An FPU design template to optimize the accuracy-efficiency-area trade-off

Davide Zoni *, Andrea Galimberti, William Fornaciari

DEIB – Politecnico di Milano, Milan, 20133, Italy

A R T I C L E I N F O

Keywords:
Floating Point Units (FPU)
Accuracy-Cost-energy tradeoff
Run-time optimization

A B S T R A C T

Modern embedded systems are in charge of an increasing number of tasks that extensively employ floating-point (FP) computations. The ever-increasing efficiency requirement, coupled with the additional computational effort to perform FP computations, motivates several microarchitectural optimizations of the FPU. This manuscript presents a novel modular FPU microarchitecture, which targets modern embedded systems and considers heterogeneous workloads including both best-effort and accuracy-sensitive applications. The design optimizes the EDP-accuracy-area figure of merit by allowing, at design-time, to independently configure the precision of each FP operation, while the FP dynamic range is kept common to the entire FPU to deliver a simpler microarchitecture. To ensure the correct execution of accuracy-sensitive applications, a novel compiler pass allows to substitute each FP operation for which a low-precision hardware support is offered with the corresponding soft-float function call. The assessment considers seven FPU variants encompassing three different state-of-the-art designs. The results on several representative use cases show that the *binary32* FPU implementation offers an EDP gain of 15%, while, in case the FPU implements a mix of *binary32* and *bfloat16* operations, the EDP gain is 19%, the reduction in the resource utilization is 21% and the average accuracy loss is less than 2.5%. Moreover, the resource utilization of our FPU variants is aligned with the one of the FPU employing state-of-the-art, highly specialized FP hardware accelerators. Starting from the assessment, a set of guidelines is drawn to steer the design of the FP hardware support in modern embedded systems.

1. Introduction

Modern embedded systems, especially those at the edge of the computing continuum, are no longer only smart sensors, but also general-purpose computing platforms performing data-processing, for which the computational efficiency is a standing design requirement. To this end, the design of (i) efficient hardware accelerators [1,2] and (ii) run-time energy-performance strategies [3] represents the de-facto solution to cope with the requirements of these new workload scenarios. In particular, such workloads are strongly heterogeneous, encompassing both critical and best-effort classes of applications. The former exhibit strict performance, security and accuracy requirements, while the latter can be executed employing a *best-effort* strategy. In a similar manner, the computation can operate on both integer and floating-point (FP) data. In particular, FP computation is at the core of the novel machine learning applications to support model training and to implement trained models [4] in a way that avoids the use of costly and approximate solutions, i.e., uniform quantized [5] or fixed-point arithmetic [6].

The importance of FP computation in modern embedded systems and the fact that it can contribute up to 50% of the total energy [7] spurred the evolution of *transprecision* and *approximate* computing techniques, both of which trade the accuracy of the final result with the efficiency of the computing platform. Transprecision computing techniques are meant to deliver a predefined accuracy for the final result while performing intermediate computations at a lower precision in order to improve the efficiency. In contrast, approximate computing techniques improve the computational efficiency by leveraging low-precision FP hardware support at the cost of a reduction in the accuracy of the final result.

However, current transprecision and approximate state-of-the-art solutions severely limit the optimal efficiency of modern embedded computing platforms, for which a sustainable, three-dimensional design space made of: (i) efficiency, i.e., Energy-Delay-Product (EDP), (ii) accuracy, and (iii) resource utilization must be considered. Transprecision computing solutions offer novel microarchitectures to support multiple implementations for the same FP operations and a huge flexibility in the design of the FP hardware support, but they impose non-negligible changes to the compiler in order to support operations with different

* Corresponding author.
E-mail addresses: davide.zoni@polimi.it (D. Zoni), andrea.galimberti@polimi.it (A. Galimberti), william.fornaciari@polimi.it (W. Fornaciari).

FP data formats. Instead, approximate computing solutions deliver low-accuracy results and they impose the use of inefficient FP software libraries, with up to 35× EDP degradation (see Section 4.3), to ensure the correct execution of mission-critical, i.e., accuracy-sensitive, applications.

We argue that the designer of an embedded system should have the option to flexibly and independently implement the optimal FP hardware support for each FP operation, with two main objectives: (i) to maximize the efficiency subject to the area and power constraints imposed by the workload at hand, and (ii) to ensure an efficient interoperability between different FP data formats, to minimize the complexity of the FP hardware support [4].

Paper contributions – We present a modular and efficient FPU design template and a framework to trade the efficiency, accuracy and resource utilization design metrics. In particular, our solution accounts for heterogeneous workloads comprised of mixed integer-float computations, as well as for the presence of both best-effort and accuracy-sensitive applications, thus improving the state-of-the-art in three directions:

- Modular FPU design framework – Depending on the target workload and the resource utilization constraints, our framework allows to deliver FP hardware support where the precision of each FP operation can be independently selected at implementation-time. Moreover, a common dynamic range is maintained across the entire FPU to simplify the hardware support and, thus, to further optimize the efficiency-area trade-off and to guarantee the interoperability between different FP data formats.
- Compile-time flexibility – The FPU design framework is coupled with an LLVM compiler pass, that can optionally and selectively transform the FP instructions for which a low precision hardware support is offered into the corresponding function calls to the soft-float library. Such architecture allows to effectively support heterogeneous workloads made of accuracy-sensitive and best-effort applications even when some FP operations are supported by low precision hardware to deliver the optimal EDP-area trade-off. We note that the compiler pass is optional and it has the purpose of increasing the flexibility in supporting accuracy-sensitive applications even on low-precision FP hardware.
- Guidelines for optimal FPU design – Starting from the IEEE-754 32-bit single-precision FP data format (*binary32*), widely adopted in embedded systems FP computations, we discuss the EDP-accuracy-area trade-off by considering FPU implementations that differ on their FP precision at the granularity of the single operation, i.e., their operations are hardware-supported in a mix of 32-, 24- and 16-bit FP data formats. We compared our FPU instances against (i) a state-of-the-art FPU, and (ii) two FPUs obtained by employing highly specialized 16- and 32-bit FP operations from the *FloPoCo* hardware library [8]. Starting from the results collected by evaluating the several FPU implementations under heterogeneous workloads, we deliver a set of guidelines to effectively drive the FPU design accounting for the EDP, area and accuracy requirements.

The rest of the manuscript is organized in four parts. Section 2 reviews the background and the state-of-the-art related to efficient computing on FP data. The architecture of the proposed FPU is presented in Section 3. Experimental results in terms of EDP, accuracy and resource utilization as well as the FPU design guidelines are discussed in Section 4. Conclusions are drawn in Section 5.

2. Background and related works

This section has a two-fold objective. First, the background on floating-point (FP) data formats is discussed in Section 2.1. Second, Section 2.2 reviews the state-of-the-art solutions to deliver an efficient floating-point computational support.

2.1. Floating-point (FP) formats and arithmetic

The FP format defines the internal representation of real numbers in digital computing systems, regardless of whether the computation is performed in software or in hardware. Due to the finite amount of bits available to represent each number, any floating-point (FP) format can only approximate the value of real numbers. An FP data format specifies the encoding of an FP number f as defined in Eq. (1), where S is the sign, M is the mantissa (or significant), i.e., the fractional part, E is the exponent and b is the base of the exponent.

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

The precision of an FP 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 FP numbers, increases with the number of bits used for the exponent and, to a lesser extent, with the bits used to encode the mantissa. However, the use of a larger number of bits to encode each number imposes a larger memory requirement. Furthermore, the complexity of some commonly-used iterative algorithms grows with the size of the FP format, requiring longer computation to perform operations.

The IEEE 754-2019 standard defines three FP formats, i.e., *binary32*, *binary64*, and *binary128*, to offer different precisions and dynamic ranges (see Fig. 1). Each format shares the binary base and a 1-bit sign representation, while the sizes of exponent and mantissa are different. *binary32*, *binary64* and *binary128* FP representations are respectively encoded by 32, 64 and 128 bits. Traditionally, the use of *binary32* represents the de-facto choice to deliver wide dynamic range and reasonable precision in embedded systems. *binary64* extends the precision of the computation 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 has been introduced to support FP computation in embedded applications for which dynamic range and precision reductions are allowed. However, the limited dynamic range offered by the *binary16* FP format represents an obstacle for the evolution of machine learning techniques. In particular, such applications require a huge dynamic range but they are not precision-sensitive. To this extent, the brain-inspired FP format (*bfloat16*) emerged as an alternative FP representation to *binary16*, able to support machine learning computation on edge devices. *bfloat16* is a 16-bit FP format, as the *binary16* one, also using a binary base for its representation. However, it offers a wider dynamic range than the *binary16*, i.e., the exponent is 8 bits instead of 5, at the cost of a loss in precision, i.e., the mantissa is encoded on 7 bits instead of 10. Moreover, to convert from *binary32* to *bfloat16* the mantissa is simply truncated in its LSBs. Conversely, to convert from *bfloat16* to *binary32* it is sufficient

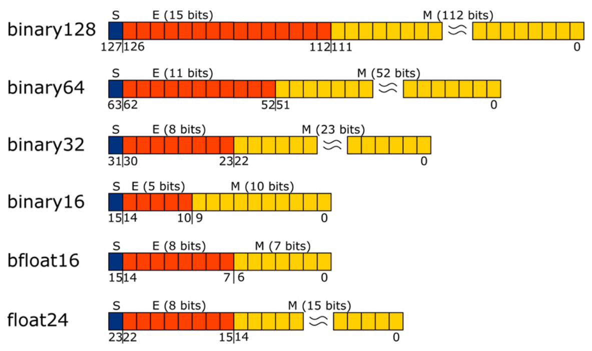


Fig. 1. Structure of the different floating-point formats. Each FP number is specified by a sign (S), an exponent (E) and a mantissa (M). The sign is always represented by 1 bit, while the widths of the exponent and of the mantissa depend on the specific FP format.

to pad the mantissa with 16 zeros as LSBs. Such an advantage ensures full compatibility apart from the precision loss in any source code compiled for *binary32* and executed on a computing platform which implements the *bfloat16* FP format.

Similarly to the *bfloat16* format, we also define a *float24* format which has an 8-bit exponent and a 15-bit mantissa. It offers therefore the same dynamic range as both *binary32* and *bfloat16*, while offering a middle solution between the former and the latter in terms of precision.

2.2. State-of-the-art floating-point solutions

Approximate computing – Depending on either the required efficiency of the platform or the resilience to the accuracy loss of the executed applications, the use of approximate computing is a viable solution to manage FP computations in IoT devices. [9] discussed the increased FPU efficiency due to the reduced precision and the dynamic range of the FPU microarchitecture. [10] proposed an approximate FPU for which its 65 nm implementation shows a 53% power-area product reduction, compared with the *binary32* compliant FPU. We note that the proposed approximate FPU is meant for accuracy-tolerant applications, e.g., image processing. [11] presented an LLVM optimization pass to transform any floating-point instruction in the corresponding set of fixed point operations within the supported subset of the implemented ISA.

Automatic FPU generators – The complexity of floating-point arithmetic motivates the search for tools to automatically generate the RTL description of the FPU. Different tools have been suggested to generate FPUs as either units in the main CPU or hardware accelerators, since the two scenarios have contrasting design requirements. In general, FPU designs for CPUs offer relatively simple floating-point operations to avoid any unbalance in the structure of a pipelined CPU. [12] presented a tool to automatically generate FPU designs for CPUs, with emphasis on the optimization of the Wallace tree multiplier and other basic logic blocks, in order to maximize the efficiency. In contrast, the use of floating-point hardware accelerators allows to design even a complex sequence of floating-point operations as a single instruction that is executed by an ad-hoc accelerator. FloPoCo [8,13] is a framework to design complex floating-point operations in the form of custom accelerators. Such accelerators are far more efficient in executing the complex sequence of FP operations than any FPU implemented as part of the main CPU. However, such accelerators are usually meant to support high-end embedded systems, for which the complexity and the high cost of implementing the accelerator are justified by the required throughput.

Transprecision computing – In the recent past, different floating-point formats emerged to trade the efficiency and the accuracy of the floating-point computations in embedded systems. *binary16* FP reduces both dynamic range and precision compared to *binary32* but it ensures a net efficiency improvement [14]. An FPU design employing a custom width for the operand is proposed in [15]. [16,17] present different FPU microarchitectures considering the *bfloat16* format, while [18] proposes an FPU targeting a custom 16-bit floating-point format. [7] discusses a SIMD transprecision FPU that can adapt the precision of the floating-point computation at run-time to maximize the efficiency. [19,20] extends the work in [7] to trade the accuracy of the results with the possibility of performing a low-precision SIMD FP instruction. In particular, [19] allows to compute either multiple low-precision operations, i.e., *bfloat16* or *binary16*, or a single *binary64* floating-point instruction. Similarly, the work in [21] presents an FMAC design for transprecision computing. The unit can perform multiple concurrent FP operations at low precision. For each low-precision result a bit is added to indicate if the result is sufficiently accurate. This bit is used to trigger the re-execution of those operations for which the low-precision computation delivers an insufficiently accurate result. Such transprecision FPU is meant for high-performance computing solutions since its huge monolithic design also requires non-negligible changes to both the ISA and the compiler to support multiple FP data formats as well as multiple execution modes for each FP instruction.

3. Methodology

This section describes the microarchitecture of the proposed modular floating-point unit (FPU). The modular design approach is meant to optimize the trade-off between the energy-delay product (EDP), the resource utilization and the accuracy of the computation. The EDP-area trade-off is optimized by allowing to configure the precision of the FP data format employed by each operation, while retaining a dynamic range common to the entire FPU to minimize the complexity of the hardware. Without loss of generality and according to the FP computations in the embedded systems domain, we employ the dynamic range of the widely used *binary32* FP data format, i.e., 8 bits for the exponent, while the width of the mantissa, which determines the precision of the floating-point representation, can be configured at any value comprised between 1 and 23 bits. By keeping the same dynamic range across the entire FPU, we allow to seamlessly and efficiently support a large variety of embedded applications, also encompassing machine learning tasks, for which the dynamic range of the *binary32* format has emerged as the best choice to balance the accuracy of the results and the computational efficiency in the embedded systems domain. To support accuracy-sensitive applications, an LLVM compiler pass can optionally and selectively transform the FP instructions for which a low precision hardware support is offered into the corresponding function calls to the soft-float library which, in turn, are selected to be *binary32*-compliant.

Fig. 2 depicts the top-level view of the FPU microarchitecture as made of three parts. The FPU is input- and output-registered to minimize the timing penalty on the CPU pipeline that employs our design. In particular, both the operands and the operational code (*opcode*) are input-registered, and the final result is registered after the rounding stage. We note that the intermediate results produced by each functional unit in the FPU are registered before the rounding stage, which is shared.

The rest of this section discusses the three stages of the FPU depicted in Fig. 2. Finally, the compiler pass to transform selected floating-point (FP) operations from their hard-float form into the corresponding soft-float one is addressed in Section 3.1.

Operand-processing stage – It prepares the operands to perform one of the implemented FPU operations. The stage is common to the entire FPU and it elaborates up to one FPU operation per clock cycle. Each operand is normalized according to the *binary32* FP format, i.e., the width of the mantissa (M) is augmented by one Most-Significant-Bit (MSB) to account for the hidden bit of the *binary32* format (1., or 0. for denormalized numbers) and the exponent (E) is modified accordingly (see *Operand Extension* in Fig. 2). We note that in case of a low-precision FP data input operand, its mantissa is right-padded with zeros to fit the 23-bit width. The *Operand-Extension* sets the new MSB of M and the LSB of E, also considering if the operand is encoding a special case number (see *Operand-Check* in Fig. 2), i.e., Zero (isZer), Infinite (isInf), Not-a-Number (isSN and isQN), Denormalized (isDen).

Floating-point computation stage (*FP-Computation*) – It takes in input the opcode, the operands (sign, extended mantissa and extended exponent) and the control bit-vector for each operand (*isZer*, *isInf*, *isSN* and *isQN*, *isDen*). The actual value of the result depends on the opcode which activates only the hardware module in charge of performing the required operation. To preserve the timing of the FPU, the output of each operation module is registered before moving to the last stage, i.e., *Rounding*. Such strategy is required to trade the support of a fully-pipelined FPU design with the area-constrained requirements that impose to share logic to reduce the area footprint. At design-time, each operation can be configured to support either the *binary32* FP data format or any other FP data format with the same dynamic range and a mantissa width comprised between 1 and 23 bits, e.g., the *bfloat16* FP data format which has a 7-bit mantissa. The implementation of low precision FP hardware, e.g., *bfloat16*-compliant operations, allows: (i) to reduce the resource utilization, and (ii) to improve the EDP, while

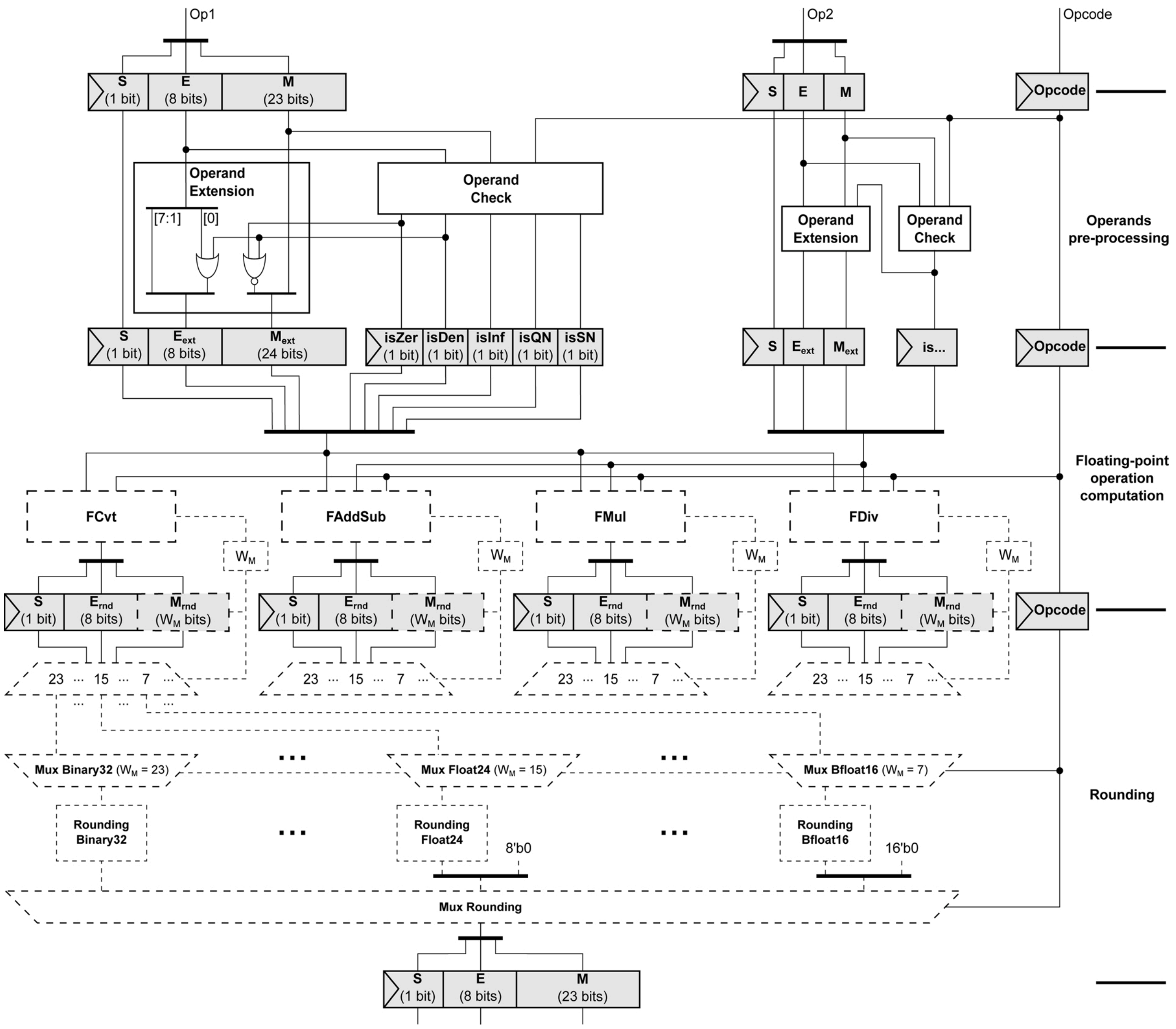


Fig. 2. Top view of the proposed floating-point unit (FPU), which is organized in three stages. First, the operands are checked and prepared regardless of the operations to be executed. The second stage is made of the operations implementing the corresponding ISA instructions. The last stage is common to the entire FPU and it performs the rounding of the result.

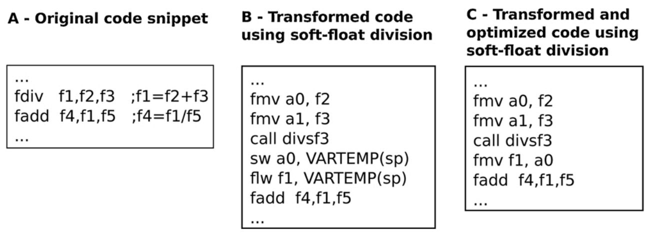
accepting a loss in the accuracy of the computation. The implemented FP operations are based on the architecture described in [22]. In particular, the division architecture implements the Goldschmidt multiplicative algorithm [23], thus leveraging the available FPGA DSPs to efficiently perform inner multiplications.

Fig. 2 shows with dashed lines the elements of the FPU that are configurable at design-time. The functional units dedicated to each FP operation, the size of the register for the resulting mantissa and the rounding to be performed on the result are determined by the chosen floating-point format of each operation. For example, if the multiplication operation is configured to support the *bfloat16* format, the functional unit will be implemented as *FMul16*, i.e., its internal datapath will be limited to 16 instead of the 32 bits of the *binary32* format, and the M_{rnd} register for the output mantissa will be of 10 bits, 7 bits for the mantissa and 3 bits for the rounding.

Rounding stage – It takes as input the result of the *FP-Computation* stage, in terms of the sign, the exponent and the mantissa, augmented with 3 LSBs and two MSBs, and it produces the final result. In particular, the 3 LSBs, i.e., Guard (*G*), Round (*R*) and Sticky (*S*) bits, are used to round the intermediate result to produce the final value. The proposed FPU can be implemented to be compliant with all the IEEE 754 rounding modes, which have been implemented as described in [22]. Depending on the actual FP data formats used by each operation, an instance of the corresponding rounding logic is implemented. Each operation is connected at design-time to its corresponding rounding logic depending on its FP data format. As for the *FP-Computation* stage, dashed lines show the FPU elements that are determined at design-time. We note that the output of each rounding logic module is right-padded with zeros to fit the 32-bit output width. For example, the result of the *bfloat16* rounding is padded with 16 0 s, making the output of the FPU always compliant with the 32-bit floating-point format, i.e., the output mantissa *M* is always 23-bit wide.

3.1. LLVM transformation pass

Traditionally, the FP instructions are either hardware- (hard-float) or software- (soft-float) supported. Hardware support delivers an FPU to implement the operations needed to execute the FP instructions of the target ISA. In contrast, software support delivers a set of software routines leveraging the integer hardware resources of the CPU to implement the FP instructions of the ISA. Considering the detailed FPU implementation employing the dynamic range of the *binary32* FP data format, we allow to selectively implement the hardware support for each FP operation using a mantissa between 1 and 23 bits. To this end, our LLVM pass allows to selectively employ, at compile time, the software support at the granularity of the single FP operation to restore the accuracy of



A - Original code snippet

```
...
fdiv  f1,f2,f3   ;f1=f2+f3
fadd  f4,f1,f5   ;f4=f1/f5
...
```

B - Transformed code using soft-float division

```
...
fmv a0, f2
fmv a1, f3
call divsf3
sw a0, VARTEMP(sp)
flw f1, VARTEMP(sp)
fadd  f4,f1,f5
...
```

C - Transformed and optimized code using soft-float division

```
...
fmv a0, f2
fmv a1, f3
call divsf3
fmv f1, a0
fadd  f4,f1,f5
...
```

Fig. 3. Example of the proposed transformation compiler pass applied to a code snippet. The FP division (`fdiv`) is the producer of the `f1` register operand for the subsequent FP addition (`fadd`). The compiler pass transforms and optimizes the code to allow the use of the software implementation of the FP division (`divsf3`).

the *binary32* FP data format even in presence of a low-precision hardware support for some or all the FP operations.

The compiler pass described in this section achieves such goal by delivering a binary code compatible with a mixed hard-soft-float architecture. In addition to the standard fully hard- or soft-float support, our solution allows the user to specify the hardware or software support for each of the four FP operations, i.e., addition, subtraction, multiplication and division. To this extent, the compiler produces code for standard hard-float (soft-float) architectures when the user selects to implement all the FP operations in hardware (software). However, all the intermediate hard-soft-float combinations are possible, thus increasing the flexibility of the produced binary code. The transformation pass has been developed for the LLVM-9.0 compiler infrastructure [24] and it operates at both the Intermediate Representation and the Code Generation stages. The pass is meant to fix three aspects: (i) make a soft-call for any instruction for which the user specifies a software implementation, (ii) fix the producer-consumer relationships between 2 FP instructions where the producer is implemented as soft-float and the consumer as hard-float, and (iii) vice-versa. Transformations due to (ii) and (iii) make use of local variables, i.e., memory locations to correctly manage data at the integer-to-float and float-to-integer boundaries. In particular, the result of a FP instruction is stored in the FP register file, while the one of an integer instruction is stored in the integer register file. Each register file is only compatible with either FP or integer instructions. To this extent, the use of a local variable to store a result and to load it afterwards allows to exchange data between float and integer domains.

Intermediate Representation (IR) stage – At this stage the pass aims to identify the complex FP expressions made of FP instructions, where at least one of them must be implemented as soft-float, as in the case (ii), i.e., soft-float producer, hard-float consumer. To this extent, the pass analyzes the producer-consumer relationship between the FP instructions. For each FP instruction (*insnProducer*) that must be implemented as a soft-float library call, all its consumers are analyzed. A change in the IR code is performed if one of the users is an FP instruction that must be implemented as hard-float (*insnConsumer*). First, a new local variable is created. Second, an integer-store instruction is added to save the value produced by *insnProducer*. Last, an FP-load instruction gets the stored value back to make it available to the subsequent *insnConsumer* instruction. Such scheme breaks down the type-dependency problem, since: (i) the output of the *insnProducer* instruction must be an integer, namely *insnProducer* is supported as a soft-float operation, thus assuming that no FP-register file is present, and (ii) *insnConsumer* is expecting an FP-type operand. We note that a more optimized version of the transformation pass allows to replace the store-load pair of instructions with a single move instruction.

Fig. 3 shows the effect of the transformation pass applied to a snippet of assembly code made of two FP instructions, i.e., division and addition, when the FP division is not supported in hardware. In the original code snippet, the FP division produces the value of the f1 register, that is consumed by the FP addition (see Fig. 3A). Considering the operands of the FP division, the transformation pass adds two FP move instructions to transfer the content of the `f2` and `f3` FP registers to the `a0` and `a1` integer registers. We note that the software implementation of the FP division, i.e., `divsf3`, takes in input integer registers and the return value is stored back in `a0`. To this end, the initial version of the modified code contains a pair of store and load instructions, to push back the value produced by the software FP division routine and to reload its value into the `f1` FP register (see Fig. 3B). As a further performance optimization, the optimized version of the code replaces the expensive store-load pair of instructions with a single register-to-register instruction, i.e., `mv f1,a0` (see Fig. 3C).

Code generation (CodeGen) stage – At this stage the pass aims at fixing (i) and (ii). The corresponding soft-float library calls replace each FP instruction for which no or low-precision FP support is offered. Moreover, the operands of the soft-float library calls are analyzed; if the producer is an hard-float FP instruction then an additional local variable is generated to serve as destination for the FP-store, and as source of the subsequent integer-load instructions. Finally, such variables are added to the code.

4. Experimental evaluation

This section reports the experimental results in terms of efficiency, i.e., Energy-Delay Product (EDP), accuracy and resource utilization, considering different FPU variants implemented by means of the proposed FPU design framework. In particular, each FPU instance has been integrated into an embedded RISC-V System-on-Chip (SoC) [25] to deliver an FPGA implementation of the entire computing platform. To provide a consistent comparison, a state-of-the-art FPU design, i.e., the FPU from the *ORPSoC-v3-mor1kx* project, has been integrated in the same SoC. The *ORPSoC-v3-mor1kx* project offers a mature SoC equipped with an FPU which has been used to realize the AR100 power management unit of the Allwinner commercial SoCs [26]. As a second representative term of comparison, the FloPoCo hardware library [13] has been combined with our FPU design framework to deliver two FPU variants, i.e., *binary32* and *bfloat16*, where the highly specialized hardware primitives from the FloPoCo library are used to generate the hardware support for each FP operation. We note that the FPU implementation is the only changing element of the considered computing platform and, thus, the reference SoC represents the computational support of choice to fairly assess the effectiveness of our solution. The rest of the discussion is organized in four parts. Section 4.1 presents both the software and hardware experimental setups and the details of the implemented FPU variants. Section 4.2 discusses the experimental results in terms of the EDP-area trade-off, while Section 4.3 describes the results under the EDP-accuracy perspective. Last, Section 4.4 focuses on a set of guidelines to drive the design of an efficient FPU for embedded systems.

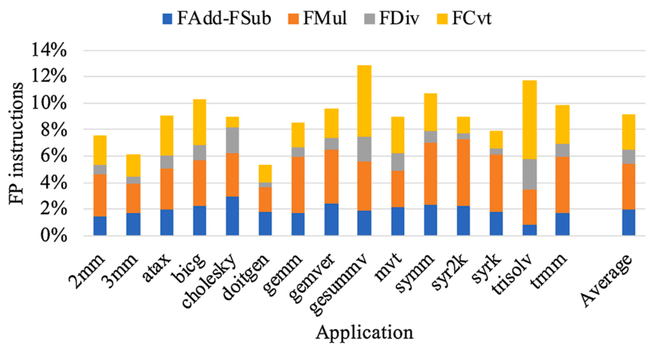


Fig. 4. Percentage of floating-point instructions for each benchmark application. Floating-point instructions are split in additions-subtractions, multiplications, divisions and conversions.

4.1. Experimental setup

Software setup – To assess the proposed FPU, we employed applications from the *PolyBench* benchmark suite [27], which offers a set of programs meant to be FP-intensive [19].

The mix of FP instructions for each application is reported in Fig. 4, as a percentage with respect to the total number of instructions executed by the application. In particular, FP instructions are split in additions-subtractions (FAdd-FSub), multiplications (FMul), divisions (FDiv) and conversions (FCvt). Due to the focus of this work, results for applications that are not using FP instructions are not reported. All the applications have been executed in bare-metal mode on the reference SoC considering different FPU implementations of our design as well as the state-of-the-art FPU (FPU_{mor1kx}). Each application is written in ANSI C and it has been compiled using the vanilla *LLVM-9.0* compiler toolchain, to which we added our LLVM compiler pass. Depending on the experiment at hand, the compiler pass can selectively convert from hard- to soft-float. Last, we employed the 32-bit RISC-V variant of the *gcc-8.2.0* compiler for the linking, since the 32-bit RISC-V target is not yet supported by the LLVM linking framework.

Hardware setup – To provide a complete evaluation of the proposed FPU, we employed a configurable state-of-the-art System-on-Chip [25] conceived for FPGA targets. In particular, we made use of an instance of the reference SoC that features a 32-bit in-order RISC-V 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 CPU of the reference SoC supports the Integer (I), Integer-Multiply-Division (M) and single-precision floating-point (F) RISC-V 32-bit ISA extensions.

We note that the floating-point support is provided by means of different FPU implementations that are integrated into the reference SoC. The complete SoC has been implemented employing Xilinx Vivado 2018.2, and using a 100 MHz clock frequency on the Digilent Nexys 4 DDR prototyping board. The Digilent board features a Xilinx Artix-7 100 FPGA chip that represents a mid-range cost-effective FPGA whose main features are 63,400 LUTs, 126,800 Flip-Flops, 135 BRAMs and 240 DSPs. To provide a fair evaluation, we implement each FPU design within the same SoC, employing the Vivado default synthesis and implementation strategies. In particular, a common FPU interface has been designed to fit all the considered FPU variants. To further improve the fairness in the comparison between different FPUs, the FPU interface registers all the input and output signals of the FPU. We note that the overall SoC performance is slightly degraded due to the additional clock cycles imposed by the FPU interface. Indeed, the experimental setup is meant to provide an unbiased scenario that enables a relative, rather than absolute, comparison between the different FPU implementations.

The FPU is therefore the only microarchitectural component that changes across different implementations of the SoC. For each FPU variant, the resource utilization of the FPU and the entire SoC are extracted from the post-implementation netlist, obtained by means of Vivado 2018.2. The power consumption is extracted by performing the post-implementation simulation of all the considered FPU variants, each integrated in the SoC. We note that the power consumption reported for the SoC equipped with each specific FPU variant is obtained as the average among the executions of all the considered benchmarks.

FPU implementations – We evaluated three FPU implementations from the state-of-the-art, i.e., FPU_{mor1kx}, FPU_{FPC32}, and FPU_{FPC16}, as well as four implementations from the proposed FPU design template, i.e., FPU_{32}, FPU_{24}, FPU_{16}, and $FPU_{16//32}$. All the FPU implementations are reported in Table 1, which shows in detail the precision of each hardware-supported FP operation and its corresponding latency, expressed in terms of number of clock cycles. FPU_{mor1kx} is the *binary32*-

Table 1
FPU latency (expressed in terms of clock cycles) considering different FPU implementations. For each operation, either the *bfloat16* (`16 bit`), *float24* (`24 bit`) or *binary32* (`32 bit`) format is employed. The * symbol is used to highlight the possibility to substitute at compile-time the lower-precision, i.e., *bfloat16* and *float24*, hardware support with the use of the *binary32* soft-float implementation.

FPU Impl.	Latency of FPU operations														
	FAdd-FSub			FMul			FDiv			FCvt			FCmp		
	16 bit	24 bit	32 bit	16 bit	24 bit	32 bit	16 bit	24 bit	32 bit	16 bit	24 bit	32 bit	16 bit	24 bit	32 bit
FPU_{32}	–	–	5	–	–	5	–	–	12	–	–	4	–	–	4
$FPU_{16//32}$	5*	–	–	–	–	5	9*	–	–	–	–	4	–	–	4
FPU_{24}	–	5*	–	–	5*	–	–	12*	–	–	4*	–	–	4*	–
FPU_{16}	5*	–	–	5*	–	–	9*	–	–	4*	–	–	4*	–	–
FPU_{FPC32}	–	–	6	–	–	4	–	–	14	–	–	4	–	–	4
FPU_{FPC16}	4*	–	–	3*	–	–	9*	–	–	4*	–	–	4*	–	–
FPU_{mor1kx}	–	–	7	–	–	8	–	–	20	–	–	5	–	–	3

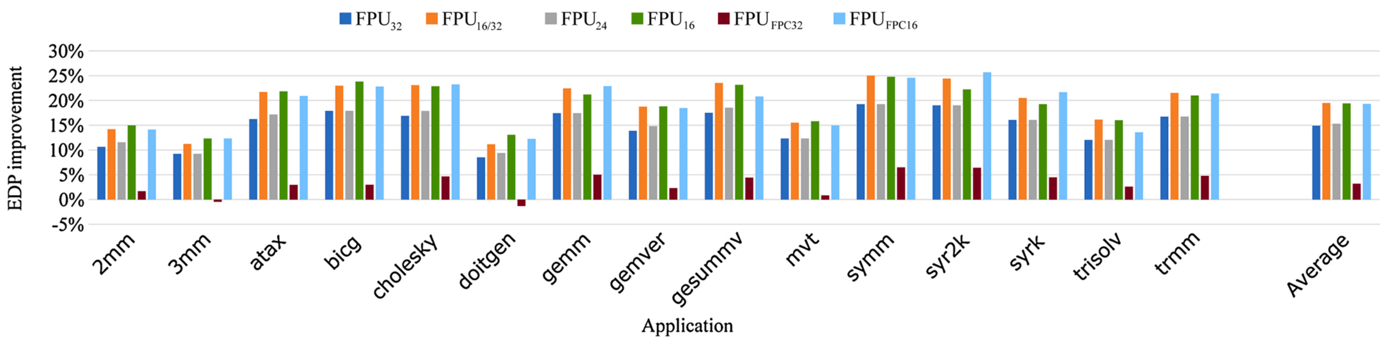


Fig. 5. Energy-delay product (EDP) improvement for the seven considered FPU implementations. Results are normalized with respect to the EDP for the *FPU_{mor1kx}* implementation.

compliant FPU from the *ORPSoC-v3-mor1kx* project and it is used as the reference design for the comparison with all the other FPU implementations. The $\text{FPU}_{\text{FPC32}}$ and $\text{FPU}_{\text{FPC16}}$ designs are two FPU variants implementing the *binary32* and *bfloat16* FP data formats, and, for each FP operation, their hardware support is obtained from integrating the FloPoCo hardware library into our FPU design template architecture. We note that $\text{FPU}_{\text{FPC32}}$ and $\text{FPU}_{\text{FPC16}}$ make use of highly specialized FP hardware accelerators and, thus, they are used to effectively assess the efficiency of our FPU solutions. Considering our FPU variants, the FPU_{32}, FPU_{24} and FPU_{16} instances implement respectively *binary32*, *float24* and *bfloat16* FPUs. In contrast, $\text{FPU}_{16//32}$ implements the *binary32* multiplication, conversions and comparisons, while the addition, subtraction and division operations are implemented according to the *bfloat16* data format. We note that the * symbol, used to mark the latency of *bfloat16* and *float24* operations in Table 1, highlights the possibility to employ, at compile-time, the *binary32* soft-float implementation in place of the hardware support. It is worth noticing that $\text{FPU}_{16//32}$ is meant to highlight an example of the flexibility of our FPU design while, more in general, any FP operation can be independently implemented with a mantissa width comprised between 1 and 23 bits. For example, we note that the *binary32* variant of the proposed FPU (FPU_{32}) offers a 30% latency reduction, on average, compared to the state-of-the-art FPU ($\text{FPU}_{\text{mor1kx}}$) while ensuring the same resource utilization.

4.2. Area-efficiency trade-off

This section addresses the effectiveness of the proposed FPU architecture in terms of efficiency (see Fig. 5), i.e., energy-delay product (EDP), and resource utilization (see Table 2). Fig. 5 reports the efficiency in terms of percentage of the EDP improvement with respect to the reference $\text{FPU}_{\text{mor1kx}}$ design, while the resource utilization for each FPU variant is reported in Table 2. The EDP results are reported for the FPU_{32}, FPU_{24}, FPU_{16}, and $\text{FPU}_{16//32}$ implementations of our design and the $\text{FPU}_{\text{FPC32}}$ and $\text{FPU}_{\text{FPC16}}$ FloPoCo designs. All results are normalized to the results obtained employing the $\text{FPU}_{\text{mor1kx}}$ FPU. In particular, the EDP is defined as the product between the energy consumption and the execution time for a benchmark application. Moreover, for each FPU implementation $i \in \{\text{FPU}_{32}, \text{FPU}_{24}, \text{FPU}_{16}, \text{FPU}_{16//32}, \text{FPU}_{\text{FPC32}}, \text{and } \text{FPU}_{\text{FPC16}}\}$ and application a, the EDP improvement ($\text{EDP}_{\text{impr}_{i,a}}$) is defined by Eq. (2).

$$\text{EDP}_{\text{impr}_{i,a}} = \frac{\text{EDP}_{\text{FPU}_{\text{mor1kx}},a}}{\text{EDP}_{i,a}} - 1 \qquad (2)$$

Table 2 reports the area details for each of the seven considered FPU variants, i.e., FPU_{32}, FPU_{24}, FPU_{16}, $\text{FPU}_{16//32}$, $\text{FPU}_{\text{FPC32}}$, $\text{FPU}_{\text{FPC16}}$, and $\text{FPU}_{\text{mor1kx}}$, in terms of look-up tables (LUTs), flip-flops (FFs) and digital signal processing elements (DSPs). For each resource type and FPU configuration, we report both the total (Tot) and normalized (TotNorm) amount of resources. TotNorm is normalized with respect to the resources used by the state-of-the-art *mor1kx* FPU. Moreover, for each of our FPU implementations, AreaDiff describes the percentage increase or reduction in the area occupation, and it is defined as $\text{TotNorm}_{\text{LUT}} - 1$. Finally, Table 2 also reports, for each FPU implementation, the resource utilization of the entire SoC in terms of LUTs, flip-flops and DSPs.

FPU_{32} **and** $\text{FPU}_{\text{FPC32}}$ – Compared to the state-of-the-art FPU, the FPU_{32} shows an average EDP improvement of 15%. Such gain is motivated by the improved latency of our design (see Table 1) that, however, employs only 3% more resources with respect to the ones required by the reference FPU (see Table 2). In contrast, the $\text{FPU}_{\text{FPC32}}$ highlights a modest 3.5% EDP improvement on average. This is due to the different FP division architecture, that employs the SRT [28] algorithm in place of the Goldschmidt [23] one used in the FPU_{32} design. To this end, the $\text{FPU}_{\text{FPC32}}$ division is slower and larger in size compared to the FPU_{32} one, thus determining a reduction in the achieved EDP improvement. Compared to $\text{FPU}_{\text{mor1kx}}$, the $\text{FPU}_{\text{FPC32}}$ design exhibits a 38% area overhead and its division latency is 14 clock cycles.

$\text{FPU}_{16//32}$**,** FPU_{16} **and** $\text{FPU}_{\text{FPC16}}$ – Considering the *bfloat16* FP data format to implement some ($\text{FPU}_{16//32}$) or all the FP operations (FPU_{16}), the average EDP improves up to 19%. The additional improvement is motivated by both the additional latency optimization due to the use of the *bfloat16* FP format as well as by the lower power consumption of the smaller and simpler *bfloat16* FPU designs. For example, FPU_{16} requires 41% less resources compared to the $\text{FPU}_{\text{mor1kx}}$ design (see Table 2), and the average latency reduction is 33%. We note that $\text{FPU}_{\text{FPC16}}$ shares similar results in terms of both EDP and resource utilization with the

Table 2
Area results for the seven considered FPU implementations. Results are absolute numbers, except for *TotNorm*, which is normalized with respect to *FPU_{mor1kx}*, and *AreaDiff*, which is expressed in percentage.

FPU Impl.	Resource type	FPU resource utilization									SoC resource utilization
		FAdd-FSub	FMul	FDiv	F2I	I2F	FCmp	Tot	TotNorm	AreaDiff	
	LUT	447	273	397	38	100	53	1948	1.03		10,567
FPU_{32}	FF	90	56	193	36	37	2	632	0.71	+ 3%	7839
	DSP	0	2	2	0	0	0	4	1.00		8
	LUT	187	272	241	38	99	53	1490	0.79		10,084
$FPU_{16//32}$	FF	52	56	115	36	37	2	484	0.54	− 21%	7663
	DSP	0	2	1	0	0	0	3	0.75		7
	LUT	276	182	278	31	146	40	1476	0.78		10,113
FPU_{24}	FF	73	31	145	36	29	2	506	0.57	− 22%	7704
	DSP	0	1	1	0	0	0	2	0.5		6
	LUT	171	228	235	26	97	33	1119	0.59		9648
FPU_{16}	FF	57	39	120	36	21	2	425	0.48	− 41%	7632
	DSP	0	0	1	0	0	0	1	0.25		5
	LUT	311	80	1639	158	161	51	2601	1.38		11,176
FPU_{FPC32}	FF	239	50	624	86	74	2	1224	1.38	+ 38%	8407
	DSP	0	2	0	0	0	0	2	0.50		6
	LUT	131	95	313	111	174	31	1070	0.52		9690
FPU_{FPC16}	FF	64	29	164	70	74	2	520	0.58	− 43%	6969
	DSP	0	0	0	0	0	0	0	0.00		4
	LUT	447	273	397	38	100	53	1890	–		10,526
FPU_{mor1kx}	FF	90	56	193	36	37	2	890	-	–	8096
	DSP	0	2	2	0	0	0	4	–		8

FPU_{16} design, thus demonstrating the possibility of improving the EDP-area metrics by reducing the FP precision. In particular, the use of either the SRT (FPU_{FPC16}) or Goldschmidt (FPU_{16}) algorithms to implement the FP division on a 16-bit FP data format does not highlight severe differences in terms of area and EDP, as observed when comparing the 32-bit FPUs, i.e., FPU_{32} and FPU_{FPC32}.

$FPU_{16//32}$ and FPU_{24} – Compared to the FPU_{32} design, we note that the FPU_{24} implementation offers a modest 1.5% EDP improvement and a 22% area reduction. In particular, the modest EDP improvement is due to the fact that both FPU_{32} and FPU_{24} have the same latency for each FP operation, since the smaller mantissa of the latter is not enough to determine a latency reduction. In contrast, $FPU_{16//32}$ offers a 21% resource utilization reduction with respect to the FPU_{mor1kx} (see Table 2), i.e., similar to the one of the FPU_{24}, while the EDP improvement of the $FPU_{16//32}$ is aligned to the one of the 16-bit implementations, i.e., FPU_{16} and FPU_{FPC16}. Compared to the 16-bit implementations, the limited area reduction is due to the use of the *binary32* FP data format for the FMul and FCvt and, thus, the need to implement two rounding modes, i.e., *binary32* for the FMul and FCvt operations and *bfloat16* for all the others. However, the net latency reduction of the $FPU_{16//32}$ with respect to the FPU_{24} allows the former to deliver a higher EDP improvement.

4.3. Accuracy-efficiency trade-off

While the use of a low-precision FPU allows to greatly improve both the EDP and the resource utilization, the target workload can contain critical applications exhibiting an accuracy-sensitive behavior. To this end, this part highlights the accuracy error reported by employing low-precision FPUs, while showing the EDP degradation due to the use of soft-float function calls to restore the *binary32* accuracy of the final results.

Fig. 6 reports the accuracy error, in terms of mean relative error, for each of the considered FPU implementations. For each benchmark application, the results produced by each FPU are compared to the corresponding ones obtained from the FPU_{mor1kx} design, which is fully-compliant with the *binary32* FP data format. As expected, both the FPU_{32} and FPU_{FPC32} designs show a zero accuracy error. In contrast, low-precision FPUs, i.e., FPU_{24}, FPU_{16}, $FPU_{16//32}$, and FPU_{FPC16} exhibit a non-zero accuracy error that is lower than 2.5% on average while, as expected, the accuracy error increases with the reduction in the precision of the FPU. We note that the cholesky benchmark application reports an accuracy error of 33% employing one among the FPU_{24}, FPU_{16}, $FPU_{16//32}$, and FPU_{FPC16}, but in the picture the results are limited to a 3% accuracy error for readability purposes. The accuracy loss for the cholesky application is due to the structure of the application, which performs long sequences of additions and subtractions between small and large floating point numbers. Such operations are known to negatively affect the accuracy of the final result, and this effect worsens with the reduction of the FP format precision.

Despite the limited accuracy error reported in Fig. 6, our compiler pass allows to selectively transform each FP instruction for which low precision hardware is implemented, into the corresponding function call to the soft-float library. The transformation can be selectively applied to accuracy-sensitive applications at compile-time, thus determining an EDP degradation only for critical applications that require full *binary32* accuracy.

Fig. 7 reports the EDP degradation, as defined in Eq. (3), due to the soft-float execution of all the FP instructions for which a lower-precision hardware support is offered.

$$EDP_{degr_{i,a}} = \frac{EDP_{i,a}}{EDP_{FPU_{mor1kx},a}} - 1 \qquad (3)$$

As expected, FPU_{32} and FPU_{FPC32} designs suffer no EDP penalty since each FP operation employs the *binary32* data format, thus avoiding the use of the soft-float support. In particular, FPU_{32} and FPU_{FPC32} show an average EDP improvement against FPU_{mor1kx} of 15% and 3.5%, respectively (see Fig. 5). In contrast, FPU_{16}, FPU_{24} and FPU_{FPC16} highlight an average EDP degradation of 25x, since all the FP instructions are soft-float executed to ensure the *binary32* accuracy for the final result. We note that the $FPU_{16//32}$ design exhibits a 3.5× EDP degradation for accuracy-sensitive applications, since only a portion of the FP instructions must be soft-float executed. In particular, $FPU_{16//32}$ provides hardware support for *binary32* multiplications and conversions, which can be used in place of the corresponding soft-float routines. To this end, the $FPU_{16//32}$ shows an EDP gain of 19% for best-effort applications, an improvement aligned to the one obtained by means of low-precision hardware, while the EDP degradation to execute accuracy-critical applications is limited to 3.5× rather than the 25× EDP degradation observed for 16-bit FPUs.

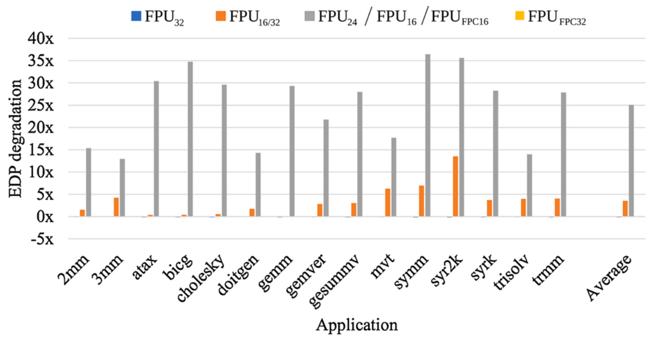


Fig. 7. Energy-delay product (EDP) degradation for the seven FPU implementations, when executing accuracy-sensitive applications. Results are normalized with respect to the EDP for the FPU_{mor1kx} implementation.

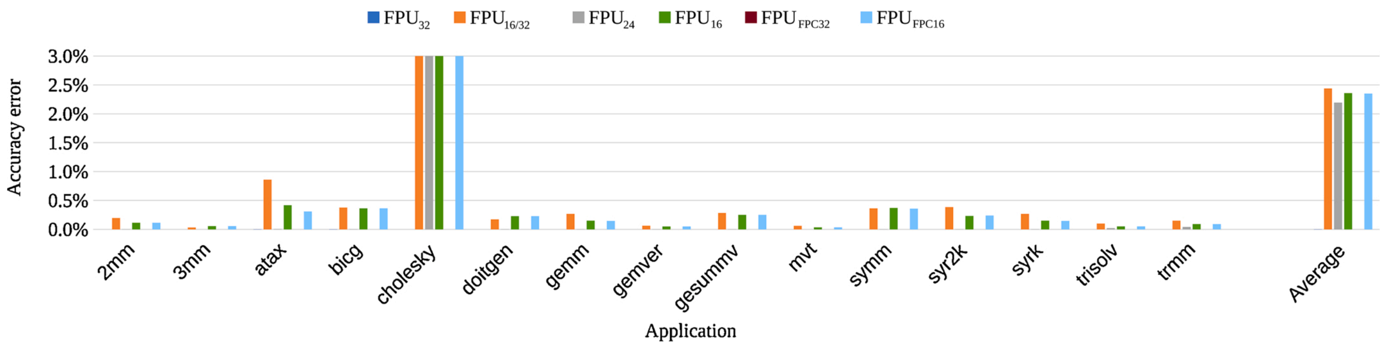


Fig. 6. Accuracy results, expressed in terms of mean relative error, considering the seven FPU implementations. Results are normalized with respect to the EDP for the FPU_{mor1kx} implementation.

4.4. FPU design guidelines

Starting from the proposed modular FPU design and from the results collected considering several FPU implementations, we demonstrated the possibility of optimizing the accuracy-EDP-area trade-off by leveraging a configurable-precision FP support at the granularity of the single operations, once the dynamic range common to the entire FPU has been fixed. This part discusses three guidelines to approach the design of the hardware FP support in modern embedded systems, based on the quantitative analysis we carried out.

Guideline 1 – The use of low precision FP hardware support is suggested when the computing platform executes best-effort workloads, since the average accuracy error is lower than 2.5%.

Guideline 2 – The possibility to independently choose the precision for the hardware support of each FP operation allows to improve the EDP-area-accuracy trade-off in heterogeneous workloads made of a mix of best-effort and accuracy-sensitive applications. Considering an FPU implementing a mix of low- and full-precision hardware support for the FP instructions, the low-precision FP units allow to reduce the latency and resource utilization when best-effort applications are executed, while full-precision FP units can be still employed to execute accuracy-sensitive tasks, thus minimizing the utilization of slow soft-float primitives.

Guideline 3 – Results considering the FPU_{24} highlight the impossibility of improving the EDP-area trade-off via a shallow precision reduction, i.e., 24-bit FP data format instead of the 32-bit one. In particular, FPU_{24} and FPU_{16} share similar accuracy errors (see Fig. 6), thus motivating the aggressive reduction of the width of the mantissa to improve both the EDP and the area metrics.

5. Conclusions

This manuscript presented a novel modular FPU microarchitectural framework, which targets modern embedded systems and considers heterogeneous workloads comprised of both best-effort and accuracy-sensitive applications. The framework optimizes a comprehensive EDP-accuracy-area figure of merit by allowing to independently select the precision of the implemented hardware support at the granularity of the single FP operation, once the dynamic range common to the entire FPU has been fixed. The use of a common dynamic range for the entire FPU simplifies the hardware support, thus reducing the resource utilization and improving the interoperability between different FP data formats. To restore the full precision required by accuracy-sensitive applications even on low-precision hardware support, the FPU hardware design framework is coupled with an LLVM compiler pass, which can selectively transform the FP instructions supported by a low-precision hardware into the corresponding soft-float function calls.

Our design template has been validated by considering several FPU variants, also encompassing one state-of-the-art *binary32*-compliant FPUs and two FPU variants generated by means of the FloPoCo hardware FP library. Our solution demonstrated the possibility to improve the EDP up to 19% over the considered state-of-the-art FPU. Moreover, our FPU variants ensure an equal or lower resource utilization with respect to the highly specialized FP hardware accelerators generated using the FloPoCo library. The discussion of the results is complemented with a set of guidelines that can be employed to drive the design of the FP hardware support in modern embedded systems.

Conflict of interest

The authors declare no conflict of interest.

Declaration of Competing Interest

The authors report no declarations of interest.

References

[1] D. Zoni, A. Galimberti, W. Fornaciari, Flexible and scalable fpga-oriented design of multipliers for large binary polynomials, IEEE Access 8 (2020) 75809–75821.
[2] D. Zoni, A. Galimberti, W. Fornaciari, Efficient and scalable fpga-oriented design of qc-ldpc bit-flipping decoders for post-quantum cryptography, IEEE Access (2020) 1.
[3] D. Zoni, L. Cremona, W. Fornaciari, All-digital control-theoretic scheme to optimize energy budget and allocation in multi-cores, IEEE Trans. Comput. 69 (5) (2020) 706–721.
[4] J. Johnson, Rethinking floating point for deep learning, CoRR (2018) vol. abs/1811.01721, [Online]. Available from: arXiv:1811.01721.
[5] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (2018) 2704–2713.
[6] D.D. Lin, S.S. Talathi, V.S. Annapureddy, Fixed point quantization of deep convolutional networks, Proceedings of the 33rd International Conference on International Conference on Machine Learning – Volume 48, ser. ICML'16 (2016) 2849–2858.
[7] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, L. Benini, A transprecision floating-point platform for ultra-low power computing, 2018 Design, Automation Test in Europe Conference Exhibition (DATE) (2018 March) 1051–1056.
[8] F. de Dinechin, B. Pasca, Designing custom arithmetic data paths with FloPoCo, IEEE Des. Test Comput. 28 (July (4)) (2011) 18–27.
[9] J.Y.F. Tong, D. Nagle, R.A. Rutenbar, Reducing power by optimizing the necessary precision/range of floating-point arithmetic, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 8 (June (3)) (2000) 273–286.
[10] V. Camus, J. Schlachter, C. Enz, M. Gautschi, F.K. Gurkaynak, Approximate 32-bit floating-point unit design with 53% power area product reduction, in: ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference, September, 2016, pp. 465–468.
[11] D. Cattaneo, A. Di Bello, S. Cherubin, F. Terraneo, G. Agosta, Embedded operating system optimization through floating to fixed point compiler transformation, 2018 21st Euromicro Conference on Digital System Design (DSD) (2018 Aug) 172–176.
[12] S. Galal, O. Shacham, J.S. Brunhaver II, J. Pu, A. Vassiliev, M. Horowitz, Fpu generator for design space exploration, 2013 IEEE 21st Symposium on Computer Arithmetic (2013 April) 25–34.
[13] F. de Dinechin, Reflections on 10 years of FloPoCo, in: 26th IEEE Symposium of Computer Arithmetic (ARITH-26), June, 2019.
[14] IEEE Standard for Floating-Point Arithmetic, 2019, pp. 1–84. IEEE Std 754-2019 (Revision of IEEE 754-2008), July.
[15] F. Glaser, S. Mach, A. Rahimi, F.K. Gürkaynak, Q. Huang, L. Benini, An 826 mops, 210uw/mhz unum alu in 65 nm, 2018 IEEE International Symposium on Circuits and Systems (ISCAS) (2018 May) 1–5.
[16] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, D. Mansell, Bfloat16 processing for neural networks, in: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), June, 2019, pp. 88–91.
[17] G. Henry, P.T.P. Tang, A. Heinecke, Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations, 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH) (2019 June) 69–76.
[18] 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), June, 2019, pp. 92–95.
[19] 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), March, 2019, pp. 654–657.
[20] 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), May, 2018, pp. 1–5.
[21] 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 32 nm cmos, 2012 IEEE International Solid-State Circuits Conference (2012 Feb) 182–184.
[22] I. Koren, Computer Arithmetic Algorithms, 2nd ed., A. K. Peters, Ltd., Natick, MA, USA, 2001.
[23] R.E. Goldschmidt, Applications of Division by Convergence, 1964.
[24] C. Lattner, V. Adve, Llvm: a compilation framework for lifelong program analysis transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004, March, 2004, pp. 75–86.
[25] "LAMP-platform," 2019. [Online]. Available from: http://www.lamp-platform.org.
[26] S. Kristiansson, Bare-Metal Introspection Application for the AR100 Controller of Allwinner A31 SoCs, 2016.
[27] L.-N. Pouchet, "PolyBench/C 3.2." [Online]. Available from: http://www.cse.ohio-state.edu/pouchet/software/polybench/.
[28] D.L. Harris, S.F. Oberman, M.A. Horowitz, Srt division architectures and implementations, Proceedings 13th IEEE Symposium on Computer Arithmetic (1997) 18–25.

Davide Zoni is a Post-doc Researcher at Politecnico di Milano, Italy. He published more than 40 papers in journals and conference proceedings. His research interests include RTL design and optimization for multi-cores with particular emphasis on low power methodologies and hardware-level countermeasures to side-channel attacks. He received two HiPEAC collaboration grants in 2013 and 2014, two HiPEAC industrial grant in 2015 and 2017 and he won the Switch2Product competition in 2019. He filed two patents on cyber-security. He is also the principal investigator of the LAMP prof-of-concept project which aims at delivering a side-channel resistant RISC-based System-on-Chip (see www.lamp-platform.org).

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.

William Fornaciari, MSc, PhD, IEEE senior member, is Associate Professor at Politecnico di Milano, Italy. He published 6 books and around 300 papers in int. journals and conferences, collecting 6 best paper awards and one certification of appreciation from IEEE. He holds three international patents on low power design. He has been coordinator of both FP7 and H2020 EU-projects. In 2016 he won the HiPEAC Technology Transfer Award. His research interests include embedded and cyber-physical systems, energy-aware design of sw and hw, run-time management of resources, High-Performance-Computing, design optimization and thermal management of multi-many cores.