Cost-effective fixed-point hardware support for RISC-V embedded systems

Davide Zoni*, Andrea Galimberti

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano,
P.zza L. Da Vinci 32, 20133 Milan, Italy

Abstract

With the ever-increasing energy-efficiency requirements for the computing platforms at the edge, precision tuning techniques highlight the possibility of improving the efficiency of floating-point computations by selectively lowering the precision of intermediate operations without affecting the accuracy of the final result. Recent trends also demonstrated the possibility of successfully employing fixed-point computations in place of floating-point ones to further optimize the efficiency in the high-performance computing (HPC) domain. However, the use of integer functional units to support the execution of fixed-point operations in embedded platforms can severely degrade the energy-delay product (EDP). This work presents a cost-effective architecture to efficiently support fixed-point computations in embedded systems with two goals. On the one hand, it allows replacing the floating-point computations, with meaningful area and EDP improvements. On the other hand, it can complement the FPU, providing flexibility in selecting the best arithmetic for the target applications depending on their accuracy and performance requirements. Experimental results were collected from a representative set of floating-point-intensive applications executed on six variants of the baseline system-on-chip (SoC). We compared the efficiency of different floating- and fixed-point architectures in terms of accuracy, area, and EDP. Our fixed-point solution demonstrated a 0.651 EDP, normalized with respect to a SoC that featured a *binary32* FPU, while achieving a negligible accuracy loss, i.e., 0.0003% on average (0.004% peak), compared to *binary32* execution. In contrast, a SoC employing the integer functional units for fixed-point execution reports a 1.796 normalized EDP to achieve the same accuracy loss. Compared to the baseline SoC implementing only the integer units, the proposed architecture shows an area overhead limited to 4%, while the SoC featuring *binary32* floating-point hardware support requires 32% more resources.

Keywords: Fixed-point arithmetic, Efficient computing, Power-performance, Embedded systems, Compilers, Microarchitecture

1. Introduction

Modern embedded systems, especially those at the edge, are no longer only smart sensors but also general-purpose computing platforms in charge of efficiently performing a large variety of computationally intensive tasks. Apart from using system-wide energy-performance optimization policies [1, 2] employing run-time power monitors either in hardware [3] or software [4], a vast amount of research targets the optimization of the floating-point computations executed in such tasks. *Approximate computing* techniques operate at compile-time to leverage the error tolerance of several emerging applications by trading the accuracy of the computed data with their energy consumption [5]. However, the reduced accuracy of the results imposed by approximate computing techniques prevented their adoption across a wide range of application scenarios. Rather than tolerating the accuracy errors due to the limited precision of the hardware and software components, *transprecision computing* systematically delivers, by design, the correct amount of precision to all the intermediate computing steps of the applications [6]. In particular, it works by allowing a reduction in the precision of the intermediate computations until when the final result meets any accuracy constraints. Considering transprecision computing, *precision tuning* defines the set of compile-, i.e., static, and run-time, i.e., dynamic, precision tuning strategies, that allow fine-grained control of the precision of the intermediate steps of the computation. Precision tuning techniques were extensively used to optimize the floating-point computations [7, 8, 9, 10] by means of novel low-precision floating-point data formats [11, 12, 13] and the use of fixed-point arithmetic in place of the floating-point one [14]. Notably, carefully tuned fixed-point data formats were successfully employed to optimize the run-time energy-efficiency in the high-performance computing (HPC) [14] and GPU [15] scenarios. In contrast, the use of fixed-point precision tuning techniques in embedded computing platforms severely affect both performance and efficiency due to the use of integer functional units, i.e., ALU and multiplication-division, to perform fixed-point computations [14]. Notably, efficient embedded systems make use either of ad-hoc 8- and 16-bit hardware accelerators or general-purpose transprecision units still using 8 and 16 bits operands [16], thus preventing the execution of accuracy-sensitive applications.

Contributions - The paper presents a cost-effective fixed-point architecture to support precision tuning and approximate com-

*Corresponding author

Email addresses: davide.zoni@polimi.it (Davide Zoni), andrea.galimberti@polimi.it (Andrea Galimberti)

puting techniques with two contributions to the state of the art:

- **Efficient fixed-point architecture -** We introduce an extension (`Zm`) of the RISC-V ISA that implements eight fixed-point multiplication/division instructions, each corresponding to an instruction from the RISC-V ISA `M` standard extension, and provide hardware support for each of them. Our lightweight architecture supports fixed-point computations for any 32-bit fixed-point format with an area overhead limited to 4% compared to the baseline SoC without the FPU.
- **Architectural design space exploration -** We compared the proposed fixed-point support in terms of area, EDP, and accuracy, against a variety of implementations providing hardware support for integer operations only, for floating-point operations with different formats, and for fixed-point operations. Compared to native floating-point execution, our solution exhibits an average EDP of 0.651 and negligible accuracy loss, i.e., 0.0003% on average (0.004% peak). The EDP improvement and the low area overhead enable fixed-point precision tuning in embedded systems with the potential goal of replacing or complementing the floating-point hardware support.

The rest of the manuscript is organized into four parts. Section 2 reviews the background and the state of the art related to precision tuning and hardware design involving floating- and fixed-point formats. Section 3 presents the cost-effective fixed-point architecture. The compiler support as well as the experimental results in terms of EDP, accuracy, and resource utilization, considering different floating-point and fixed-point implementations, are discussed in Section 4. Conclusions are drawn in Section 5.

2. Background and related works

The goal of this section is twofold. Section 2.1 discusses the background on floating- and fixed-point data formats. Section 2.2 reviews the state-of-the-art considering hardware and software proposals related to precision tuning and transprecision computing.

2.1. Background

The floating- and fixed-point formats define the internal representation of real numbers in digital computing systems, regardless of whether the computation is performed in either software or hardware. Floating- and fixed-point data formats share the limitation of only approximating real values due to the finite amount of bits available to represent each number. Traditionally, floating-point data formats offer a wide dynamic range at the cost of complex arithmetic that imposes the use of specialized and high-latency computing platforms. In contrast, fixed-point data formats offer simpler arithmetic that, in general, can be computed employing the already available integer functional units. However, the latter solution presents two critical drawbacks. First, the fixed-point representation offers a smaller dynamic range than the floating-point representation. Second, the use of integer functional units to execute fixed-point computations can severely affect the energy-efficiency metric. Notably, the logarithmic number system (LNS) [17] offers an alternative number representation which simplifies multiplications and divisions as well as powers and roots at the cost of more complex additions, subtractions, and conversions between logarithmic and absolute values. The use of LNS is thus convenient only in applications that require few conversions and where most operations are multiplications and divisions, as in the case of digital signal processing [18] while fixed- and floating-point notations are advantageous in all the other application scenarios.

Floating-point representation - The floating-point data format specifies the encoding of a floating-point number f as defined by Equation (1), where S is the sign, M is the mantissa (or significand), i.e., the fractional part, E is the exponent, and b is the base.

$$f = (-1)^S \cdot M \cdot b^E \tag{1}$$

The precision of the floating-point representation increases with the number of bits used to encode the mantissa since the maximum distance between the encoded value and the real number decreases. Moreover, the dynamic range, i.e., the ratio between the largest and the smallest representable floating-point numbers, increases with the number of bits used for the exponent and, to a lesser extent, with the number of bits used to encode the mantissa. However, the growth of size in the floating-point format can negatively affect the energy consumption and performance of the computation [19].

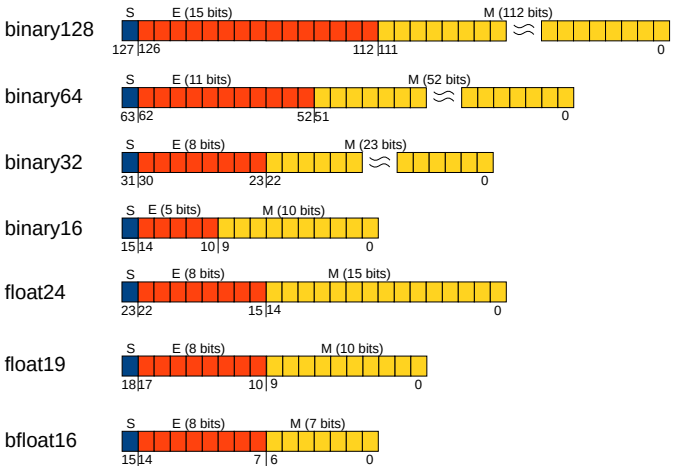

Figure 1: Encoding of floating-point formats. Each floating-point number is represented by a sign (S), an exponent (E), and a mantissa (M), according to Equation 1. The sign is encoded by 1 bit, while the bitwidth of the exponent and the mantissa depend on the specific floating-point format.

In order to offer different combinations of precision and dynamic range, the IEEE 754-2019 standard defines three floating-point formats, i.e., *binary32*, *binary64*, and *binary128*. Each format shares the binary base and a 1-bit sign encoding, while

the sizes of exponent and mantissa are different (see Figure 1). The *binary32*, *binary64*, and *binary128* floating-point formats are respectively encoded by 32, 64, and 128 bits. Traditionally, the use of *binary32* represents the de-facto choice to deliver a wide dynamic range and reasonable precision in embedded systems. *binary64* extends the format precision while *binary128* is usually reserved for scientific computations where no compromises on precision and dynamic range can be allowed. Moreover, the 16-bit IEEE *binary16* format was introduced to support floating-point computations in embedded applications for which dynamic range and precision reductions are allowed. However, the limited dynamic range offered by the *binary16* format represents an obstacle in those applications that require a wide dynamic range and that are not precision-sensitive, e.g., deep learning applications. To this extent, the brain-inspired floating-point format (*bfloat16*) emerged as an effective alternative to *binary16*. *bfloat16* is a 16-bit floating-point format that offers a wider dynamic range than *binary16* at the cost of a precision reduction. The *bfloat16* exponent is encoded by 8 bits instead of 5, while its mantissa is encoded by 7 bits instead of 10. Moreover, conversions to and from the *binary32* format can be easily performed by right-padding and truncating the mantissa, respectively. Likewise, other non-standard floating-point data formats can be defined to support precision tuning methodologies. For example, *float19* and *float24* are characterized by an 8-bit exponent, while their mantissa is encoded by 10 and 15 bits, respectively (see Figure 1).

Fixed-point representation - The fixed-point data format specifies a fixed-point number Q as composed by the integral (I) and the fractional (K) parts. A radix point separates the integral and the fractional parts, and its position is almost always implicit, i.e., the radix point is not explicitly encoded. Indeed, the programmer, by means of the compiler, assigns the radix point position and tracks it through every fixed-point computation. Equation (2) defines the formula to retrieve the encoded value from a fixed-point representation, where s is the sign and b is the base used in the encoding.

$$Q = (-1)^s \cdot \left(\sum_{j=0}^{I+K-1} x_j \cdot b^{j-K} \right) \tag{2}$$

From the implementation viewpoint, fixed-point numbers are stored as integers, and, in general, the hardware to implement integer instructions can also be reused to perform fixed-point computations. Figure 2 depicts the layout for unsigned ($U_{I+K,K}$) and signed ($S^*_{I+K,K}$) fixed-point numbers. The numbers are encoded using $I + K$ bits, where the integral part uses I bits while the fractional part uses K bits. We note that the signed fixed-point format can employ either the sign-magnitude ($S^{s/m}$) or the two's complement (S^{2c}) representation.The two's complement representation is the de-facto standard to encode signed fixed-point numbers, since it avoids the dual representation of the zero also allowing to use the standard integer functional units to perform fixed-point computations. In general, using the same

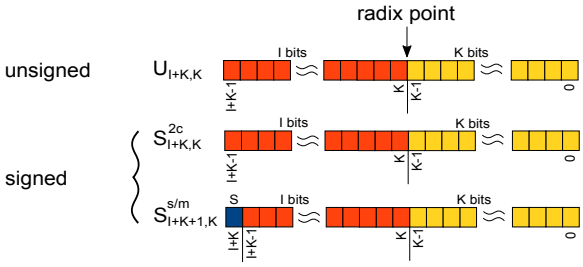

Figure 2: Structure of unsigned (U) and signed (S) fixed-point formats. Each fixed-point number is encoded using I bits for the integral part and K bits for the fractional part. For sign-magnitude-encoded fixed-point numbers ($S^{s/m}$), the sign bit is explicit and borrowed from the integral part.

number of bits, the fixed-point representation offers a narrower dynamic range than the floating-point one since the number of bits used to encode the fixed-point number limits the dynamic range. The dynamic range is characterized by a maximum value of $b^{|I|} - b^{-|K|}$ and $b^{|I+1|} - b^{-|K|}$ for signed and unsigned formats, respectively, and a minimum value (excluding zero) of $b^{-|K|}$ for both signed and unsigned fixed-point formats.

However, fixed-point arithmetic offers two critical advantages over the floating-point one. First, it allows to freely configure the radix point at the granularity of the single fixed-point instruction can deliver more precise representations of data of which the dynamic range is not a concern. Indeed, a 32-bit fixed-point format has a 32-bit precision, while the 32-bit *binary32* standard floating-point format has a 24-bit precision, which corresponds to the 23-bit mantissa. Second, fixed-point arithmetic enables faster and efficient computations without the need to convert between floating and integer data formats. However, the actual energy-efficiency due to the reuse of the hardware-level integer support to perform fixed-point computations depends on the actual microarchitecture. Considering embedded computing platforms, our research demonstrates the inefficiency of reusing the already available integer support for scalar fixed-point computations since, compared to *binary32* floating-point versions of the application, the normalized EDP worsens to 1.796 on average with a maximum of 6.854 when executing floating-point-intensive benchmarks (see experimental results in Section 4).

2.2. State of the art

This part reviews the state of the art of *i)* precision tuning frameworks and of *ii)* fixed- and floating-point computing platforms that support precision-tuned applications.

Precision tuning - Precision tuning techniques aim to maximize the run-time energy-efficiency by tuning the bit-width used to store and compute program data without affecting the accuracy of the final result. The majority of the precision tuning techniques in the literature target floating-point computations since they significantly contribute to the power consumption of the computing platforms.

Precision tuning techniques can be classified as either static or dynamic. Static precision tuning techniques deliver a mixed-precision version of the application that optimizes the energy-performance trade-off according to precision requirements de-

fined by the user. In contrast, dynamic precision tuning techniques can adapt the mixed-precision version of the application at run-time depending both on the run-time conditions of the computing platform and on the input data. Several state-of-the-art static precision tuning methodologies are available, but few of them leverage the possibility of reconfiguring the application at run-time, i.e., adopting a dynamic precision tuning approach.

Considering static precision-tuning methodologies, *fpPrecisionTuning* [20] proposes a near-optimal, distributed precision tuning approach for large floating-point applications. The framework allows to specify the accuracy of the final results, and then it iteratively executes the program to fine-tune the precision of each variable in the intermediate steps. However, the emitted code is restricted to *binary32* and *binary64* floating-point formats, forcing the compiler to several time-consuming conversions that severely limit the energy-performance gain of the precision tuning methodology. Similarly, *Promise* [9] and Precimonious [8] discuss precision tuning techniques targeting *binary32* and *binary64* floating-point data formats. *FPTuner* [10] offers automatic precision tuning of real-valued expressions, generating a mixed-precision version of the application that uses single, double, and quad precision floating-point data formats. *FlexFloat* [7] presents a precision tuning library that enables the design of transprecision applications. It supports multiple floating-point data formats to allow controlling at a fine grain the bitwidth of the exponent and mantissa of each variable of the program. [21] presented a tool that transformed the floating-point computations within a ANSI C code into a C code that leverages a fixed-point data type added by the authors as an extension to ANSI C. Such transformation from floating-point to fixed-point C code is part of the larger FRIDGE framework [22] , that translates a floating-point C code description into a fixed-point implementation that targets either dedicated hardware or programmable DSPs. *TAFFO* [14] discusses a precision tuning technique that allows converting the floating-point computation into the corresponding fixed-point version. Experimental results demonstrated significant performance improvements when the methodology is applied in the HPC domain. In contrast, little or no performance gain is shown in the embedded domain. [23] proposed a methodology to identify the optimal bitlength for fixed-point and floating-point operands in a hardware implementation, given the corresponding software description of the computation to be performed.

Considering dynamic precision-tuning methodologies, the *PetaBricks* [24] compiler allows the user to specify different mixed-precision versions of the application at compile time. The *PetaBricks* optimizer then decides dynamically which version to execute depending on the run-time conditions. We note that *PetaBricks* forces the user to define the mixed-precision versions of the applications manually. In contrast, [25] extends the *TAFFO* framework to support dynamic precision tuning. It offers *i)* a compile-time component that automatically generates multiple mixed-precision versions of the applications and *ii)* a run-time optimizer that selects the best candidate to execute.

Transprecision computing platforms - Traditionally, the computing platforms at the edge implement energy-efficient, general-purpose micro controllers to efficiently manage lightweight tasks. In this scenario, the use of ad-hoc hardware accelerators represents the de-facto solution to deliver the computational capacity required to support medicine [26, 27], security [28, 29], and deep learning [30, 31] applications without degrading the energy-efficiency of the embedded computing platform. Indeed, hardware accelerators offer the highest energy-efficiency, but their application-specific nature hinders their scalability and reusability.

However, the constant evolution of IoT applications imposes the design of scalable and reusable computing platforms to meet tight time-to-market deadlines. Thus, recent trends shifted from employing specialized hardware accelerators to designing efficient and flexible general-purpose computing platforms that implement either fixed-point [16] or floating-point [32] transprecision units. The *FloPoCo* framework [33, 34] allows designing complex floating-point operations in the form of custom accelerators that can be integrated into a general-purpose CPU. [35] presented a tool to automatically generate FPU designs for CPUs, emphasizing the optimization of the Wallace tree multiplier and the other basic logic blocks to maximize the efficiency of the generated FPU.

[36] proposes an FPU design employing a custom width for the operand. [13, 37] present different FPU microarchitectures considering the *bfloat16* format, while [12] proposes an FPU targeting a custom 16-bit floating-point format. [19] discusses a single instruction, multiple data (SIMD) transprecision FPU that can adapt the precision of the floating-point computation at run-time to maximize the efficiency. [32, 38] extend the work in [19] to trade the accuracy of the results with the possibility of performing low-precision SIMD floating-point instructions. In particular, [32] allows computing either multiple low-precision operations, i.e., *bfloat16* or *binary32*, or a single *binary64* floating-point instruction. Similarly, the work in [39] presents a fused multiply–accumulate (FMAC) design for transprecision computing that allows executing multiple concurrent FP operations at low precision. The work in [40] proposes *SMURF*, a scalar multi-precision floating-point unit that allows fine-tuning the width of the mantissa up to 512 bits.

Considering fixed-point arithmetic in the transprecision domain, [16] presents a near-threshold computing platform implementing low-precision fixed-point SIMD instructions. However, it targets ultra-low power platforms processing 8- and 16-bit data signals and does not consider the optimization of 32-bit scalar fixed-point hardware. In contrast, our manuscript presents a scalar fixed-point architecture that delivers higher efficiency than both current fixed- and floating-point solutions in terms of area, EDP, and accuracy.

The RISC-V P extension [41], currently in draft status, extends the RISC-V ISA by introducing a set of packed SIMD instructions, among which there are some fixed-point instructions. However, such instructions support a set of fixed-point formats limited to Q1.63, Q1.31, Q1.15, and Q1.7. [42] added support for the RISC-V packed SIMD P draft extension [41] to the RISC-V 64-bit CVA6 processor [43]. [44] presented an end-to-end compiler to optimize the code generation behavior of quantized neural networks that leverages the RISC-V P ex-

```
...
; load operands
lui     a0,%hi(a)
flw     ft0,%lo(a)(a0)
lui     a0,%hi(b)
flw     ft1,%lo(b)(a0)
; floating-point multiplication
fmul.s  ft0,ft0,ft1




; store result
lui     a0,%hi(res)
fsw     ft0,%lo(res)(a0)
...
```

(a) Floating-point `mul` using `F`

```
...
; load operands
lui   a0,%hi(a.fixp)
lw    a0,%lo(a.fixp)(a0)
lui   a1,%hi(b.fixp)
lw    a1,%lo(b.fixp)(a1)
; fixed-point multiplication
mul   a2,a0,a1
srli  a2,a2,10
mulh  a0,a0,a1
slli  a0,a0,22
or    a0,a0,a2
; store result
lui   a1,%hi(res.fixp)
sw    a0,%lo(res.fixp)(a1)
...
```

(b) Fixed-point `mul` using `M`

```
...
; load operands
lui    a0,%hi(a.fixp)
lw     a0,%lo(a.fixp)(a0)
lui    a1,%hi(b.fixp)
lw     a1,%lo(b.fixp)(a1)
; fixed-point multiplication
z.mul  a0,a0,a1,10




; store result
lui    a1,%hi(res.fixp)
sw     a0,%lo(res.fixp)(a1)
...
```

(c) Fixed-point `mul` using `Zm`

Listing 1: RISC-V assembly code for (a) a 32-bit floating-point multiplication using the `F` extension, (b) a 32-bit fixed-point multiplication using the `M` extension, and (c) a 32-bit fixed-point multiplication using the optimized `Zm` ISA extension proposed in this work.

tension [41] to optimize quantized neural network applications.

3. Fixed-point architecture

This section presents the cost-effective architecture to efficiently perform scalar fixed-point computations splitting the discussion into three parts. First, Section 3.1 shows the limitations of executing a fixed-point version of the application with the traditional integer functional units. Second, Section 3.2 introduces the proposed `Zm` extension to the RISC-V ISA, that is meant to enable scalar fixed-point instructions. Last, Section 3.3 discusses the proposed hardware design. In the following, we target the 32-bit RISC-V ISA [45] since, without loss of generality, it provides a representative use case for our research and it is becoming a de-facto industry and academic standard due to its extensibility, efficiency, and its royalty-free license.

3.1. Comparing fixed- and floating-point binaries

By analyzing different assembly versions of the same application, this section discusses the limitations of using standard integer hardware to execute fixed-point computations on 32-bit embedded systems. In particular, we analyze the baseline operations, i.e., multiplication, division, addition-subtraction, and comparison, considering the RISC-V ISA and three assembly versions, respectively exploiting the floating-point (*F*), integer (*M*), and fixed-point (*Zm*) extensions.

Fixed-point multiplication - Considering the 32-bit RISC-V ISA, Listing 1 details three assembly versions that perform a multiplication between the values stored in the `a` and `b` variables and store the result in the `res` variable. In particular, Listing 1(a) details the floating-point implementation, Listing 1(b) details the fixed-point version using the integer multiplication support, i.e., the `M` extension to the RISC-V ISA, and Listing 1(c) details the fixed-point implementation using a dedicated `mul` instruction, i.e., the proposed `Zm` extension to the RISC-V ISA. We use global variables for both the operands and the result to uniform the structure of each assembly fragment and, thus, to focus on the implementation of the `mul` instruction. In particular, the code in Listing 1(a) uses the *binary32* format, while, without loss of generality, fixed-point codes in Listing 1(b) and Listing 1(c) use a 32-bit fixed-point format with 10 bits for the fractional part.

Limiting the analysis to the actual computation, the FPU allows to use of the `fmul.s` instruction to efficiently perform the single-precision floating-point multiplication. In contrast, current precision tuning techniques deliver fixed-point versions of the application leveraging the integer multiplication and division support under the assumption that integer computations are faster than floating-point ones. We note that the argument is valid only if the target platform implements a 64-bit architecture.

Indeed, from the theoretical point of view, the multiplication between two 32-bit operands results in a 64-bit product. If the two operands are $Q_{I+K,K}$ fixed-point numbers, then the full-precision 64-bit result will have a $Q_{2(I+K),2K}$ fixed-point encoding. The leftmost I and rightmost K bits must be discarded to obtain a result with the same encoding as the two operands, thus resulting in truncation and precision loss. However, a 32-bit architecture can not implement a 32x32-bit multiplication that directly returns a 64-bit product. Instead, the `M` extension of the RISC-V ISA implements two distinct instructions, `mul` and `mulh`, that return the least and the most significant halves of the product of two signed operands, respectively. The two halves of the product must then be composed to obtain the fixed-point product by means of additional instructions, i.e., a pair of shift operations and the logical `OR` between the two shifted halves of the result (see Figure 3). Without loss of generality, Listing 1(b) shows that the fixed-point multiplication between two $Q_{32,10}$ operands on a 32-bit processor requires five instructions, with a negative impact on both the energy and performance metrics. In contrast, our solution delivers an efficient fixed-point hardware support through a dedicated fixed-point multiplication instruc-

```
...
; load operands
lui     a0,%hi(a)
flw     ft0,%lo(a)(a0)
lui     a0,%hi(b)
flw     ft1,%lo(b)(a0)
; floating-point division
fdiv.s  ft0,ft0,ft1




; store result
lui     a0,%hi(res)
fsw     ft0,%lo(res)(a0)
...
```

(a) Floating-point `div` using `F`

```
...
; load operands
lui   a0,%hi(a.fixp)
lw    a0,%lo(a.fixp)(a0)
lui   a1,%hi(b.fixp)
lw    a2,%lo(b.fixp)(a1)
; fixed-point division
srai  a1,a0,31
srai  a3,a2,31
srli  a4,a0,22
slli  a1,a1,10
or    a1,a1,a4
slli  a0,a0,10
call  __divdi3@plt
; store result
lui   a1,%hi(res.fixp)
sw    a0,%lo(res.fixp)(a1)
...
```

(b) Fixed-point `div` using `M`

```
...
; load operands
lui   a0,%hi(a.fixp)
lw    a0,%lo(a.fixp)(a0)
lui   a1,%hi(b.fixp)
lw    a1,%lo(b.fixp)(a1)
; fixed-point division
z.div a0,a0,a1,10




; store result
lui   a1,%hi(res.fixp)
sw    a0,%lo(res.fixp)(a1)
...
```

(c) Fixed-point `div` using `Zm`

Listing 2: RISC-V assembly code for (a) a 32-bit floating-point division using the `F` extension, (b) a 32-bit fixed-point division using the `M` extension, and (c) a 32-bit fixed-point division using the optimized `Zm` ISA extension proposed in this work.

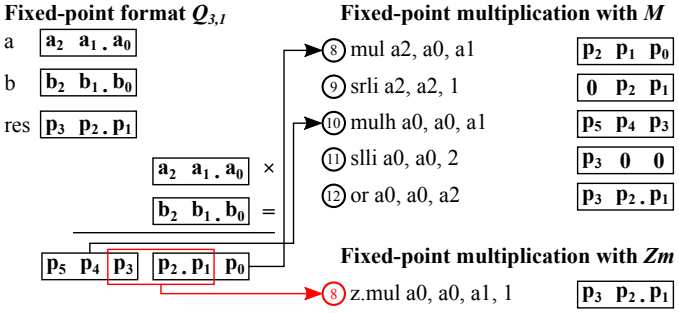

Figure 3: Example of fixed-point multiplication between two $Q_{3,1}$ operands `a` and `b`. It shows how the $Q_{3,1}$ product `res` is composed starting from the `mul` and `mulh` integer instructions and lists the results of each instruction from Listing 1(b) and Listing 1(c).

tion, i.e., `z.mul` (see Listing 1(c)).

Fixed-point division - Similarly to multiplication, using the hardware support for integer multiplication and division to implement the fixed-point division severely affects the execution latency (see Listing 2). In particular, Listing 2(a) details the floating-point implementation leveraging the FPU, while Listing 2(b) and Listing 2(c) detail the implementation of the fixed-point division using the integer functional units and the proposed hardware fixed-point support, respectively.

Notably, the fixed-point division between the two 32-bit fixed-point operands requires the execution of a 64-bit integer division, when hardware support is limited to the `M` extension for integer multiplication and division. 32-bit embedded platforms must therefore resort to the software implementation of the 64-bit integer division (see line 14 in Listing 2(b)). In this scenario, the use of the integer functional units to perform fixed-point divisions is far more detrimental than the multiplication scenario due to the use of a software-implemented routine, whose execution takes a number of clock cycles in the order of hundreds. This costly penalty further motivates using the proposed fixed-point support to optimize the energy-efficiency (see Listing 2(c)).

Other fixed-point operations - Concerning the remaining baseline fixed-point operations, we note that addition-subtraction and comparison operations directly correspond to their equivalent integer instructions, merely treating the fixed-point numbers as if they were plain integers. This equivalence is valid for binary operators when the two operands implement the same fixed-point format. Otherwise, there is a need to shift one of the two operands to have the same fixed-point encoding before performing the actual computation. However, we have not optimized such instructions since the cumulative latency of such fixed-point operations is still better than the one of the corresponding floating-point instructions (see Table 2 in Section 4).

3.2. Fixed-point ISA extension

This section details the extension to the RISC-V ISA to support the scalar fixed-point operations. Once more, we note that the use of the 32-bit RISC-V ISA [45] is motivated by its wide adoption in both industry and research, even if our research is not tied to any specific ISA.

Table 1 details the instruction formats for the standard ISA extensions as well as the fixed-point one introduced by our research, i.e., `Zm-type`. The `Zm-type` instruction format makes use of the `7b'0001011` opcode to lie in the `cust0` ISA map, that represents one of the available ISA mapping spaces for experimental RISC-V ISA non-standard extensions [45]. The `Zm` ISA extension implements the fixed-point version of the eight instructions belonging to the standard `M` ISA extension. As shown in Section 3.1, the proposed `Zm` ISA extension supports otherwise time-consuming fixed-point instructions, i.e., divisions and multiplications, without considering those that can be efficiently implemented by means of the baseline integer support, i.e., additions-subtractions and comparisons. We note that

Table 1 Instruction formats of the standard and Zm RISC-V ISA extensions. The Zm fixed-point extension uses the cust0 opcode map, i.e., *insn*[6 : 0] = 00_010_11.

31	24	19	14	11	6	0	
funct7	rs2	rs1	funct3	rd	opcode		R-type
imm[11:0]		rs1	funct3	rd	opcode		I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		S-type
imm[{12,10:5}]	rs2	rs1	funct3	imm[{4:1,11}]	opcode		B-type
imm[31:12]				rd	opcode		U-type
imm[{20,10:1,11,19:12}]				rd	opcode		J-type

31	24	19	14	11	6	0	
imm7	rs2	rs1	funct3	rd	00_010_11		Zm-type

the M ISA extension makes use of the R-type instruction format to implement the 32-bit integer multiplication and division instructions between two values stored in rs1 and rs2. The MUL and MULH instructions execute the integer multiplication between 32-bit signed operands and return the lower and upper 32-bit halves of the 64-bit product. The MULHU and MULHSU instructions compute the multiplication between unsigned-unsigned and signed-unsigned operands, respectively, and return the upper 32 bits of the 64-bit product. DIV and DIVU perform the 32-bit signed and unsigned integer division, rounding towards zero, and return the 32-bit quotient. REM and REMU return the 32-bit remainder of the corresponding integer divisions.

We note that the fixed-point instructions must encode the position of the radix point. To this end, the Zm-type defines a new *register-register-immediate* instruction format, where the computation is executed by processing the values from two registers, i.e., rs1 and rs2, and the 7-bit immediate, i.e., imm7, and the result is stored in the rd register. Moreover, the 3-bit funct3 field allows to encode the 8 instructions of the Zm ISA extension. In particular, the 8 instructions of the Zm ISA extension offer the same semantic of the corresponding instructions in the M ISA extension, while the former make use of the fixed-point arithmetic. To this end, the Zm instructions have the same name as the corresponding M instructions with the additional z. prefix, e.g., z.mul implements the 32-bit fixed-point multiplication. Considering the Zm-type instruction format, we note that the values stored in the registers of any Zm instruction are considered to be fixed-point, and the 7-bit immediate field encodes the position of the radix point. Moreover, each instruction assumes a common fixed-point format for both the operands and the result, that share a radix point position that is specified by the 7-bit immediate. Thus, additional shift instructions are eventually required to uniform the fixed-point format of all the values before performing any Zm instruction.

3.3. Scalar fixed-point architecture

This section presents our modular architecture to support the M and the Zm extensions of the RISC-V ISA to efficiently perform integer and fixed-point multiplication-division instructions. To maximize the computational efficiency, the fixed-point additional logic leverages the already available integer multiplication-division support and a user-defined design-time parameter controls its implementation. The design targets the embedded computing platforms, where the fixed-point support represents a viable computing option to complement or to replace the floating-point unit (FPU). Our solution delivers a novel cost-effective architecture for integer and fixed-point multiplication-division instructions and allows a configurable integration of the additional logic supporting the fixed-point instructions.

The proposed architecture extends an existing functional unit devoted to supporting integer multiplication and division instructions from the RISC-V M extension. Figure 4 depicts the top view of the three-stage architecture that implements both the M and Zm extensions of the RISC-V ISA. The black blocks and wires compose the functional unit that supports the M extension, i.e., integer multiplications and divisions, while the blocks and wires highlighted in red provide the additional logic to support the Zm extension, i.e., fixed-point multiplications and divisions. The architecture takes five inputs, i.e., the operands (OpA and OpB), the 7-bit immediate (imm7), the 3-bit funct3, and the flag to signal whether the operation is an integer or a fixed-point one (isFixed), and produces one output, i.e., the result (res). The rest of this part details the three stages of the architecture.

Operands pre-processing stage - Starting from the value of the funct3 and isFixed signals, this stage prepares the operands to perform one of the sixteen instructions, split into eight integer and eight fixed-point multiplication-division instructions. funct3 (insn[14:12] in Table 1) determines the actual instruction, and we note that the funct3 values of the Zm instructions have the same encoding as the corresponding instructions in the M ISA extension. isFixed signals instead whether the instruction is either a fixed-point (isFixed = 1) or integer one (isFixed = 0). The value of isFixed is determined by the opcode (see insn[6:0] in Table 1) of the instruction encoding, and it is set to 1 if opcode is equal to 00_010_11 and to 0 otherwise.

Depending on the executed instruction, MUX_{A-D} propagate the magnitude of each operand. The unsigned operands are forwarded to the next stage unmodified, while signed operands are converted to the sign-magnitude format, since the actual computation is performed sign-less. Finally, the position of the radix point is stored in the fBits register, that contains either 0, for integer instructions, or the value of the imm7 input, for fixed-point ones.

Integer/fixed-point computation stage - The IMul block performs the multiplication between the two unsigned values stored in A_M and B_M inter-stage registers, treating them as plain binary integers. Considering fixed-point multiplication instructions, the programmable shift SRL_P then adjusts the radix point of the result, right-shifting the 64-bit output from the IMul unit by a number of bits equal to the imm7 field. We note that a right shift by zero bits is performed in case of an integer multiplication, i.e., $fBits = 0$, leaving the 64-bit product unchanged. For signed multiplications, the sign can be computed as either the

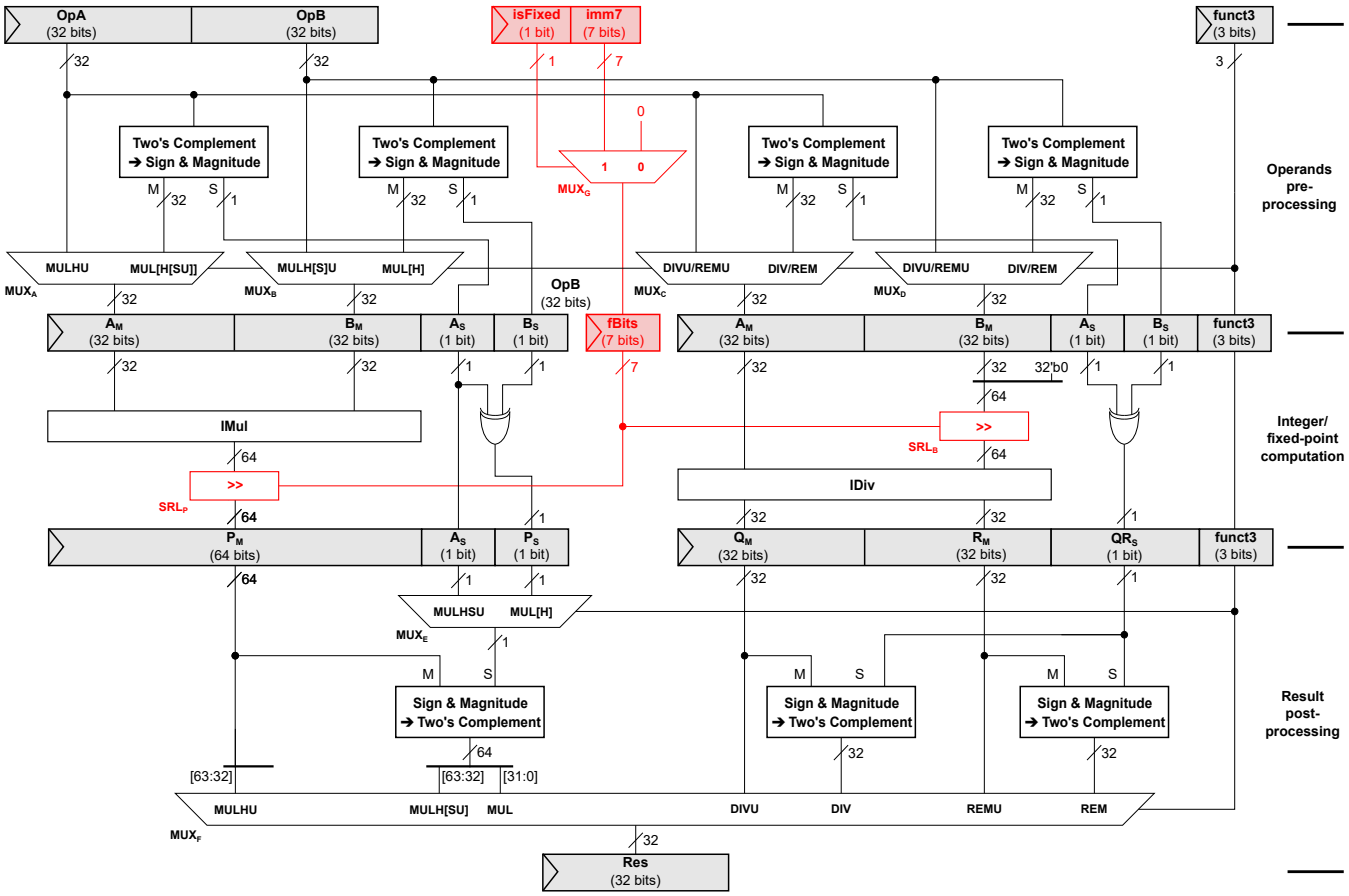

Figure 4: Top view of the proposed functional unit for the hardware support of the integer (`M`) and fixed-point (`Zm`) multiplication and division. The hardware required to implement the `M` extension of the RISC-V ISA is drawn in black, while red blocks and wires show the additional components that enable the `Zm` extension for fixed-point arithmetic.

sign of the `rs1` operand, i.e., the value stored in the A_S flip-flop, or the exclusive OR (*XOR*) between the signs of the two operands, i.e., the value stored in the P_S flip-flop.

The `IDiv` block performs the division between the two unsigned values stored in A_M and B_M and outputs the 32-bit quotient (Q_M) and the 32-bit remainder (R_M). For signed division and remainder instructions, the sign of the result is the *XOR* between the signs of the two operands and is stored in the QR_S flip-flop. Before the actual unsigned division carried out by the `IDiv` block, the divisor is adjusted through a programmable logical right shift (SRL_B) by a number of bits equal to the value contained in the `fBits` register. The `IDiv` block implements the radix-16 restoring division algorithm [18], which takes eight clock cycles to perform the unsigned division. Notably, even if the actual division is performed on 32-bit operands, the proposed `IDiv` architecture leverages a 64-bit internal structure, allowing to accommodate any 32-bit fixed-point format for the operands without increasing the computational latency and with minimum impact on the resource utilization.

Result post-processing stage - Starting from the intermediate values computed by means of either the `IMul` or the `IDiv` blocks, the MUX_F outputs one out of the eight results. The eight results correspond to the outputs of the eight integer or fixed-point instructions implemented in the `M` and `Zm` ISA extensions, respectively.

Depending on the executed operation, the MUX_F inputs are unsigned or signed results obtained either from the inter-stage registers or from the conversion blocks. In particular, the conversion blocks transform a sign-magnitude pair into the corresponding two's complement signed value.

4. Experimental Evaluation

This section discusses the experimental results of the proposed scalar fixed-point architecture in terms of area, EDP, and accuracy. The fixed-point architecture has been integrated into an embedded-class RISC-V system-on-chip (SoC) [46] to deliver an FPGA implementation of the entire computing platform. To provide a consistent comparison, we explored different SoC variants implementing multiple FPUs and the baseline hardware support for the integer multiplication and division (`M`) extension of the RISC-V ISA [45].

The rest of the discussion is organized into three parts. Section 4.1 details the experimental software setup. Section 4.2 presents the experimental hardware setup and discusses the latency of the integer, fixed-, and floating-point instructions. Section 4.3 presents the experimental results in terms of area, EDP, and accuracy.

Table 2 Operations latency of the different SoC implementations. For each hardware configuration, the number of clock cycles is reported for integer, fixed-point, and floating-point operations. Operations latency is detailed for additions-subtractions, multiplications, and divisions. The `sf` annotation highlights that soft-float function calls carry out the corresponding floating-point operations on SoC_{IM} and SoC_{IM+Zm}.

HW config.	Operations latency											
	Integer, Zm fixed-point				M fixed-point				Floating-point			
	AddSub	Mul	Div	Cmp	AddSub	Mul	Div	Cmp	AddSub	Mul	Div	Cmp
SoC_{IM}	1	5	14	1	1	14	>350	1	`sf`	`sf`	`sf`	`sf`
SoC_{IM+F32}	1	5	14	1	1	14	>350	1	5	5	12	4
SoC_{IM+F24}	1	5	14	1	1	14	>350	1	5	5	12	4
SoC_{IM+F19}	1	5	14	1	1	14	>350	1	4	4	10	4
SoC_{IM+F16}	1	5	14	1	1	14	>350	1	4	4	8	4
SoC_{IM+Zm}	1	5	14	1	1	5	14	1	`sf`	`sf`	`sf`	`sf`

4.1. Experimental software setup

The proposed fixed-point architecture was evaluated by employing a set of applications from the *PolyBench/C 4.2* benchmark suite [47], all of which contain a significant component of floating-point instructions. In particular, we configured the *PolyBench/C 4.2* benchmarks to use their "small dataset" problem size.

Fixed- and floating-point binaries - Each application is written in ANSI C, and it was compiled using the *LLVM 12.0* compiler toolchain to emit different fixed- and floating-point binaries. We note that the manuscript addresses the problem of efficiently supporting scalar fixed-point arithmetic in 32-bit embedded computing platforms. Thus, the design of a new precision tuning strategy falls outside the scope of our research.

For each application, we consider a floating-point binary and two distinct fixed-point versions. The floating-point binary targets the *binary32* data format and it is executed by four different FPUs, implementing hardware support for the *binary32*, *float24*, *float19*, and *bfloat16* floating-point formats, respectively. The first fixed-point version uses the hardware support for integer multiplication and division, while the second one also exploits the proposed architecture for fixed-point arithmetic. For both fixed-point versions, the fixed-point data format was selected through a static design space exploration between four 32-bit fixed-point formats, i.e., $Q_{32,23}$, $Q_{32,15}$, $Q_{32,10}$, and $Q_{32,7}$. For each application, we selected the fixed-point format delivering the best accuracy compared to the one delivered by the *binary32* version of the same application. Appendix A details the experimental results in terms of energy-delay product (EDP) and accuracy obtained by the bare-metal execution of all the applications. Table A.4 and Table A.5 report data for nine binary versions, i.e., a floating-point one, four fixed-point ones leveraging the `M` ISA extension, and four fixed-point ones using the proposed `Zm` ISA extension. The floating-point binary is executed on four FPUs implementing different formats.

LLVM compiler support - The LLVM compiler takes the C sources of the target applications annotated using the TAFFO precision tuning framework [14] at the granularity of the single floating-point instruction to select the fixed- or floating-point format to use. Such annotations are propagated to the LLVM IR level as part of the TAFFO precision tuning strategy. Adding support for the proposed `Zm` fixed-point ISA extension (see Section 3) required us to modify two parts of the LLVM compiler back-end: the `CodeGen` module and the `Target-dependent machine code` module. The `CodeGen` module transforms the high-level, target-independent LLVM intermediate representation (IR) into low-level, target-dependent machine language. The `Target-dependent machine code` module generates the machine-dependent code and emits the final binary RISC-V object. Within `CodeGen`, we created a new code generation path to generate fixed-point binaries that made use of the fixed-point instructions from the `Zm` extension. Within `Target-dependent machine code`, we created a new set of instruction formats and the corresponding matching rules to generate the binary form of the added Zm instructions. To this end, starting from the IR decorated with the annotations from TAFFO, our modified LLVM can generate binaries containing floating-point instructions, fixed-point ones with support for instructions from the `I` and `M` extensions only, and fixed-point ones with support for `Zm` fixed-point multiplication and division instructions.

Instructions mix - For each application, Figure 5 reports the mix of floating-point instructions as the percentage of the total number of instructions executed by the kernel, i.e., the region of interest of the application. The instructions are classified as additions-subtractions (AddSub), multiplications (Mul), divisions (Div), and comparisons (Cmp). We note that addition-subtraction and multiplication instructions dominate the instructions mix, while the number of executed division and comparison instructions is negligible but in few benchmarks.

To this end, the efficient support for both fixed-point additions-subtractions and multiplications is critical to optimize the energy-efficiency. From a different but related perspective, the hardware support for fixed-point divisions is also critical since they require the software execution of a 64-bit integer division if the computing platform exclusively implements the support to the `M` ISA extension, thus resulting in a significant EDP penalty.

4.2. Experimental hardware setup

We employed a configurable state-of-the-art SoC meant for FPGA targets [46] to deliver a complete evaluation of the proposed ISA extension. In particular, we considered an instance of the reference SoC that features a 32-bit in-order RISC-V

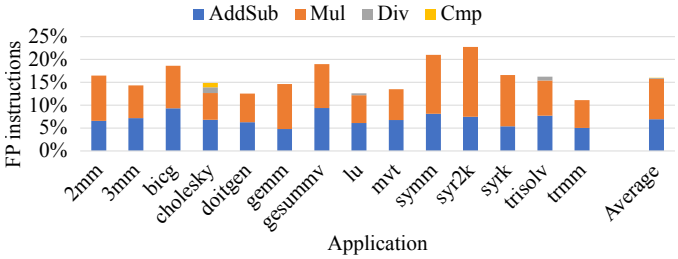

Figure 5: Percentage of floating-point instructions for each benchmark application. Floating-point instructions are split into additions-subtractions (AddSub), multiplications (Mul), divisions (Div), and comparisons (Cmp).

CPU, a 32-bit Wishbone bus, a 64KB BRAM-based main memory, a user-space UART for application input and output, and the debug infrastructure to allow the communication between the host and either the prototyping board or the simulation environment.

The baseline RISC-V CPU of the reference SoC supports both the base integer (`I`) and integer multiplication and division (`M`) extensions, representing the bare-minimum support to offer a computing platform that is fully functional for general-purpose applications. Starting from the baseline CPU, we considered the additional implementation of the single-precision, i.e., 32-bit, floating-point (`F`) RISC-V ISA extension. Furthermore, this work adds hardware support for scalar 32-bit fixed-point instructions through the experimental `Zm` RISC-V ISA extension. We note that the integration into the reference SoC of multiple FPU instances provides floating-point hardware support. The considered FPUs enable finding the optimal area-energy-accuracy trade-off by implementing different floating-point formats, and they represent the state-of-the-art solutions for low-power, high-performance embedded systems [48].

The FPU design is not a contribution provided by this work, and the scope of this research motivates the use of the FPUs from [48] in place of transprecision FPUs such as [19]. The employed FPUs target energy-efficient computing platforms, while transprecision FPUs implement multiple floating-point data formats within the same architecture, thus delivering flexible floating-point support at the cost of a reduction in both the energy- and area-efficiency.

The experimental results were obtained considering 6 SoC implementations (see Table 2). The four floating-point implementations, i.e., SoC_{IM+F32}, SoC_{IM+F24}, SoC_{IM+F19}, and SoC_{IM+F16}, support the base integer (`I`), integer multiplication and division (`M`), and single-precision floating-point (`F`) extensions. In particular, each one of the four SoCs supports a specific floating-point data format. More in detail, SoC_{IM+F32}, SoC_{IM+F24}, SoC_{IM+F19}, and SoC_{IM+F16} support the *binary32*, *float24*, *float19* and *bfloat16* floating-point data formats, respectively. We note that all the considered floating-point formats share the same 8-bit exponent bitwidth while offering different precision, i.e., the mantissa bitwidth ranges from 23 to 7 bits (see Section 2.1). SoC_{IM} offers only the baseline integer (`I`) and integer multiplication and division (`M`) hardware-level support, while SoC_{IM+Zm} adds the hardware support to the fixed-point instructions defined in the `Zm` RISC-V ISA extension (see Section 3).

Instructions latency analysis - For each SoC, Table 2 details the latency required to perform arithmetic instructions, expressed in terms of clock cycles. Latency is reported for the addition-subtraction, multiplication, division, and comparison instructions considering the integer, fixed-point, and floating-point data formats. We note that all the SoC_{IM+Fx} SoCs, where $x \in \{32, 24, 19, 16\}$, offer efficient hardware-level floating-point support with a maximum latency of 12 clock cycles for the floating-point division employing the *binary32* data format. In contrast, SoC_{IM} and SoC_{IM+Zm} do not offer hardware-level floating-point support, thus any floating-point operation must be executed in software by the corresponding soft-float function call. However, SoC_{IM+Zm} distinguishes itself for providing fast fixed-point hardware support. Additions and subtractions take 1 clock cycle, while the latency of fixed-point multiplications and divisions, i.e., 5 and 14 clock cycles, is almost on par with the latency of the corresponding hardware-supported floating-point instructions, that take up to 5 and 12 clock cycles respectively. Notably, fixed-point division takes 14 clock cycles due to 8 clock cycles for the radix-16 restoring division algorithm implemented by `IDiv`, 2 more clock cycles in `IDiv`, of which one to prepare the operands and one to prepare the output of the division, and 4 clock cycles due to the interstage registers (see Figure 4). Similarly, fixed-point multiplication takes 5 clock cycles, which can be divided into 1 clock cycle for the `IMul` component and 4 clock cycles due to the interstage registers.

We note that SoC_{IM} offers instead poor fixed-point performance for multiplications and divisions, since multiple integer instructions are required to implement a single fixed-point arithmetic operation (see Section 3.1).

SoC implementations - The complete SoC was implemented employing Xilinx Vivado 2019.2, using a 100MHz clock frequency, on the Digilent Nexys 4 DDR prototyping board. The Digilent board is equipped with a Xilinx Artix-7 100 FPGA, a mid-range cost-effective chip that features 63400 LUTs, 126800 flip-flops, 135 BRAMs, and 240 DSPs. To provide a fair evaluation, we implemented each one of the six design variants within the same SoC, employing the Vivado default synthesis and implementation strategies. The resource utilization of the functional units supporting the `M`, the `F`, and the `Zm` extensions and the whole SoC was extracted from the post-implementation netlist for each SoC variant. Power consumption results were collected for each SoC and each benchmark application as the dynamic power obtained by Vivado Report Power from post-implementation simulation. Accordingly, the energy consumption for an application executing on a SoC was computed as the product of the corresponding power consumption and execution time.

4.3. Area, EDP and accuracy results

Considering the six implemented SoCs, this section discusses the experimental results in terms of area (see Table 3), EDP (see Figure 6), and accuracy (see Figure 7).

Table 3 reports, for each SoC, the resource utilization in terms of look-up tables (*LUTs*), flip-flops (*FFs*), and digital signal processing (*DSP*) elements, providing *i)* a breakdown

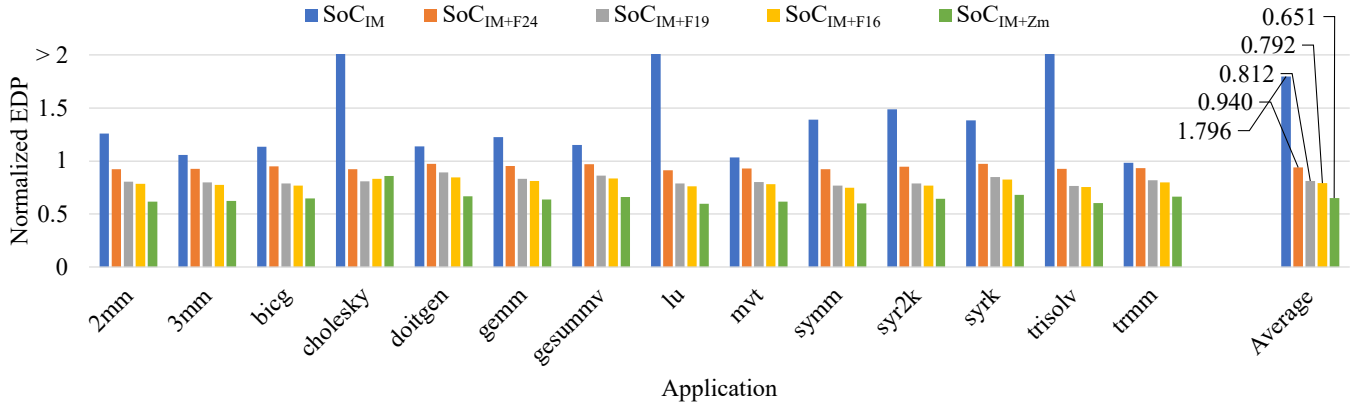

Figure 6: Energy-delay product (EDP), normalized with respect to SoC_{IM+F32}, for all considered SoCs. The normalized EDP is defined according to Equation (4).

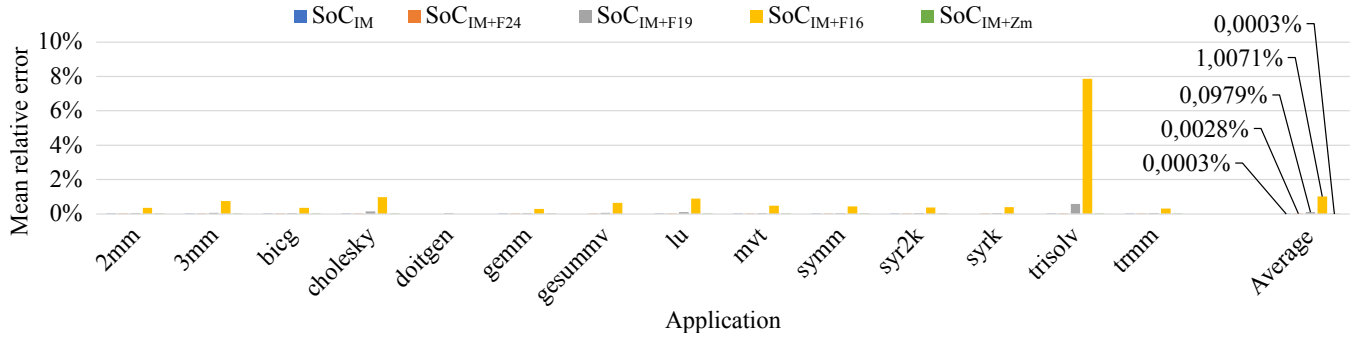

Figure 7: Accuracy of the results for all considered SoCs, expressed in terms of mean relative error (MRE) with respect to the results obtained from SoC_{IM+F32}. The MRE is defined according to Equation (5).

Table 3 Area results for the six considered SoCs. The resource utilization for the hardware support of the implemented RISC-V ISA extensions M, F, and Zm and for the whole SoC is reported in terms of look-up tables (LUT), flip-flops (FF) and digital signal processors (DSP), while $Area^{norm}$ measures the area normalized to the baseline SoC_{IM}.

HW config.	Res.	Resource utilization M	F	Zm	Total SoC	$Area^{norm}$
SoC_{IM}	LUT	3900	-	-	8474	1
	FF	2289	-	-	5775	
	DSP	4	-	-	4	
SoC_{IM+F32}	LUT	3900	2042	-	11223	1.32
	FF	2289	664	-	7599	
	DSP	6	6	-	10	
SoC_{IM+F24}	LUT	3900	1531	-	10713	1.26
	FF	2289	509	-	7444	
	DSP	4	2	-	6	
SoC_{IM+F19}	LUT	3900	1289	-	10498	1.24
	FF	2289	307	-	7312	
	DSP	4	3	-	7	
SoC_{IM+F16}	LUT	3900	1284	-	10493	1.24
	FF	2289	357	-	7292	
	DSP	4	0	-	4	
SoC_{IM+Zm}	LUT	3900	-	352	8859	1.04
	FF	2289	-	33	5810	
	DSP	4	-	0	4	

for the hardware support of each implemented extension of the RISC-V ISA and *ii)* the total amount required for the whole system-on-chip. Since LUT are the scarcest resource, i.e., the most used resource for implementing the whole SoC relatively to the total resources available on Xilinx FPGAs, we defined the area metric as the ratio between the LUT required by a SoC implementation SoC_i, where $i \in \{SoC_{IM}, SoC_{IM+F32}, SoC_{IM+F24}, SoC_{IM+F19}, SoC_{IM+F16}, SoC_{IM+Zm}\}$, and the LUT required by the baseline SoC_{IM}. Equation (3) defines such normalized area metric $Area^{norm}$, which allows to easily compare the resource utilization of the different hardware configurations. We note that smaller values of $Area^{norm}$ indicate a lower amount of utilized LUT resources.

$$Area_i^{norm} = \frac{LUT_i}{LUT_{SoC_{IM}}} \tag{3}$$

For each benchmark, Figure 6 and Figure 7 report the EDP and accuracy results of all the considered SoCs. In particular, the results are normalized to the ones obtained with SoC_{IM+F32}. Considering the execution of application a using the SoC implementation SoC_i, the EDP metric is defined as the product between the energy consumption of the SoC and the execution time of the application. Consistently, for each benchmark a and each SoC implementation SoC_i, $EDP_{a,i}^{norm}$, i.e., the EDP normalized with respect to SoC_{IM+F32}, is defined according to Equation (4). We note that values of EDP^{norm} less than 1 indicate an improved EDP-efficiency compared to SoC_{IM+F32},

while values greater than 1 signal a worse efficiency.

$$EDP^{norm}_{a,i} = \frac{EDP_{a,i}}{EDP_{a,SoC_{IM+F32}}} \quad (4)$$

Similarly, Equation (5) defines the accuracy metric, i.e., the mean relative error (MRE). For each application a and each SoC_i, the MRE is defined as the mean of the relative errors computed on the N_a output variables of the considered application.

$$MRE_{a,i} = \frac{\sum_{j=1}^{N_a} RelErr_{a,i,j}}{N_a} \quad (5)$$

In particular, Equation (6) defines the relative error for the output variable y_j of application a by comparing its value when executed by SoC_i with the one obtained by using SoC_{IM+F32}. To this end, SoC_{IM+F32} defines the golden accuracy since *binary32* represents the de-facto floating-point standard in the embedded systems domain.

$$RelErr_{a,i,j} = \frac{|y_{a,i,j} - y_{a,_{IM+F32},j}|}{y_{a,_{IM+F32},j}} \quad (6)$$

Area discussion - We start by considering SoC_{IM} as the baseline computing platform. SoC_{IM} supports both the base integer instructions (`I`) and the integer multiplication and division instructions (`M`). In this scenario, the implementation of the floating-point ISA extension (`F`) supports the execution of modern data-processing applications. However, the floating-point computation was shown as the first contribution to power consumption, thus motivating several methodologies to optimize the energy-performance trade-off. In particular, precision tuning techniques optimize the precision of intermediate floating-point computations to optimize the energy-budget without affecting the accuracy of the final result. To explore different precision-area trade-offs, we considered four SoCs with hardware support for floating-point arithmetic, SoC_{IM+F32}, SoC_{IM+F24}, SoC_{IM+F19}, and SoC_{IM+F16}. The implementation of the *binary32* FPU within SoC_{IM+F32} highlights the use of 2042 LUTs, 664 FFs, and 6 DSPs. By lowering the precision of the floating-point format, we observe a corresponding reduction in resource utilization. In particular, the FPU in SoC_{IM+F16}, which implements the *bfloat16* floating-point data format, requires 1284 LUT, i.e., 63% of the 2042 LUT required by the FPU of SoC_{IM+F32}, which implements the *binary32* floating-point data format. While the whole SoC_{IM+F32} requires 1.32× the area of SoC_{IM}, the normalized area metric $Area^{norm}$ for SoC_{IM+F16} decreases to 1.24×.

From a different but related perspective, precision tuning solutions optimize the energy-performance trade-off by replacing floating-point computations with fixed-point ones. In particular, SoC_{IM} represents the baseline solution to support the execution of the fixed-point versions of the application. We note that removing the FPU from the implemented SoC provides a significant saving in terms of area, as can be seen from the lower resource utilization for SoC_{IM}. However, the performance overhead resulting from the use of standard integer functional units to perform fixed-point computations significantly exceeds the area gain (see instruction latencies in Table 2 and EDP in Figure 6). SoC_{IM+Zm} implements the `Zm` ISA extension, which we introduced to overcome such efficiency penalty. Our architecture optimizes the execution of scalar fixed-point computations with limited area overhead, since the additional support for the `Zm` instructions requires 352 LUTs, 33 FFs, and no DSPs, i.e., i.e., 17% of the 2042 LUT required by the FPU of SoC_{IM+F32}, with the whole SoC_{IM+Zm} requiring just 1.04× the area of SoC_{IM}, i.e., 4% more LUT resources, as shown in Table 2.

EDP discussion - Figure 6 reports the EDP results obtained for the six considered SoCs and for each application. As defined in Equation (4), such EDP values are normalized to the ones obtained using SoC_{IM+F32}, which offers hardware support for *binary32*, i.e., the IEEE-754 single-precision floating-point data format. As expected, by limiting the analysis to the considered floating-point formats, the EDP improves by reducing the precision of the floating-point computations. In particular, we report an average normalized EDP of 0.940, 0.812, and 0.792 for SoC_{IM+F24}, SoC_{IM+F19}, and SoC_{IM+F16}, respectively. The EDP improvement is motivated by the faster execution (see Table 2) and lower resource utilization (see Table 3) of low-precision FPUs. Compared to SoC_{IM+F32}, for example, SoC_{IM+F16} highlights a 37% smaller FPU, in terms of LUT, and a latency reduction of the floating-point instructions by 1 clock cycle for additions, subtractions, and multiplications, and by 4 clock cycles for divisions.

In contrast, SoC_{IM} reports an average normalized EDP of 1.370. For each application, SoC_{IM} executes the optimal fixed-point version of the application, i.e., the one that ensures, in the first place, the maximum accuracy and, in the second place, the best EDP. The degradation in the EDP is due to the higher latency needed to perform fixed-point multiplications and divisions by means of integer instructions (see Table 2). SoC_{IM} takes 14 clock cycles to perform a fixed-point multiplication, i.e., almost three times slower than the SoCs supporting hardware-level floating-point computations, and few hundreds of clock cycles to perform a fixed-point division, i.e., a slowdown by more than an order of magnitude. We note that the EDP degradation is compensated neither by the speedup in performing the addition-subtraction instructions, which take 1 clock cycle for SoC_{IM} and at least 4 clock cycles for the SoCs implementing the FPUs, nor by the lower resource utilization, although SoC_{IM} saves up to 2749 LUTs compared to the SoCs implementing floating-point hardware support. We note that the implementation of the `Zm` extension to the RISC-V ISA delivers an average normalized EDP of 0.651 (see SoC_{IM+Zm} results in Figure 6). The sharp EDP improvement is due to *i)* the shallow area overhead, which is limited to 352 LUTs on top of SoC_{IM}, and *ii)* the low latency to execute the fixed-point instructions. In particular, SoC_{IM+Zm} requires only 1 clock cycle to execute addition-subtraction and comparison instructions, while the SoCs implementing the floating-point support take between 4 and 5 clock cycles, depending on their precision. Moreover, the latency to perform fixed-point multiplications and divisions is on par with the one required by the SoCs implementing the

hardware support for floating-point arithmetic.

Accuracy discussion - Several state-of-the-art proposals highlighted the emergence of precision-tolerant applications in different use case scenarios. This trend of accepting some accuracy loss in the output motivated the introduction of approximate computing and of precision tuning strategies that aim to optimize the efficiency by selectively lowering the precision of the computations.

Our experimental results confirm such a trend, highlighting an average accuracy loss within 1% for most executed applications and regardless of the computing platform. Considering the SoCs executing the floating-point versions of the applications, i.e., SoC_{IM+F32}, SoC_{IM+F24}, SoC_{IM+F19}, and SoC_{IM+F16}, the accuracy loss increases with the decrease in the precision of the implemented FPU.

We note that the choice of employing a low-precision floating-point unit to optimize the efficiency metric, i.e., the EDP, represents a limiting design strategy. In particular, the execution of precision-sensitive floating-point applications will suffer the use of low-precision FPUs, therefore requiring to resort to soft-float function calls to achieve full precision, at the cost of a significant drop-off in execution time. In contrast, the proposed `Zm` ISA extension can effectively support advanced precision tuning techniques to optimize both precision-sensitive and precision-tolerant applications at the cost of a tolerable area overhead in the order of 350 LUTs. The proposed fixed-point support ensures a significant EDP improvement, while the use of software-level precision tuning techniques can further optimize the already acceptable accuracy loss.

5. Conclusions

This work introduced a cost-effective architecture targeting embedded systems to efficiently perform integer and fixed-point computations. To this end, we introduced a RISC-V ISA extension that implements eight fixed-point multiplication/division instructions, each corresponding to an instruction from the RISC-V ISA M standard extension. The proposed design allows either to replace the floating-point computations to achieve a meaningful area reduction and EDP improvement or to complement the FPU to provide flexibility in the selection of the best arithmetic to use given a target application and its requirements. Experimental results in terms of area, EDP, and accuracy were collected from a representative set of floating-point intensive applications executed by means of six variants of the baseline SoC. In particular, we compared the efficiency of different floating- and fixed-point architectures. Each variant of the SoC was implemented on the Xilinx Artix-7 100 FPGA and validated by means of the Digilent Nexys 4 DDR board. Compared to the SoC implementing the sole integer support, the SoC implementing the *binary32* floating-point arithmetic uses 32% more resources while our fixed-point design limits the resource overhead to 4%. Compared to the SoC implementing the floating point support, our fixed-point architecture offers an average normalized EDP of 0.651 within a negligible accuracy loss, i.e., lower than 0.0003% on average. In contrast, the use of integer support to perform fixed-point computations degrades the normalized EDP up to 1.796, on average.

References

[1] D. Zoni, L. Cremona, W. Fornaciari, All-digital control-theoretic scheme to optimize energy budget and allocation in multi-cores, IEEE Transactions on Computers 69 (5) (2020) 706–721. doi:10.1109/TC.2019.2963859.

[2] D. Zoni, L. Cremona, W. Fornaciari, All-digital energy-constrained controller for general-purpose accelerators and cpus, IEEE Embedded Systems Letters 12 (1) (2020) 17–20. doi:10.1109/LES.2019.2914136.

[3] L. Cremona, W. Fornaciari, D. Zoni, Automatic identification and hardware implementation of a resource-constrained power model for embedded systems, Sustainable Computing: Informatics and Systems 29 (2021) 100467. doi:https://doi.org/10.1016/j.suscom.2020.100467.
URL https://www.sciencedirect.com/science/article/pii/S2210537920301918

[4] M. J. Walker, S. Diestelhorst, A. Hansson, A. K. Das, S. Yang, B. M. Al-Hashimi, G. V. Merrett, Accurate and stable run-time power modeling for mobile and embedded CPUs, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36 (1) (2017) 106–119.

[5] Q. Xu, T. Mytkowicz, N. S. Kim, Approximate computing: A survey, IEEE Design Test 33 (1) (2016) 8–22. doi:10.1109/MDAT.2015.2505723.

[6] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, N. Wehn, The transprecision computing paradigm: Concept, design, and applications, in: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018, pp. 1105–1110. doi:10.23919/DATE.2018.8342176.

[7] G. Tagliavini, A. Marongiu, L. Benini, Flexfloat: A software library for transprecision computing, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39 (1) (2020) 145–156. doi:10.1109/TCAD.2018.2883902.

[8] C. Rubio-González, Cuong Nguyen, Hong Diep Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, D. Hough, Precimonious: Tuning assistant for floating-point precision, in: SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1–12. doi:10.1145/2503210.2503296.

[9] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, N. Shanbhag, Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms, in: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), 2018, pp. 43–56. doi:10.1109/ISCA.2018.00015.

[10] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, Z. Rakamarić, Rigorous floating-point mixed-precision tuning, SIGPLAN Not. 52 (1) (2017) 300–315. doi:10.1145/3093333.3009846.
URL https://doi.org/10.1145/3093333.3009846

[11] J. Johnson, Rethinking floating point for deep learning, CoRR abs/1811.01721. arXiv:1811.01721.
URL http://arxiv.org/abs/1811.01721

[12] A. Agrawal, B. Fleischer, S. Mueller, X. Sun, N. Wang, J. Choi, K. Gopalakrishnan, Dlfloat: A 16-b floating point format designed for deep learning training and inference, in: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), 2019, pp. 92–95. doi:10.1109/ARITH.2019.00023.

[13] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, D. Mansell, Bfloat16 processing for neural networks, in: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), 2019, pp. 88–91. doi:10.1109/ARITH.2019.00022.

[14] S. Cherubin, D. Cattaneo, M. Chiari, A. D. Bello, G. Agosta, Taffo: Tuning assistant for floating to fixed point optimization, IEEE Embedded Systems Letters 12 (1) (2020) 5–8. doi:10.1109/LES.2019.2913774.

[15] C.-L. Lee, M.-Y. Hsu, B.-S. Lu, M.-Y. Hung, J.-K. Lee, Experiment and enabled flow for gpgpu-sim simulators with fixed-point instructions, Journal of Systems Architecture 111 (2020) 101783. doi:https://doi.org/10.1016/j.sysarc.2020.101783.
URL https://www.sciencedirect.com/science/article/pii/S1383762120300771

[16] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, L. Benini, Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25 (10) (2017) 2700–2713. doi:10.1109/TVLSI.2017.2654506.

[17] E. Swartzlander, A. Alexopoulos, The sign/logarithm number system, IEEE Transactions on Computers C-24 (12) (1975) 1238–1242. doi:10.1109/T-C.1975.224172.

[18] I. Koren, Computer Arithmetic Algorithms, 2nd Edition, A. K. Peters, Ltd., Natick, MA, USA, 2001.

[19] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, L. Benini, A transprecision floating-point platform for ultra-low power computing, in: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018, pp. 1051–1056. doi:10.23919/DATE.2018.8342167.

[20] N. Ho, E. Manogaran, W. Wong, A. Anoosheh, Efficient floating point precision tuning for approximate computing, in: 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), 2017, pp. 63–68. doi:10.1109/ASPDAC.2017.7858297.

[21] M. Willems, V. Bürsgens, H. Keding, T. Grötker, H. Meyr, System level fixed-point design based on an interpolative approach, in: Proceedings of the 34th Annual Design Automation Conference, DAC '97, Association for Computing Machinery, New York, NY, USA, 1997, p. 293–298. doi:10.1145/266021.266105.
URL https://doi.org/10.1145/266021.266105

[22] H. Keding, M. Willems, M. Coors, H. Meyr, Fridge: a fixed-point design and simulation environment, in: Proceedings Design, Automation and Test in Europe, 1998, pp. 429–435. doi:10.1109/DATE.1998.655893.

[23] A. Gaffar, O. Mencer, W. Luk, Unifying bit-width optimisation for fixed-point and floating-point designs, in: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 79–88. doi:10.1109/FCCM.2004.59.

[24] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, S. Amarasinghe, Petabricks: A language and compiler for algorithmic choice, SIGPLAN Not. 44 (6) (2009) 38–49. doi:10.1145/1543135.1542481.
URL https://doi.org/10.1145/1543135.1542481

[25] S. Cherubin, D. Cattaneo, M. Chiari, G. Agosta, Dynamic precision autotuning with taffo, ACM Trans. Archit. Code Optim. 17 (2). doi:10.1145/3388785.
URL https://doi.org/10.1145/3388785

[26] Y. Zhang, F. Zhang, Y. Shakhsheer, J. Silver, A. Klinefelter, M. Nagaraju, J. Boley, J. N. Pandey, A. Shrivastava, E. J. Carlson, A. Wood, B. H. Calhoun, B. P. Otis, A batteryless 19 μw mics/ism-band energy harvesting body sensor node soc for exg applications, IEEE J. Solid State Circuits 48 (1) (2013) 199–213. doi:10.1109/JSSC.2012.2221217.
URL https://doi.org/10.1109/JSSC.2012.2221217

[27] S. R. Sridhara, M. DiRenzo, S. Lingam, S. Lee, R. Blazquez, J. Maxey, S. Ghanem, Y. Lee, R. Abdallah, P. Singh, M. Goel, Microwatt embedded processor platform for medical system-on-chip applications, IEEE Journal of Solid-State Circuits 46 (4) (2011) 721–730. doi:10.1109/JSSC.2011.2108910.

[28] D. Zoni, A. Galimberti, W. Fornaciari, Efficient and scalable fpga-oriented design of qc-ldpc bit-flipping decoders for post-quantum cryptography, IEEE Access 8 (2020) 163419–163433. doi:10.1109/ACCESS.2020.3020262.

[29] D. Zoni, A. Galimberti, W. Fornaciari, Flexible and scalable fpga-oriented design of multipliers for large binary polynomials, IEEE Access 8 (2020) 75809–75821. doi:10.1109/ACCESS.2020.2989423.

[30] L. Lai, N. Suda, V. Chandra, Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus (2018). arXiv:1801.06601.

[31] K. Majumder, U. Bondhugula, A flexible FPGA accelerator for convolutional neural networks, CoRR abs/1912.07284. arXiv:1912.07284.
URL http://arxiv.org/abs/1912.07284

[32] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, L. Benini, Design and evaluation of smallfloat simd extensions to the risc-v isa, in: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), 2019, pp. 654–657. doi:10.23919/DATE.2019.8714897.

[33] F. de Dinechin, B. Pasca, Designing custom arithmetic data paths with FloPoCo, IEEE Design & Test of Computers 28 (4) (2011) 18–27.

[34] F. de Dinechin, Reflections on 10 years of FloPoCo, in: 26th IEEE Symposium of Computer Arithmetic (ARITH-26), 2019.

[35] S. Galal, O. Shacham, J. S. Brunhaver II, J. Pu, A. Vassiliev, M. Horowitz, Fpu generator for design space exploration, in: 2013 IEEE 21st Symposium on Computer Arithmetic, 2013, pp. 25–34. doi:10.1109/ARITH.2013.27.

[36] F. Glaser, S. Mach, A. Rahimi, F. K. Gürkaynak, Q. Huang, L. Benini, An 826 mops, 210uw/mhz unum alu in 65 nm, in: 2018 IEEE International Symposium on Circuits and Systems (ISCAS), 2018, pp. 1–5. doi:10.1109/ISCAS.2018.8351546.

[37] G. Henry, P. T. P. Tang, A. Heinecke, Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations, in: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), 2019, pp. 69–76. doi:10.1109/ARITH.2019.00019.

[38] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, L. Benini, A transprecision floating-point architecture for energy-efficient embedded computing, in: 2018 IEEE International Symposium on Circuits and Systems (ISCAS), 2018, pp. 1–5. doi:10.1109/ISCAS.2018.8351816.

[39] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, S. Borkar, A 1.45ghz 52-to-162gflops/w variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm cmos, in: 2012 IEEE International Solid-State Circuits Conference, 2012, pp. 182–184. doi:10.1109/ISSCC.2012.6176987.

[40] A. Bocco, Y. Durand, F. De Dinechin, Smurf: Scalar multiple-precision unum risc-v floating-point accelerator for scientific computing, in: Proceedings of the Conference for Next Generation Arithmetic 2019, CoNGA'19, Association for Computing Machinery, New York, NY, USA, 2019. doi:10.1145/3316279.3316280.
URL https://doi.org/10.1145/3316279.3316280

[41] Risc-v "p" extension proposal, version 0.9.11-draft20211209 (2019).
URL https://github.com/riscv/riscv-p-spec/raw/5a12c90b2c206c501a4489eb79e5d4d46afa1014/P-ext-proposal.pdf

[42] D. Koene, Implementation and evaluation of packed-simd instructions for a risc-v processor, Master's thesis, Delft University of Technology (2021).
URL https://repository.tudelft.nl/islandora/object/uuid%3Ac4162ff8-9419-4434-852d-c1c3297df808

[43] F. Zaruba, 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 Transactions on Very Large Scale Integration (VLSI) Systems 27 (11) (2019) 2629–2640. doi:10.1109/TVLSI.2019.2926114.

[44] Y.-R. Chen, H.-H. Liao, C.-H. Chang, C.-C. Lin, C.-L. Lee, Y.-M. Chang, C.-C. Yang, J.-K. Lee, Experiments and optimizations for tvm on risc-v architectures with p extension, in: 2020 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), 2020, pp. 1–4. doi:10.1109/VLSI-DAT49148.2020.9196477.

[45] A. Waterman, K. Asanovíc, The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2 (2019).
URL https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[46] G. Scotti, D. Zoni, A fresh view on the microarchitectural design of fpga-based risc cpus in the iot era, Journal of Low Power Electronics and Applications 9 (2019) 19. doi:10.3390/jlpea9010009.

[47] L. N. Pouchet, PolyBench/C 4.1.
URL http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[48] D. Zoni, A. Galimberti, W. Fornaciari, An fpu design template to optimize the accuracy-efficiency-area trade-off, Sustainable Computing: Informatics and Systems 29 (2021) 100450. doi:https://doi.org/10.1016/j.suscom.2020.100450.
URL https://www.sciencedirect.com/science/article/pii/S2210537920301761

Appendix A. EDP and MRE raw data

This appendix contains the results in terms of energy-delay product (see Table A.4) and mean relative error (see Table A.5) collected during the design space exploration (DSE) campaign. For each application, we analyzed nine binary versions, i.e., a

Table A.4 Energy-delay product (EDP) results for all considered SoCs, normalized with respect to the EDP for SoC_{IM+F32}, as defined according to Equation (4). Normalized EDP values of SoC_{IM} and SoC_{IM+Zm} are reported for four different fixed-point formats, i.e., $Q_{32,23}$, $Q_{32,15}$, $Q_{32,10}$, and $Q_{32,7}$, while normalized EDP values for floating-point execution are reported for SoCs with F_{32}, F_{24}, F_{19}, and F_{16} hardware support.

	Normalized EDP (with respect to SoC_{IM+F32})											
	SoC_{IM}				SoC_{IM+F}				SoC_{IM+Zm}			
Bench	$Q_{32,23}$	$Q_{32,15}$	$Q_{32,10}$	$Q_{32,7}$	F_{32}	F_{24}	F_{19}	F_{16}	$Q_{32,23}$	$Q_{32,15}$	$Q_{32,10}$	$Q_{32,7}$
2mm	1,357	1,298	1,192	1,192	1,000	0,923	0,806	0,784	0,636	0,620	0,606	0,606
3mm	1,121	1,060	1,026	1,026	1,000	0,927	0,798	0,776	0,656	0,618	0,618	0,599
bicg	1,194	1,162	1,090	1,090	1,000	0,949	0,790	0,768	0,674	0,652	0,631	0,631
cholesky	6,854	5,988	5,615	5,167	1,000	0,923	0,809	0,831	0,764	0,721	0,910	1,045
doitgen	1,111	1,146	1,146	1,146	1,000	0,973	0,894	0,845	0,666	0,667	0,667	0,667
gemm	1,257	1,219	1,236	1,196	1,000	0,952	0,832	0,811	0,645	0,625	0,638	0,638
gesummv	1,169	1,157	1,138	1,138	1,000	0,971	0,861	0,837	0,691	0,671	0,644	0,644
lu	2,462	2,361	2,282	2,280	1,000	0,913	0,787	0,761	0,613	0,603	0,586	0,586
mvt	1,083	1,051	1,019	0,986	1,000	0,930	0,801	0,781	0,632	0,624	0,606	0,606
symm	1,466	1,423	1,334	1,334	1,000	0,923	0,770	0,748	0,637	0,599	0,580	0,580
syr2k	1,549	1,499	1,450	1,450	1,000	0,947	0,790	0,768	0,673	0,650	0,629	0,629
syrk	1,414	1,414	1,354	1,354	1,000	0,973	0,848	0,824	0,692	0,670	0,684	0,684
trisolv	4,132	3,634	3,466	3,347	1,000	0,927	0,765	0,755	0,617	0,596	0,600	0,600
trmm	1,038	0,987	0,958	0,958	1,000	0,933	0,818	0,798	0,719	0,657	0,637	0,637
Average	1,943	1,814	1,736	1,690	1,000	0,940	0,812	0,792	0,665	0,641	0,645	0,654

Table A.5 Accuracy of the results for all considered SoCs, expressed in terms of mean relative error (MRE) with respect to the results obtained from SoC_{IM+F32}, as defined according to Equation (5). MRE values of SoC_{IM} and SoC_{IM+Zm} are reported for four different fixed-point formats, i.e., $Q_{32,23}$, $Q_{32,15}$, $Q_{32,10}$, and $Q_{32,7}$, while MRE values for floating-point execution are reported for SoCs with F_{32}, F_{24}, F_{19}, and F_{16} hardware support.

	Mean Relative Error (with respect to SoC_{IM+F32})											
	SoC_{IM}				SoC_{IM+F}				SoC_{IM+Zm}			
Bench	$Q_{32,23}$	$Q_{32,15}$	$Q_{32,10}$	$Q_{32,7}$	F_{32}	F_{24}	F_{19}	F_{16}	$Q_{32,23}$	$Q_{32,15}$	$Q_{32,10}$	$Q_{32,7}$
2mm	0,000%	0,013%	0,376%	2,995%	0,000%	0,001%	0,045%	0,343%	0,000%	0,013%	0,376%	2,995%
3mm	0,004%	0,852%	25,46%	100,0%	0,000%	0,003%	0,065%	0,753%	0,004%	0,852%	25,46%	100,0%
bicg	0,000%	0,011%	0,340%	2,907%	0,000%	0,001%	0,040%	0,343%	0,000%	0,011%	0,340%	2,907%
cholesky	0,000%	0,003%	0,082%	0,604%	0,000%	0,005%	0,154%	0,967%	0,000%	0,003%	0,092%	0,695%
doitgen	0,000%	0,000%	0,000%	0,000%	0,000%	0,000%	0,045%	0,000%	0,000%	0,000%	0,000%	0,000%
gemm	0,000%	0,010%	0,264%	2,687%	0,000%	0,001%	0,051%	0,286%	0,000%	0,010%	0,264%	2,687%
gesummv	0,000%	0,001%	0,036%	0,794%	0,000%	0,001%	0,069%	0,651%	0,000%	0,001%	0,036%	0,794%
lu	0,000%	0,001%	0,033%	0,251%	0,000%	0,005%	0,108%	0,893%	0,000%	0,001%	0,033%	0,251%
mvt	0,000%	0,010%	0,273%	2,329%	0,000%	0,001%	0,036%	0,486%	0,000%	0,010%	0,273%	2,329%
symm	0,000%	0,003%	0,111%	0,898%	0,000%	0,001%	0,034%	0,429%	0,000%	0,003%	0,111%	0,898%
syr2k	0,000%	0,007%	0,183%	1,678%	0,000%	0,001%	0,048%	0,378%	0,000%	0,007%	0,183%	1,678%
syrk	0,000%	0,001%	0,034%	1,178%	0,000%	0,000%	0,037%	0,384%	0,000%	0,001%	0,034%	1,178%
trisolv	0,000%	0,007%	0,211%	1,934%	0,000%	0,018%	0,589%	7,870%	0,000%	0,007%	0,211%	1,934%
trmm	0,000%	0,007%	0,211%	1,700%	0,000%	0,001%	0,048%	0,316%	0,000%	0,007%	0,211%	1,700%
Average	0,000%	0,066%	1,972%	8,568%	0,000%	0,003%	0,098%	1,007%	0,000%	0,066%	1,973%	8,575%

floating-point one, four fixed-point ones leveraging the `M` ISA extension, and four fixed-point ones using the proposed `Zm` ISA extension. The floating-point binary is executed on four FPUs implementing different formats. We note that no precision tuning optimization was employed to optimize the fixed-point data format, since such research falls outside the scope of the manuscript. The four employed fixed-point formats were chosen to span across the range of bitwidth for the integer and fractional parts, maintaining the overall bitwidth equal to 32 bits. To this end, the obtained results are conservative with respect to the ones that could have been obtained by combining our fixed-point architecture with a precision tuning framework.

Davide Zoni is Assistant Professor at Politecnico di Milano, Italy. He published more than 50 papers in journals and conference proceedings. His research interests include RTL design and optimization of complex embedded systems with emphasis on low power methodologies and hardware security. He filed two patents on cyber-security and he won the Switch2Product

competition in 2019.

Andrea Galimberti, MSc, is a PhD student at Politecnico di Milano, Italy. He received his MSc degree in 2019 in Computer Science and Engineering at Politecnico di Milano. His research interests include computer architectures, hardware-level countermeasures to side-channel attacks and design of hardware accelerators.