. Such units are fully compliant with the current posit standard [20].

Lastly, it is important to mention that all programs in this evaluation were compiled with the `-O3` optimization option, which applies a standard set of optimizations. By adopting this approach, the focus is placed squarely on the capabilities and limitations of the HLS tool itself, without introducing additional custom optimization strategies. This allows for a clear assessment of the baseline performance achievable through compiler optimizations alone.

B. Numerical Error

Prior to the synthesis evaluation, it is important to ascertain the benefit of each of the encodings proposed in this work in terms of numerical accuracy. In order to evaluate the error of each approach, we performed software simulations of each experiment for multiple dataset sizes ranging from MINI to LARGE. The error is computed by the Frobenius (or element-wise) norm against the result obtained with an extended precision format. Such a metric becomes useful when comparing the precision of different arithmetic formats, as it effectively measures how much two simulations deviate from each other by penalizing large errors and giving less importance to minor differences. In fact, it can be used for either scalars or matrices and vectors, which is the case in the PolyBench applications.

[1] https://sourceforge.net/projects/polybench/

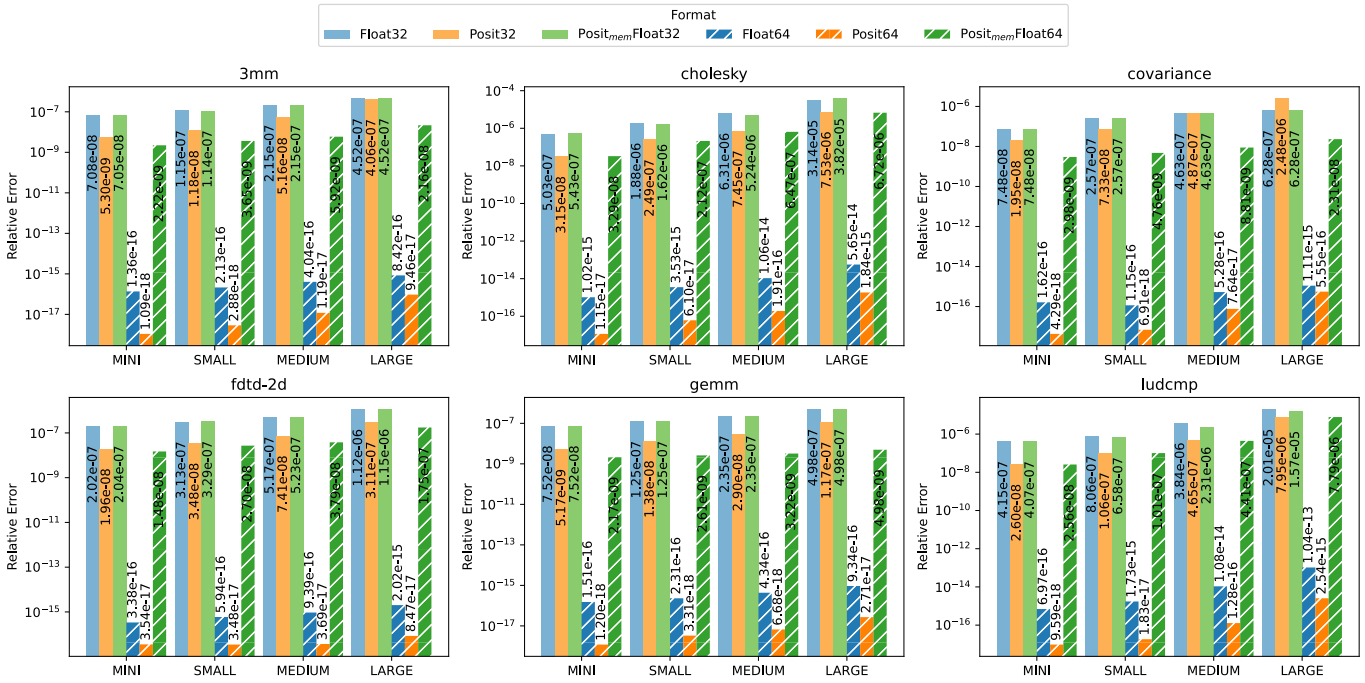


Fig. 5. PolyBench benchmarks error comparison for different number formats.

To obtain these metrics, we compared the results of the three cases depicted in Fig. 3 (under both 32 and 64-bit precision) to the same algorithm computed using the double extended 80-bit format present in x86 processors. The relative error results (with respect to the norm of the baseline) are shown in Fig. 5. Note the logarithmic scale on the Y-axis. The trend in every benchmark is a significantly lower error when using posit numbers. This is up around one order of magnitude for 32 bits, and between two and three orders less in the case of 64 bits, depending on the benchmark.

Case #3, which mixes both arithmetic formats reveals that using Posit32 in memory rather than Float64 and performing computations in the former precision can be a useful option when posit circuitry is not available, and provides less error than using Float32 and even Posit32. On the other hand, results show that, from the point of view of accuracy, there is no benefit in performing computations with a less accurate format than the stored data.

Once the error of the different proposed formats and approaches has been characterized, we will evaluate the performance of specific accelerators using HLS.

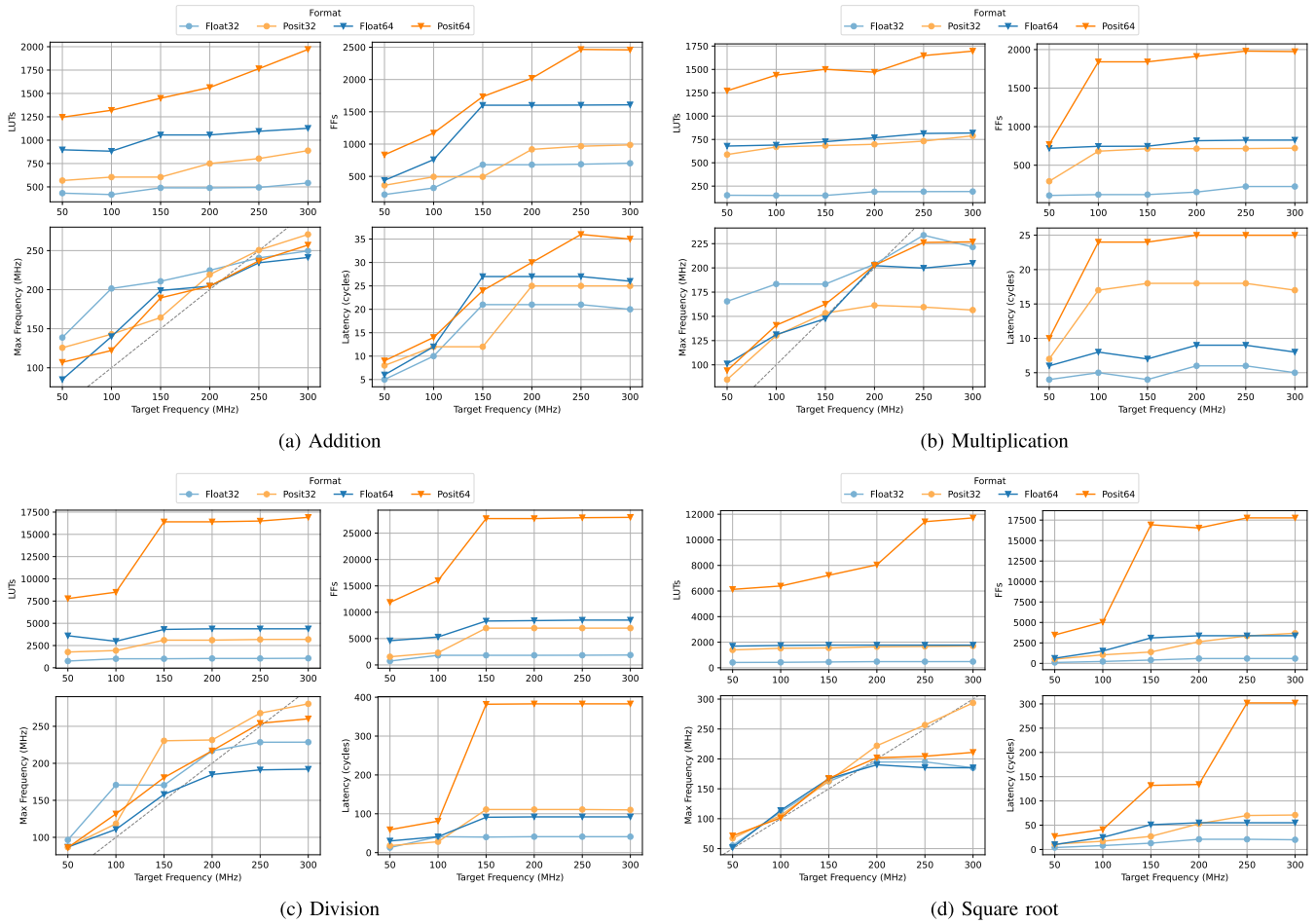
C. Operation-Level Evaluation

Prior to the HLS of the PolyBench applications, a synthesis evaluation of the basic arithmetic operators has been conducted as the initial step in our assessment of the FPGA implementation results. This evaluation aimed to provide a more fine-grained and detailed analysis of the outcomes, isolating as much as possible the library of posit arithmetic operators from the rest of the HLS tool. By focusing on individual operations within the hardware design, we were able to gain in-depth insights into the performance, efficiency, and potential bottlenecks at a granular level. The findings and observations obtained from this operation-level evaluation will guide the subsequent evaluation of complete applications.

The synthesis results for each arithmetic unit, as well as the clock cycles obtained by the RTL simulation, are reported in Fig. 6. Posit adders require about 1.5× hardware resources (LUTs and FFs) than the corresponding float units, while this overhead is between 2× and 6× for the rest of the operators. Nonetheless, the amount of resources required by Posit32 is always fewer than by Float64 units. Regarding the frequency, all the functional units except the Float64 multiplier satisfy the timing target conditions up to 150 MHz. For a target frequency of 200 MHz a few operators violate the timing constraint, and none of them reach 300 MHz. Therefore, 150 MHz is a clear candidate as the target frequency for the HLS of complex applications. Finally, it must be noted how the iterative algorithm used for division and square root has a direct impact on the latency of such units as the target frequency increases, especially for the Posit64 format.

In addition to the resources shown in Fig. 6, HLS results show that, independently of the target frequency, the 32 and 64-bit floating-point multipliers require 2 and 9 DSPs, respectively, and the corresponding posit multipliers make use of 2 and 12 DSPs, respectively. Also, the design of the floating-point division includes a table for fast computation, which requires 7 and 14 extra BRAMs when synthesizing the 32 and 64-bit designs, respectively.

Case #3 proposed in this work considers input data to be in Posit32 format, while the computation done within the accelerator is in floating-point. For this hybrid scenario, input and output data conversion must be done, so it is important to evaluate separately the hardware overhead of such conversions. Synthesis results are reported in Fig. 7. As can be seen, the library of posit converters can be synthesized with Bambu up to 300 MHz seamlessly. In addition, the units exhibit quite low latency (3 cycles or less) when targeting up to 150 MHz.



(a) Addition (b) Multiplication

(c) Division (d) Square root

Fig. 6. Bambu HLS results for basic arithmetic operators.

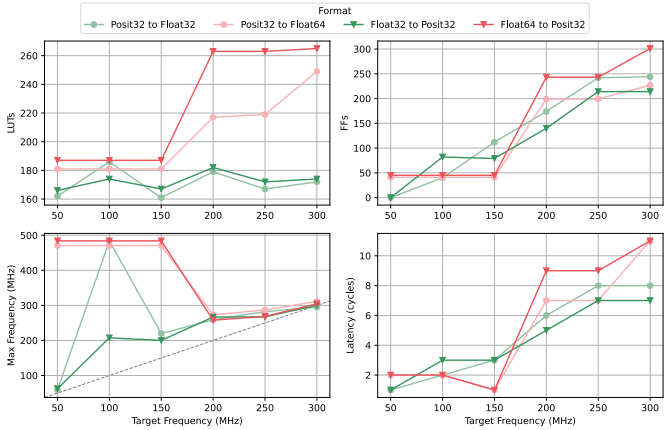
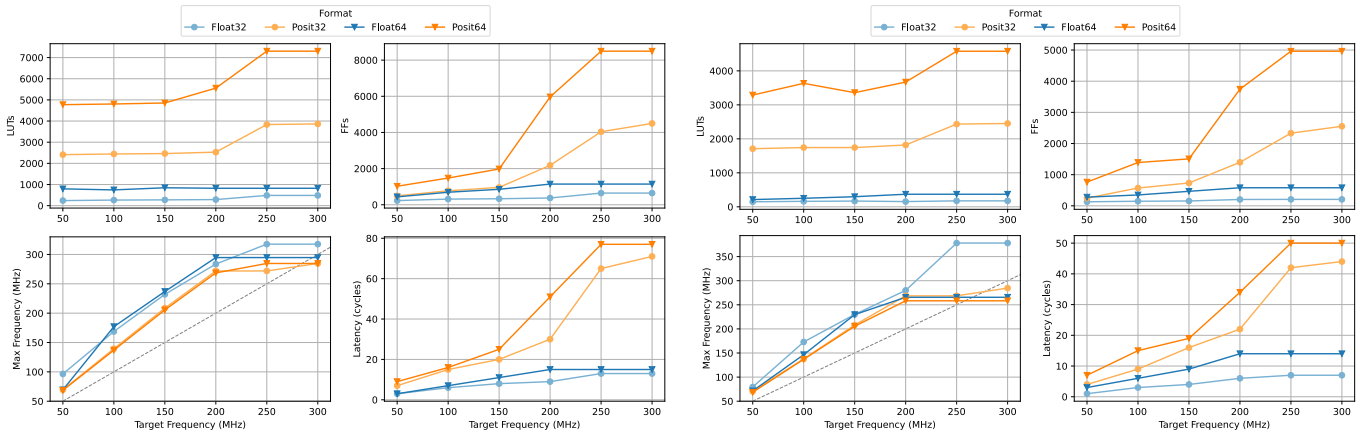


Fig. 7. HLS results for posit-float converters.

1) Comparison With Previous Works: To the best of our knowledge, MArTo [16] is the only library that provides posit arithmetic support for HLS up to date. It consists of a templatized C++ library compliant with Vitis HLS. It currently offers standalone posit adders, subtracters, and multipliers, but does not support operations such as division, square root, or comparison, unlike the proposed work. We use Vitis HLS 2021.2 to perform HLS of the designs, targeting the same device (Artix-7) and frequency (150 MHz). The RTL synthesis is performed by Xilinx Vivado with the default configuration. Although the focus of this work is posit arithmetic, floating-point designs are generated as well for better comparison and understanding of the results. It is worth mentioning that while MArTo provides arithmetic units designs for posit arithmetic, the floating-point units are from the proprietary Xilinx Floating-Point Operator IP.

In the same manner as with Bambu, we first conduct experiments for single arithmetic operators. Note that, in this case, we are not only using a different posit operator library, but also the HLS tool changes. Therefore, such an operator-level evaluation allows us to isolate as much as possible the differences between different HLS tools and to be able to analyze in more detail the differences between the two posit operator libraries. The synthesis results for posit addition and multiplication units provided by MArTo under different target frequencies are depicted in Fig. 8.

In contrast with the proposed library of posit operators, in this case, the Posit32 units require even more area than the corresponding Float64 operators. When compared with Fig. 6, on average the 32-bit floating-point adders generated with FloPoCo/Bambu require up to 1.5× more LUTs and 1.33× more FFs than the ones generated with MArTo/Vitis (respectively, 1.26× and 1.36× for 64-bit adders). However, the situation is very different with respect to posit adders. The units generated with the latter tools require on average 4.14× more LUTs and 2.66× more FFs than the proposed 32-bit units (respectively, 3.7× and 2.25× for 64-bit posit adders). The same behavior is observed in the multiplier units, which



(a) Addition (b) Multiplication

Fig. 8. Vitis HLS results for basic arithmetic MArTo [16] operators.

also make use of DSP units, again with an overhead for those multipliers defined at the MArTo library.

If comparing the latency of the operators, noticeable differences are found between the posit libraries. The 32-bit adders from MArTo require 1.74× more cycles, on average, than the proposed units (1.54× more for 64-bit adders), and for the 32-bit multipliers, the overhead is 1.36× more cycles, on average (1.25× for the 64-bit units).

Overall, we can conclude that at the operation level, the library of posit operators for HLS proposed in this work presents more efficient units in terms of both area and performance than those proposed in previous works.

D. Application-Level Evaluation

By considering the behavior of a diverse range of applications, we aim to assess the suitability and efficacy of utilizing floating-point and posit formats in accelerators for real-world applications. The chosen PolyBench programs were synthesized with Bambu. This evaluation leverages the frequency results obtained from the operation-level analysis. In consideration of the above results, a target frequency of 150 MHz has been set for all experiments.

Synthesis results for the multiple benchmarks under the MINI dataset size are depicted in Fig. 9. Larger sizes mainly affected the latency of the accelerators, in the same proportion for all the cases. Although each benchmark yields different results, similar patterns can be found in each of the metrics. In addition, the geometric mean across the proposed benchmarks is included for the sake of comprehension.

As can be seen, when comparing LUTs and FFs between cases #1 and #2, there are two patterns clearly distinguishable. For the applications that only require addition and multiplication, the posit overhead is slightly higher than for the floating-point case, between 1.06× and 1.45×. On the other hand, for cholesky, covariance and ludcmp benchmarks, the overhead of posit accelerators ranges from 1.58× to 3.96×. This corresponds to the applications using division and square root. As shown in Fig. 6, those posit operators required much more hardware resources in comparison with the floating-point ones, especially in the case of Posit64. Such a discrepancy suggests that the posit division and square root operators are not optimized as well as the corresponding floating-point units. Overall, the Posit32 accelerators have an average overhead of 1.46× and 1.72× in terms of LUTs and FFs, respectively, and similarly, for the 64-bit case, the LUTs and FFs overhead is 1.73× and 2.12×, respectively.

As for the use of DSPs, all benchmarks show exactly the same result, which in turn is the same as the one requiring a single multiplier unit, as mentioned above. This reveals that the amount of floating-point or posit multipliers is constant (and equal to one) across the experiments. This is in line with the behavior of Bambu, which allocates a single instance of each function/floating-point operation.

For BRAMs evaluation, again two different patterns can be distinguished between the applications that use division and those that do not. As has been mentioned, floating-point division units require 7 and 14 extra BRAMs for 32 and 64-bit designs, respectively. For the rest of the applications, the amount of BRAMs required by posit-based accelerators is the same as that used by floating-point kernels since they are encoded with the same number of bits.

Although the target frequency is set at 150 MHz, as shown in Fig. 9 this cannot be satisfied by all benchmarks. In particular, only 32-bit benchmarks satisfy this timing condition in most cases. The other time-related metric, latency, presents more differences between number formats. Posit32 requires between 1.09× up to 1.56× more cycles than Float32 accelerators (1.30× more on average), while the overhead for 64-bit formats ranges from 1.29× to 1.98× (1.59× more on average). Again, the higher increments are on the benchmarks that make use of division and square root operators.

Case #3 is worth mentioning separately. The amount of DSPs and BRAMs is exactly the same as their floating-point counterpart, since the arithmetic units employed in this hybrid scenario are the same as for case #1. However, as depicted in Fig. 7, such conversions introduce certain hardware overhead (about 10-15% more LUTs for 32-64 bits, and about 18-19% more FFs for 32-64 bits, respectively). In addition, the posit-float conversion also requires some extra clock cycles, but this is just 1 or 2 cycles per conversion, which results in less than 3% of overhead when considering any of the PolyBench programs.

1) Comparison with Previous Works: For a further comparison, we used MArTo to generate posit-based accelerators for

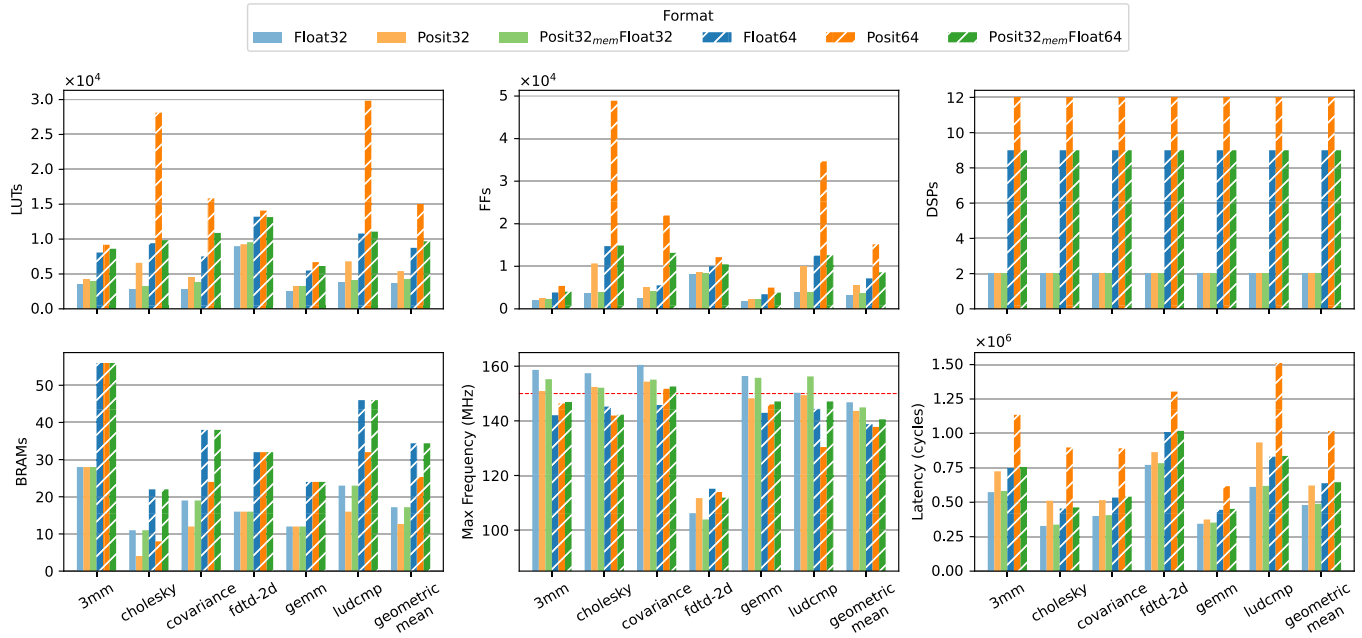


Fig. 9. Bambu HLS results for PolyBench benchmarks.

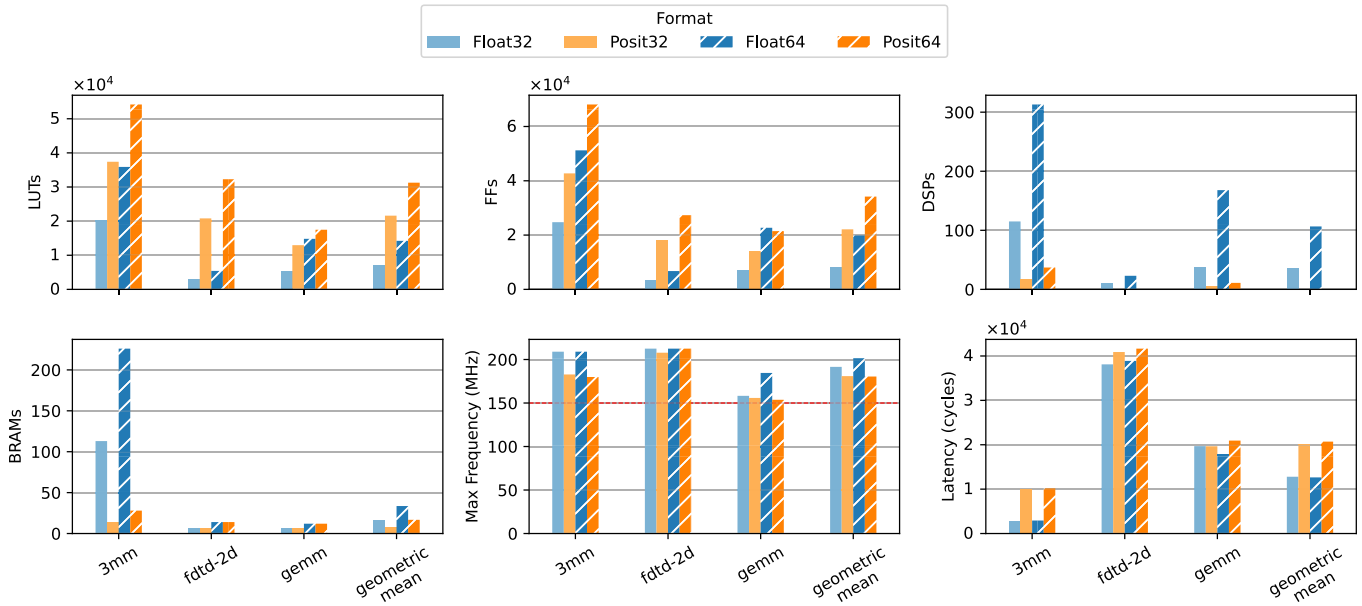


Fig. 10. Vitis HLS results for PolyBench benchmarks.

the aforementioned PolyBench applications supported by this tool, i.e., excluding cholesky, covariance and ludcmp because they use division and square root, which is not supported. Within MArTo, posit multiplications return an internal data format that must be converted back into posit, and subtractions must be re-written as additions with negative numbers, among other changes that must be done in the source code. Again, we use Xilinx Vitis to perform the HLS of the designs.

As can be seen in Fig. 10, the designs obtained with Vitis HLS lead to very different synthesis results than the ones generated with Bambu. At first glance, it can be seen that in this scenario the Posit32 accelerators generally require more LUTs and FFs than the Float64 accelerators, and also exhibit higher latency than the latter. In contrast to the previous case, the synthesis results for Vitis vary considerably between the benchmarks. Although there are important differences in the floating-point case, let us focus on the comparison in the posit designs between the two HLS tools. Comparing the Vitis accelerators with the one generated by Bambu, it can be seen that, except for the fdtd-2d application (where the posit designs do not use any DSPs), the designs generated by Vitis generally require more hardware resources (between 2.6× and 8.8× more LUTs, 2.1× and 16.3× more FFs, and 0.91× and 8.5× more DSPs, depending on the application and format). On the other hand, the number of BRAMs used for Vitis designs is half that of Bambu designs.

This hardware overhead has certain advantages in terms of frequency, as the designs generated with MArTo and Vitis meet the target frequency in all the benchmarks, even for the 64-bit scenario. The latency is also lower than for the Bambu accelerators. This may seem to contradict the conclusions of the previous operator-level comparison. However, there are

two aspects that need to be highlighted here. First, as can be seen, not only is the latency of the posit-based accelerators lower, but so is that of the floating-point accelerators. Also, according to the official Xilinx documentation [39], *the default operation of Vitis HLS is to first maximize performance*, while Bambu generates a single instance of each function or floating-point operation (even if there are multiple call points in the program), thus providing more area-efficient designs.

The reason for this discrepancy in results is therefore the scheduling and binding algorithms used by the different HLS tools, rather than differences in RTL design. However, it should be remembered that the purpose of this study is to analyze the impact of the different encodings, rather than to compare the performance of different HLS tools. In this sense, the results obtained with Vitis for the PolyBench application follow the same patterns observed in the previous analysis for the basic arithmetic operations, i.e., the use of posit arithmetic requires higher latency and more hardware resources than even Float64.

Despite the potential benefits of Vitis HLS, our evaluations indicate that our posit library leads to higher performance and lower area under the same HLS tool. Specifically, our experiments show that the Posit32 (respectively, Posit64) designs produced by MArTo require, on average, 1.57× (respectively, 1.64×) more clock cycles than the corresponding float designs. However, the same set of Posit32 (respectively, Posit64) accelerators proposed in this work exhibits a lower latency overhead of 1.15× (respectively, 1.40×) compared to the float designs. Also, in terms of area requirements, the posit designs from MArTo present a higher increment factor with respect to the corresponding designs in floating-point. For example, the Posit32 designs from previous work use on average 3.1× more LUTs and 2.67× more FFs than the Float32 designs, while the increment in the proposed work is just of 1.15× and 1.19× for LUTs and FFs, respectively. Similar figures are obtained for 64-bit precision.

VIII. Conclusion and Future Work

This paper presented an end-to-end HLS flow with support for posit arithmetic to create custom accelerators that exploit the higher accuracy of such arithmetic format. Additionally, a memory customization approach that uses posit numbers in memory while keeping the logic of the accelerators in floating-point is proposed, leveraging the benefits of posits in situations where there is no support for this alternative format. These solutions are implemented on top of open-source tools like FloPoCo (for implementation of the RTL library of posit operators) and Bambu (for the HLS). To illustrate the capabilities of the proposed workflow, as well as analyze the effects of the different arithmetic formats, floating-point and posit kernels for computer-intensive applications were deployed. The results showed that posit arithmetic consistently outperforms classical floating-point notations in terms of accuracy, without increasing memory usage or requiring additional bits for numerical representation. This finding could be particularly valuable in scenarios where memory is a critical resource or where the cost of expanding memory outweighs the benefits of improved accuracy. However, such extra accuracy has an associated cost. The 32-bit posit-based accelerators require about 1.46× more LUTs, 1.73× more FFs, and 1.30× more cycles than the corresponding floating-point accelerators. The proposed HLS flow was compared with previous works that rely on commercial HLS tools. The FPGA synthesis outcomes demonstrate that, when compared to the corresponding accelerators devised for floating-point arithmetic, the proposed posit designs exhibit reduced overhead than those from previous works. Overall, the findings of this paper demonstrate a clear trade-off between precision and hardware resources in posit arithmetic. Consequently, the decision to utilize and implement a specific format should be left to the discretion of the hardware designers, taking into consideration their specific requirements and constraints.

In the future, integration of posit fused arithmetic (using the quire accumulator) in the HLS flow will be explored, which will provide even more accurate results. The proposed workflow will be further optimized to reduce the current gap between posit and floating-point arithmetic.

References

[1] J. L. Hennessy and D. A. Patterson, "A new golden age in computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[2] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar. 2018.

[3] P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "High-level synthesis of accelerators in embedded scalable platforms," in *Proc. 21st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2016, pp. 204–211.

[4] L. Du et al., "A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 1, pp. 198–208, Jan. 2018.

[5] X. Xie, J. Lin, Z. Wang, and J. Wei, "An efficient and flexible accelerator design for sparse convolutional neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 7, pp. 2936–2949, Jul. 2021.

[6] N. C. Thompson and S. Spanuth, "The decline of computers as a general purpose technology," *Commun. ACM*, vol. 64, no. 3, pp. 64–72, Mar. 2021.

[7] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Dordrecht, The Netherlands: Springer, 2008.

[8] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 8–17, Jul./Aug. 2009.

[9] P. Mantovani et al., "Agile SoC development with open ESP," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 2020, pp. 1–9.

[10] C. H. Vun, A. B. Premkumar, and W. Zhang, "A new RNS based DA approach for inner product computation," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 60, no. 8, pp. 2139–2152, Aug. 2013.

[11] J. Hormigo and J. Villalba, "HUB floating point for improving FPGA implementations of DSP applications," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 64, no. 3, pp. 319–323, Mar. 2017.

[12] J. Osorio, A. Armejach, E. Petit, G. Henry, and M. Casas, "A BF16 FMA is all you need for DNN training," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 3, pp. 1302–1314, Jul. 2022.

[13] A. A. Del Barrio, R. Hermida, and S. Ogrenci-Memik, "A combined arithmetic-high-level synthesis solution to deploy partial carry-save radix-8 booth multipliers in datapaths," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 2, pp. 742–755, Feb. 2019.

[14] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2019, Rev. IEEE 754-2008, IEEE Computer Society, 2019.

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

[16] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the hardware cost of the posit number system," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Barcelona, Spain, Sep. 2019, pp. 106–113.

[17] R. Murillo, A. A. D. Barrio, and G. Botella, "Customized posit adders and multipliers using the FloPoCo core generator," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

[18] F. de Dinechin, L. Forget, J.-M. Müller, and Y. Uguen, "Posits: The good, the bad and the ugly," in *Proc. Conf. Next Gener. Arithmetic*, Singapore, Mar. 2019, pp. 1–10.

[19] R. Murillo, A. A. D. 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.

[20] (2022). *Standard for Posit^{TM} Arithmetic, Posit Standard, Posit Working Group Standard*. [Online]. Available: https://github.com/posit-standard/Posit-Standard-Community-Feedback

[21] A. Guntoro et al., "Next generation arithmetic for edge computing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 1357–1365.

[22] D. Mallasén, R. Murillo, A. A. D. Barrio, G. Botella, L. Piñuel, and M. Prieto-Matias, "PERCIVAL: Open-source posit RISC-V core with quire capability," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 3, pp. 1241–1252, Jul. 2022.

[23] R. Murillo, D. Mallasén, A. A. D. Barrio, and G. Botella, "Energy-efficient MAC units for fused posit arithmetic," in *Proc. IEEE 39th Int. Conf. Comput. Design (ICCD)*, Oct. 2021, pp. 138–145.

[24] L. Crespo, P. Tomás, N. Roma, and N. Neves, "Unified posit/IEEE-754 vector MAC unit for transprecision computing," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 5, pp. 2478–2482, May 2022.

[25] Y. Wang, D. Deng, L. Liu, S. Wei, and S. Yin, "PL-NPU: An energy-efficient edge-device DNN training processor with posit-based logarithm-domain computing," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 69, no. 10, pp. 4042–4055, Oct. 2022.

[26] H. Zhang and S.-B. Ko, "Design of power efficient posit multiplier," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 5, pp. 861–865, May 2020.

[27] R. Murillo, A. A. D. Barrio, G. Botella, M. S. Kim, H. Kim, and N. Bagherzadeh, "PLAM: A posit logarithm-approximate multiplier," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 4, pp. 2079–2085, Oct. 2022.

[28] S. H. Fatemi Langroudi, T. Pandit, and D. Kudithipudi, "Deep learning inference on embedded devices: Fixed-point vs posit," in *Proc. 1st Workshop Energy Efficient Mach. Learn. Cogn. Comput. Embedded Appl. (EMC)*, Williamsburg, VA, USA, Mar. 2018, pp. 19–23.

[29] M. Klöwer, P. D. Düben, and T. N. Palmer, "Number formats, error mitigation, and scope for 16-bit arithmetics in weather and climate modeling analyzed with a shallow water model," *J. Adv. Model. Earth Syst.*, vol. 12, no. 10, Oct. 2020, Art. no. e2020MS002246.

[30] 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.*, vol. 10, no. 4, pp. 1898–1908, Oct. 2022.

[31] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–4.

[32] F. Ferrandi et al., "Invited: Bambu: An open-source research framework for the high-level synthesis of complex applications," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Dec. 2021, pp. 1327–1330.

[33] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design Test Comput.*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[34] F. de Dinechin, "Reflections on 10 years of FloPoCo," in *Proc. IEEE 26th Symp. Comput. Arithmetic (ARITH)*, Jun. 2019, pp. 187–189.

[35] R. Murillo, D. Mallasén, A. A. D. Barrio, and G. Botella, "Comparing different decodings for posit arithmetic," in *Proc. Conf. Next Gener. Arithmetic (CoNGA)*, vol. 13253, 2022, pp. 84–99.

[36] R. Murillo, A. A. D. Barrio, and G. Botella, "A suite of division algorithms for posit arithmetic," in *Proc. 34th IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*. Porto, Portugal: IEEE Computer Society, Jul. 2023.

[37] E. T. L. Omtzigt and J. Quinlan, "Universal: Reliable, reproducible, and energy-efficient numerics," in *Proc. Conf. Next Gener. Arithmetic (CoNGA)*, vol. 13253, 2022, pp. 100–116.

[38] A. Canis et al., "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2, pp. 1–27, Sep. 2013.

[39] Xilinx. (2021). *Vitis High-Level Synthesis User Guide*. [Online]. Available: https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls

Raul Murillo received the B.Sc. degree in computer science and in mathematics and the M.Sc. degree in computer science from the Complutense University of Madrid (UCM), in 2019 and 2021, respectively, where he is currently pursuing the Ph.D. degree in computer engineering. Since 2022, he has been an Assistant Professor in computer science with the Department of Computer Architecture and Automation, UCM. He has performed research stays with Politecnico di Milano, Milan. His research interests include computer arithmetic, computer architecture, approximate computing, and deep neural networks.

Alberto A. Del Barrio (Senior Member, IEEE) received the Ph.D. 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 has been an Associate Professor (tenure-track and civil-servant) in computer science with the Department of Computer Architecture and System Engineering, UCM. His research interests include design automation, next generation arithmetic, and quantum computing. He has been the PI of the PARNASO Project, funded by the Leonardo Grants Program by Fundación BBVA. He is also the PI of the ASIMOV Project, funded by the Spanish MICINN, which includes a work package to research on the deployment of posits on RISC-V cores and accelerators. Since December 2020, he has been an ACM Senior Member.

Guillermo Botella (Senior Member, IEEE) received the M.A. (Sc.) degree in physics (fundamental) in 1999, the M.A. (Sc.) degree in electronic engineering in 2001, and the Ph.D. degree in computer engineering from the University of Granada, Spain, in 2007. He was a Research Fellow funded by EU working with Universidad de Granada and the Vision Research Laboratory, University College London, U.K. After that, he joined as an Assistant Professor with the Department of Computer Architecture and Automation, Complutense University of Madrid, Spain. He is currently an Associate Professor (accredited as a Full Professor in 2022). He has performed researching stays acting also as visiting professor with the Department of Electrical and Computer Engineering, Florida State University, Tallahassee, USA, from 2008 to 2012. His current research interests include hardware acceleration of signal processing systems, next generation arithmetic, and unconventional computing paradigms, such as analog and quantum computing. He has been the PI of the DESCARTES Project, funded by the Santander Bank which used those interest topics.

Christian Pilato (Senior Member, IEEE) received the Ph.D. degree in information technology from Politecnico di Milano, Milan, Italy, in 2011. He is currently a tenure-track Assistant Professor with Politecnico di Milano. He was a Post-Doctoral Research Scientist with Columbia University, New York City, NY, USA, from 2013 to 2016, and Universitã della Svizzera Italiana, Lugano, Switzerland, from 2016 to 2018. He was also a Visiting Researcher with New York University, New York City; TU Delft, Delft, The Netherlands; and the Chalmers University of Technology, Gothenburg, Sweden. His research interests include high-level synthesis, reconfigurable systems, and system-on-chip architectures, with an emphasis on memory and security aspects. He served as the Program Chair for EUC 2014 and ICCD 2022. He serves in the organizing and program committees of many conferences on EDA, CAD, embedded systems, and reconfigurable architectures, such as DAC, ICCAD, DATE, ASP-DAC, CASES, FPL, FPT, and ICCD. He is a Senior Member of ACM and a member of HiPEAC.