FPnew: An Open-Source Multiformat Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing

Stefan Mach, Fabian Schuiki, Florian Zaruba, *Graduate Student Member, IEEE*, and Luca Benini, *Fellow, IEEE*

***Abstract*— The slowdown of Moore's law and the power wall necessitates a shift toward finely tunable precision (a.k.a. transprecision) computing to reduce energy footprint. Hence, we need circuits capable of performing floating-point operations on a wide range of precisions with high energy proportionality. We present FPnew, a highly configurable open-source transprecision floating-point unit (TP-FPU), capable of supporting a wide range of standard and custom FP formats. To demonstrate the flexibility and efficiency of FPnew in general-purpose processor architectures, we extend the RISC-V ISA with operations on half-precision, bfloat16, and an 8-bit FP format, as well as SIMD vectors and multiformat operations. Integrated into a 32-bit RISC-V core, our TP-FPU can speedup the execution of mixed-precision applications by 1.67× with respect to an FP32 baseline, while maintaining end-to-end precision and reducing system energy by 37%. We also integrate FPnew into a 64-bit RISC-V core, supporting five FP formats on scalars or 2, 4, or 8-way SIMD vectors. For this core, we measured the silicon manufactured in Globalfoundries 22FDX technology across a wide voltage range from 0.45 to 1.2 V. The unit achieves leading-edge measured energy efficiencies between 178 Gflop/sW (on FP64) and 2.95 Tflop/sW (on 8-bit mini-floats), and a performance between 3.2 and 25.3 Gflop/s.**

***Index Terms*— Energyefficient, floating-point unit, multiformat, RISC-V, transprecision computing.**

I. Introduction

THE last decade has seen explosive growth in the quest for energy-efficient architectures and systems. An era of exponentially improving computing efficiency, driven mostly by CMOS technology scaling, is coming to an end as Moore's law falters. The so-called thermal- or power-wall obstacle is fueling a push toward computing paradigms, which hold energy efficiency as the ultimate figure of merit for any hardware design.

Manuscript received July 2, 2020; revised November 3, 2020; accepted November 28, 2020. Date of publication December 30, 2020; date of current version April 1, 2021. This work was supported in part by the European Union's Horizon 2020 Research and Innovation Program through the Project "OPRECOMP" under Grant 732631 and in part by the European Union's H2020 EPI Project under Grant 826647. *(Corresponding author: Stefan Mach.)*

Stefan Mach, Fabian Schuiki, and Florian Zaruba are with the Integrated Systems Laboratory, ETH Zürich, 8092 Zürich, Switzerland (e-mail: smach@iis.ee.ethz.ch; fschuiki@iis.ee.ethz.ch; zarubaf@iis.ee.ethz.ch).

Luca Benini is with the Integrated Systems Laboratory, ETH Zürich, 8092 Zürich, Switzerland, and also with the Department of Electrical, Electronic, and Information Engineering Guglielmo Marconi, University of Bologna, 40136 Bologna, Italy (e-mail: lbenini@iis.ee.ethz.ch).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2020.3044752.

Digital Object Identifier 10.1109/TVLSI.2020.3044752

Simultaneously, rapidly evolving workloads, such as machine learning focuses on the computing industry and always demand higher compute performance at constant or decreasing power budgets, ranging from the data center, and high performance computing (HPC) scale down to the Internet of Things (IoT) domain. In this environment, achieving high energy efficiency in numerical computations requires architectures and circuits which are fine-tunable in precision and performance. Such circuits can minimize the energy cost per operation by adapting performance and precision to the application requirements in an agile way. The paradigm of "transprecision computing" [1] aims to create a holistic framework ranging from algorithms and software to hardware and circuits that offer many knobs to fine-tune workloads.

The most flexible and dynamic way of performing numerical computations on modern systems is floating-point (FP) arithmetic. Standardized in IEEE 754, it has become truly ubiquitous in most computing domains: from general-purpose processors, accelerators for graphics computations (GPUs) to supercomputers, but also increasingly in high-performance embedded systems and ultralow-power microcontrollers. While fixed-point computation, which usually uses integer datapaths, sometimes offers an efficient alternative to FP, it is not nearly as flexible and universal. Domain-specific knowledge by human experts is usually required to transform FP workloads into fixed point, as numerical range and precision tradeoffs must be managed and tracked manually. IEEE 754's built-in rounding modes, graceful underflow, and representations for infinity are there to make FP arithmetic more robust and tolerant to numerical errors [2]. Furthermore, many applications such as scientific computing with physical and chemical simulations are infeasible in fixed point and require high dynamic range, which FP offers.

FP precision modulation as required for efficient transprecision computing has been limited to the common "double" and "float" formats in CPUs and GPUs in the past. However, a veritable "Cambrian Explosion" of FP formats, e.g. Intel Nervana's Flexpoint [3], Microsoft Brainwave's 9-bit floats [4], the Google TPU's 16-bit "bfloats" [5], or NVIDIA's 19-bit TF32, implemented in dedicated accelerators such as Tensor Cores [6], shows that new architectures with extreme transprecision flexibility are needed for FP computation, strongly driven by machine learning algorithms and applications. Our goal is to create a flexible and customizable transprecision floating-point unit (TP-FPU) architecture that can be utilized across a wide variety of computing systems and applications.

To leverage such transprecision-enabled hardware, there must, of course, also be support and awareness across the entire software stack. An instruction set architecture (ISA) forms the interface between hardware and software. RISC-V [7] is an open-source ISA which natively supports computation on the common "double" and "float" formats. Furthermore, the ISA explicitly allows nonstandard extensions where architects are free to add their own instructions. Lately, RISC-V has gained traction in both industry and academia due to its open and extensible nature with growing support from hardware and software projects. In this work, we leverage the openness and extensibility of the RISC-V ISA by adding extensions for operations on additional FP formats not found in current RISC-V processor implementations [8].

In this work, we also demonstrate a fully functional silicon implementation of a complete open-source TP-FPU inside a RISC-V application-class core in a 22-nm process [9]. The taped-out architecture supports a wide range of data formats including IEEE 754 double (FP64), single (FP32), and half-precision floats (FP16), as well as 16-bit bfloats (FP16alt) and a custom 8-bit format (FP8), initially introduced in [10]. Furthermore, there is full support for single instruction multiple data (SIMD) vectorization, as well as vectorial conversions and data packing.

To summarize, our contributions are as follows.

1) The design of a highly configurable architecture for a transprecision floating-point unit written in SystemVerilog. All standard RISC-V operations are supported along with various additions such as SIMD vectors, multiformat fused multiply-add (FMA) operations, or convert-and-pack functionality to dynamically create packed vectors. The unit is fully open source and thus extensible to support even more functions. Unlike SOA designs, we increase the circuit area to achieve high energy proportionality and efficiency.
2) Extensions to the RISC-V ISA to support transprecision FP operations on FP64, FP32, FP16, FP16alt, and FP8 [10]. Programmers can leverage transprecision through standard operations in high-level programming languages, use compiler-enabled auto-vectorization, or make further optimization using compiler-intrinsic function calls to transprecision instructions [8].
3) Integration of the TP-FPU into RI5CY [11], a 32-bit embedded RISC-V processor core. An application case study shows that using transprecision ISA extension can achieve a 1.67× speedup to an FP32 baseline without sacrificing any precision in the result. Furthermore, the processor energy required to complete the workload is reduced by 37%.
4) Integration of the TP-FPU into Ariane [12], a 64-bit application-class RISC-V processor core, and subsequent silicon implementation in GLOBALFOUNDRIES 22FDX [9]. Energy and performance measurements of the manufactured silicon confirm the substantial energy proportionality and leading-edge energy efficiency of our architecture. We perform a detailed breakdown of per-instruction energy cost, vectorization gains, and an evaluation of the voltage/frequency scaling and body biasing impact on the manufactured silicon. Our design surpasses the SOA of published floating-point unit (FPU) designs in both flexibility and efficiency.

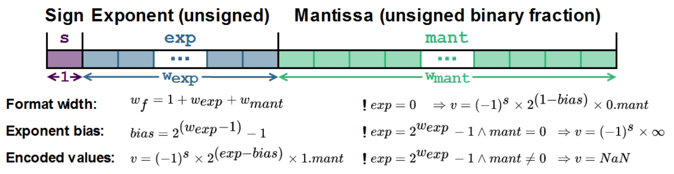

Sign
Exponent (unsigned)
Mantissa (unsigned binary fraction)
s
exp
mant
1
wexp
wmant
Format width:
Exponent bias:
Encoded values:
$w_f = 1 + w_{exp} + w_{mant}$
$bias = 2^{(w_{exp}-1)} - 1$
$v = (-1)^s \times 2^{(exp-bias)} \times 1.mant$
$! \; exp = 0 \Rightarrow v = (-1)^s \times 2^{(1-bias)} \times 0.mant$
$! \; exp = 2^{w_{exp}} - 1 \wedge mant = 0 \Rightarrow v = (-1)^s \times \infty$
$! \; exp = 2^{w_{exp}} - 1 \wedge mant \neq 0 \Rightarrow v = NaN$

Fig. 1. FP format encoding as specified by IEEE 754 and its interpretation.

The rest of this article is organized as follows. Section II describes in-depth the requirements and architecture of the proposed TP-FPU. Section III outlines our work on transprecision ISA extensions, the implementation of the hardware unit into the two processor cores, and the full implementation into silicon. Section IV contains a transprecision case study performed on the RI5CY core system as well as the silicon measurement results of the Ariane core system. The last sections of this article contrast our work with related works and provide a summary.

II. ARCHITECTURE

FPnew is a flexible, open-source hardware IP block that adheres to IEEE 754 standard principles, written in SystemVerilog. The aim is to provide FP capability to a wide range of possible systems, such as general-purpose processor cores as well as domain-specific accelerators.

A. Requirements

To address the needs of many possible target systems, applications, and technologies, FPnew had configurability as one of the driving factors during its development. The ease of integration with existing designs and the possibility of leveraging target-specific tool flows were also a guiding principle for the design. We present some key requirements that we considered during the design of the unit:

1) FP Format Encoding: As outlined in Section I, it is becoming increasingly attractive to add custom FP formats (often narrower than 32 bit) into a wide range of systems. While many of the systems mentioned earlier abandon standard compliance for custom formats in pursuit of optimizations in performance or circuit complexity, general-purpose processors are generally bound to adhere to the IEEE 754 standard. As such, the TP-FPU is designed to support any number of arbitrary FP formats (in terms of bit width) that all follow the principles for IEEE 754-2008 *binary* formats, as shown in Fig. 1.

2) Operations: To provide a complete FPU solution, we aim at providing the general operations mandated by IEEE 754, supporting arithmetic operations, comparisons, and conversions. Most notably, the FMA operation that was first included in a processor in 1990 [13] has since been added to IEEE 754-2008 and is nowadays ubiquitous in efficient AI and BLAS-type kernels. It computes $(a \times b) + c$ with only one final rounding step. We aim at natively supporting at least all FP operations specified in the RISC-V ISA.

Furthermore, for implementations supporting more than one FP format, conversions among all supported FP formats and integers are required. Nonstandard multiformat arithmetic is also becoming more common, such as performing the multiplication and accumulation in an FMA using two different formats in tensor accelerators [5], [6].

3) SIMD Vectors: Nowadays, most general-purpose computing platforms offer SIMD accelerator extensions, which pack several narrow operands into a wide datapath to increase throughput. While it is possible to construct such a vectorized wide datapath by duplicating entire narrow FPUs into vector lanes, operations would be limited to using the same narrow width. Flexible conversions amongst FP types are crucial for efficient on-the-fly precision adjustment in transprecision applications [14] and require support for vectored data. The architecture of the TP-FPU thus must be able to support this kind of vectorization to support multiformat operations on SIMD vectors.

4) Variable Pipeline Depths: To be performant and operate at high speeds, commonly used operations inside an FPU require pipelining. However, pipeline latency requirements for FP operations are very dependent on the system architecture and the choice of implementation technology. While a GPU, for example, will favor a minimum area implementation and is capable of hiding large latencies well through its architecture, the impact of operation latency can be far more noticeable in an embedded general-purpose processor core [14].

As such, the TP-FPU must not rely on hard-coding specific pipeline depths to support the broadest possible range of application scenarios. As circuit complexity differs significantly depending on the operation and FP format, the number of registers shall be configurable independently for each.

5) Design Tool Flow: To ensure accessibility and interoperability with existing design flows, the TP-FPU is written in the industry-standard SystemVerilog HDL (IEEE 1800), supported by all commonly used design flows. Novel hardware construction languages such as Chisel [15] were not considered as the lack of native tool support considerably adds complexity with respect tostandard design and verification flows.

Target-specific synthesis flows (e.g. for application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA) technologies) differ in available optimized blocks, favoring inferable operators over direct instantiation. Synthesis tools will pick optimal implementations for arithmetic primitives such as DSP slices in FPGAs or Wallace-Tree-based multipliers for ASICs with high timing pressure. As available optimizations also differ between targets, the unit is described in a way to enable automatic optimizations, including clock-gating and pipelining, wherever possible.

B. Building Blocks

In the following, we present a general architectural description of our TP-FPU, shown in Fig. 2. Concrete configurations chosen for the integration into processor cores and the implementation in silicon are discussed in Sections III and IV.

1) FPU Top Level: At the top level of the TP-FPU (see Fig. 2-1), up to three FP operands can enter the unit per clock cycle, along with control signals that determine the type of operation as well as the format(s) involved. One FP result leaves the unit along with the status flags raised by the current operation according to IEEE 754-2008. The width of the input and output operands is parametric and will be henceforth referred to as the *unit width* ($\mathtt{w}_{\text{fpu}}$).

Input operands are routed toward one of four operation group blocks, each dedicated to a class of instructions. Arbiters feed the operation group outputs toward the output of the unit. As only one operation group block can receive new data in any given clock cycle, clock and datapath gating can be employed to silence unused branches of the FPU, thereby eliminating spurious switching activity.

2) Operation Group Blocks: The four operation group blocks making up the TP-FPU are as follows.

1) *ADDMUL:* addition, multiplication and FMA.
2) *DIVSQRT:* division and square root.
3) *COMP:* comparisons and bit manipulations.
4) *CONV:* conversions among FP formats, to/from integers.

Each of these blocks forms an independent datapath for operations to flow through (see Fig. 2-2). When multiple FP formats are present in the unit, the blocks can host several slices that are either implemented as format-specific (parallel) or multiformat (merged). In the parallel case, each slice hosts a single FP format, giving the broadest flexibility in terms of path delay and latency, as each slice can contain its internal pipeline. While this duplication increases the circuit area, which is cheap in scaled ASIC technologies, it offers flexibility to implement each format efficiently. Inactive format slices can be clock-gated and silenced. In contrast, a merged slice can lower total area costs by sharing hardware and housing multiple formats at reduced flexibility. Furthermore, merging may incur energy and latency overheads due to small formats reusing the same over-dimensioned datapath, with the same pipeline depth for all formats.

3) Format Slices: Format slices host the functional units which perform the operations that the block is specialized in.

A SIMD vector datapath can be created if the format can be packed into the unit width ($\mathtt{w}_{\text{fpu}} \geq 2 \times \mathtt{w}_f$). In this case, slices will host multiple vector lanes, denoted `lane[1]...lane[k]`.

In the parallel case [see Fig. 2-3(a)], the lanes are duplicate instances of the same functional unit, and the number and width of lanes are determined as follows:

$$\mathtt{k}_{\text{parallel}} = \left\lfloor \frac{\mathtt{w}_{\text{fpu}}}{\mathtt{w}_f} \right\rfloor$$
$$\mathtt{w}_{\text{lane,parallel}} = \mathtt{w}_f.$$

In the merged case [see Fig. 2-3(b)], the total number of lanes is determined by the smallest supported format, and the width of each lane depends on the containing formats. Individual lanes within merged slices have the peculiar property of differing in bit width, and each lane needs support for a different set of formats [see Fig. 2-4(b)]

$$\mathtt{k}_{\text{merged}} = \left\lfloor \frac{\mathtt{w}_{\text{fpu}}}{\min\limits_{\forall \text{format} \in \text{slice}} \mathtt{w}_f} \right\rfloor$$
$$\mathtt{w}_{\text{lane}[i],\text{merged}} = \max_{\forall \text{format} \in \text{slice}} \mathtt{w}_f \Big|_{\mathtt{w}_f \leq \frac{\mathtt{w}_{\text{fpu}}}{i}}.$$

Depending on whether the current operation is scalar or vectored, either one or several lanes are used to compute the slice's result while unused lanes are silenced. The merged slices in the CONV block require a more complex data distribution and collection scheme for SIMD vectors as input and output format widths can differ. Furthermore, it is possible

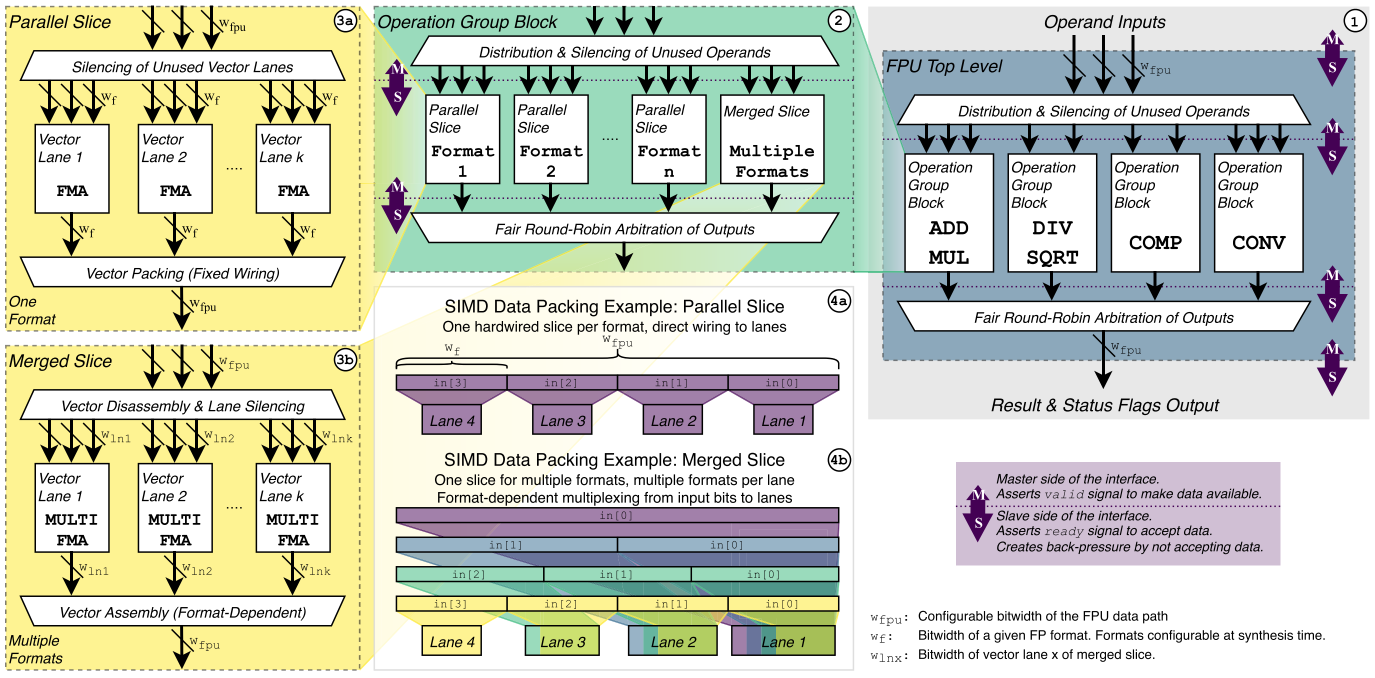

Parallel Slice
3a
Silencing of Unused Vector Lanes
Vector Lane 1 FMA
Vector Lane 2 FMA
Vector Lane k FMA
Vector Packing (Fixed Wiring)
One Format
Merged Slice
3b
Vector Disassembly & Lane Silencing
Vector Lane 1 MULTI FMA
Vector Lane 2 MULTI FMA
Vector Lane k MULTI FMA
Vector Assembly (Format-Dependent)
Multiple Formats
Operation Group Block
2
Distribution & Silencing of Unused Operands
Parallel Slice Format 1
Parallel Slice Format 2
Parallel Slice Format n
Merged Slice Multiple Formats
Fair Round-Robin Arbitration of Outputs
Operand Inputs
1
FPU Top Level
Distribution & Silencing of Unused Operands
Operation Group Block ADD MUL
Operation Group Block DIV SQRT
Operation Group Block COMP
Operation Group Block CONV
Fair Round-Robin Arbitration of Outputs
Result & Status Flags Output
SIMD Data Packing Example: Parallel Slice
4a
One hardwired slice per format, direct wiring to lanes
in[3] in[2] in[1] in[0]
Lane 4 Lane 3 Lane 2 Lane 1
SIMD Data Packing Example: Merged Slice
4b
One slice for multiple formats, multiple formats per lane
Format-dependent multiplexing from input bits to lanes
Master side of the interface. Asserts valid signal to make data available.
Slave side of the interface. Asserts ready signal to accept data. Creates back-pressure by not accepting data.
w_fpu: Configurable bitwidth of the FPU data path
w_f: Bitwidth of a given FP format. Formats configurable at synthesis time.
w_lnx: Bitwidth of vector lane x of merged slice.

Fig. 2. Datapath block diagram of the TP-FPU with its underlying levels of hierarchy. It supports multiple FP formats, and the datapath width is configurable.

to cast two scalar FP operands and insert them as elements of vectors for the dynamic creation of vectors at runtime. If SIMD is disabled, there is only one lane per slice.

4) Functional Units: The functional units within a slice can either be fully pipelined or use a blocking (e.g., iterative) implementation.

The ADDMUL block uses fully pipelined FMA units compliant with IEEE 754-2008, implemented using a single-path architecture [13], [16], providing results within $1/2$ ulp. A multiformat version of the FMA unit is used in merged slices, supporting mixed-precision operations that use multiple formats. Namely, the multiplication is done in the source format `src_fmt` while the addition is done using the destination format `dst_fmt`, matching the C-style function prototype *`dst_fmt fma(src_fmt, src_fmt, dst_fmt)`*.

In the DIVSQRT block, divisions and square roots are computed using an iterative nonrestoring divider. The iterative portion of the unit computes three mantissa bits of the result value per clock cycle and is implemented as a merged slice. The number of iterations performed can be overridden to be fewer than needed for the correctly rounded result to trade throughput for accuracy in transprecision computing. The operational unit in the COMP block consists of a comparator with additional selection and bit manipulation logic to perform comparisons, sign manipulation as well as FP classification. Lastly, the CONV block features multiformat casting units that can convert between any two FP formats, from integer to FP formats, and from FP to integer formats.

5) Unit Control Flow: While only one operation may enter the FPU per cycle, multiple values coming from paths with different latencies may arrive at the slice outputs in the same clock cycle. The resulting data from all slices and blocks are merged using fair round-robin arbitration. To stall internal pipelines, a simple synchronous *valid-ready* handshaking protocol is used within the internal hierarchies as well as on its outside interface of the unit.

As the unit makes heavy use of handshaking, data can traverse the FPU without the need for a priori knowledge of operation latencies. Fine-grained clock gating based on handshake signals can thus occur within individual pipeline stages, silencing unused parts and "popping" pipeline bubbles by allowing data to catch up to the stalled head of a pipeline. Coarse-grained clock gating can be used to disable operation groups or the entire TP-FPU if no valid data are present in the pipeline.

C. Configuration, Parametrization, and Usage

The TP-FPU offers ample configurability through a SystemVerilog package and instance parameters. In the package, any number of custom formats can be defined containing any number of exponent and mantissa bits, not limited to power-of-two format widths as in traditional computing systems.[1] Formats are treated according to IEEE 754-2008 (see Fig. 1) and support all standard rounding modes. Using parameters, operations on defined formats can be implemented as either a parallel or a merged slice or disabled completely. Furthermore, SIMD vectors can be enabled globally for all formats with $\mathtt{w}_f \leq \mathtt{w}_{\text{fpu}}/2$, and $\mathtt{w}_{\text{fpu}}$ can be chosen much wider than the largest supported format (e.g., for vector accelerators). Also parametric is the number of pipeline stages for each format and operation group, with all formats of merged slices sharing a pipeline. Retiming features of synthesis tools might be required to optimize these registers' placement for given target technologies.

[1]To meaningfully interpret bit patterns as FP values according to IEEE 754, a format should contain at least 2 bit each of exponent and mantissa. The SystemVerilog language does not guarantee support for signal widths above 2^{16} bit, which is far beyond the reasonable use case of a FP format.

All TP-FPU instance parameters are fixed into hardware during synthesis, and multiple different instances can be created from the same FPnew RTL without modifications. In fact, all implementations in this work were parametrized from the default package. As the unit is open source, designers are free to extract or replace functional units or add further operation groups of their own.

III. Integrating FPnew in RISC-V Cores

The TP-FPU has been integrated into several designs, and this work focusses on the implementation within RISC-V processor cores. To leverage transprecision computing on RISC-V platforms, we have extended the ISA with special instructions. We integrated the unit into RI5CY, a 32-bit low-power core, and Ariane, a 64-bit application-class processor.

A. ISA Extensions

The RISC-V ISA offers ample opportunities for extensions with custom operations. Therefore, we add nonstandard FP formats and instructions to enable transprecision computing and fully leverage our TP-FPU in general-purpose processors.

1) FP Formats: In addition to the IEEE 754 *binary32* and *binary64* formats included in RISC-V "F" and "D" standard extensions, respectively, we also offer smaller-than-32 bit formats proposed in [10]. The available FP formats in our implementations are:

1) *binary64 (FP64):* IEEE 754 double-precision (11, 52);
2) *binary32 (FP32):* IEEE 754 single-precision (8, 23);
3) *binary16 (FP16):* IEEE 754 half-precision (5, 10);
4) *binary16alt (FP16alt):* custom half-precision (8, 7)[2]
5) *binary8 (FP8):* custom quarter-precision minifloat (5, 2).

Data in all these formats are treated analogously to standard RISC-V FP formats, including the support for denormals, NaN and the NaN-boxing of narrow values inside wide FP registers.

2) Operations: The new operations can be roughly grouped into three parts, namely *scalar*, *vectorial*, and *auxiliary* extensions [8].

a) Scalar instructions: The scalar extensions map all the operations found in the "F" standard extension, such as arithmetic, comparisons, conversions, and data movement to the newly introduced formats. Conversions among all supported FP types were added to enable efficient runtime precision-scaling in transprecision applications.

b) Vectorial instructions: We add SIMD capabilities on all supported FP formats that are narrower than the FP register file size (FLEN in RISC-V parlance). Thus, in a core with support for up to FP32 (FLEN = 32), a packed vector of 2 × FP16 is possible. Our ISA extension includes vectorial versions of all scalar instructions. Furthermore, we add *vector-scalar* versions of these operations where the second operand is a scalar. The scalar operand is replicated to all elements of the input vector, allowing SIMD matrix product computations without the need for transposing one matrix in memory, for example.

Converting between two formats require special care for vectors as their length can differ. In a system with FLEN = 32, converting a vector of 2 × FP16 to FP8 yields only two elements (16 bit) of the 4-element (32 bit) destination. Conversely, converting a vector of 4 × FP8 to FP16 would produce a 64-bit result, which does not fit the register. Therefore, we provide separate instructions to use the lower or upper part of a vector for vectorial conversions, allowing for flexible precision scaling in transprecision applications.

c) Auxiliary instructions: Some nonstandard operations to address the needs of transprecision computing systems complete our ISA extensions. For example, we add an expanding FMA operation, which performs the sum on a larger FP format than the multiplication, mapping to multiprecision operations of the merged FMA slice of the TP-FPU architecture. Our *cast-and-pack* instructions convert two scalar operands using the vectorial conversion hardware and subsequently pack them into elements of the result vector.

3) Encoding: The encoding of these new instructions was implemented as RISC-V brown-field nonstandard ISA extensions.[3] As the scalar instructions only introduce new formats, the encoding of the standard RISC-V FP instructions is reused and adapted. We use a reserved format encoding to denote the FP16 format and reuse the encodings in the quad-precision standard extension "Q" to denote FP8, as we are not targeting any RISC-V processor capable of providing 128-bit FP operations. Operations on FP16alt are encoded as FP16 with a reserved rounding mode set in the instruction word. Vectorial extensions make use of the vast unused space in the integer operation opcode space, similar to the encoding of DSP extensions realized for the RI5CY core [11]. Auxiliary instructions are encoded either in unused FP or integer operation opcode space, depending on whether they operate on scalars or vectors.

4) Compiler Support: Programmers require high-level support for novel features in computer architectures to make efficient use of them. As such, the formats and operations mentioned above were added into the RISC-V GCC compiler toolchain to allow for native support of transprecision operations in user programs [8]. Custom formats can be used like the familiar native FP types in the C/C++ programming language, for example, the new `float8` C type denotes an FP8 variable.

B. RI5CY With Transprecision FPU

RI5CY is an open-source 32-bit, four stage, in-order RISC-V RV32IMFC processor.[4] This small core is focused on embedded and DSP applications, featuring several custom nonstandard RISC-V extensions for higher performance, code density, and energy efficiency [11]. With this core, we want to showcase nonstandard transprecision operations within a low-power MCU-class open-source RISC-V core, which has gained broad industry adoption.[5]

1) ISA Extension Support: RI5CY supports the RISC-V "F" standard ISA extension, which mandates the inclusion of 32 32-bit FP registers. The core offers the option to omit the FP

[2]This format has been popularized under the name *bfloat16*. Our implementation differs from *bfloat16*; insofar we always follow IEEE 754 principles regarding subnormal and infinity values, not a number (NaN), and support all rounding modes.

[3]iis-git.ee.ethz.ch/smach/smallFloat-spec/blob/v0.5/smallFloat_isa.pdf

[4]https://github.com/pulp-platform/riscv

[5]https://www.openhwgroup.org

TABLE I
CONFIGURATION OF THE TP-FPU AS IMPLEMENTED INTO THE RI5CY CORE. THE FPU WIDTH IS $w_{\text{fpu}} = 32$ bit

Format	**Implementation** (number of cycles, number of lanes)			
	ADDMUL	DIVSQRT	COMP	CONV
FP32	merged (1,1)	disabled (-,-)	parallel (1,1)	merged (1,2)
FP16	merged (1,2)	disabled (-,-)	parallel (1,2)	merged (1,2)
FP16alt	merged (1,2)	disabled (-,-)	parallel (1,2)	merged (1,2)

registers and host FP data within the general-purpose register file to conserve area and reduce data movement.[6]

We add support for operations on FP16 and FP16alt, including packed SIMD vectors. By reusing the general-purpose register file for FP values, we can leverage the SIMD shuffling functionality present in the integer datapath through the custom DSP extensions. Support for both cast-and-pack as well as expanding FMA is added to the core as well.

2) Core Modifications: To handle these new instructions, we extend the processor's decoder with the appropriate instruction encodings. RISC-V requires so-called NaN-boxing of narrow FP values where all unused higher order bits of a FP register must be set to logic high. We extend the core's load/store unit to allow for one-extending scalar narrow FP data by modifying the preexisting sign-extension circuitry. We do not enforce the checking of NaN-boxing in the operation units, however, to be able to treat SIMD data as scalars if needed. Other than replacing RI5CY's FP32 FPU with the TP-FPU, the changes to the core itself are not very substantial when compared to the infrastructure already in place for the "F" extension.

3) FPU Configuration: We enable support for the above formats without adding any extra pipeline stages, as shown in Table I. Low-power MCUs target relatively relaxed clock targets, such that FP operations can complete within a single cycle. As XLEN $= 32$ and FP operations use the general purpose register file, w_{fpu} is set to 32 bit.

The ADDMUL block is implemented as a merged multiformat slice to allow for multiformat operations among FP16[alt] and FP32. The DIVSQRT block has been disabled as we do not utilize it for our case study and to demonstrate the fine-grained configurability of the TP-FPU. The CONV block uses two 32-bit lanes in a merged slice to enable cast-and-pack operations from two FP32 operands.

C. Ariane With Transprecision FPU

Ariane is an open-source 64-bit, six stage, partially in-order RISC-V RV64GC processor.[7] It has full hardware support for running an operating system as well as private instruction and data caches. To speedup sequential code, it features a return address stack, a branch history table, and a branch target buffer [12]. We aim at bringing a full transprecision computing system to silicon with this core, with support for energy-proportional computation supporting many formats.

1) ISA Extension Support: Ariane supports the RISC-V "F" and "D" standard ISA extensions, which makes the FP register

[6]At the time of writing, this extension is being considered as an official RISC-V extension 'Zfinx,' but specification work is not completed.

[7]https://github.com/pulp-platform/ariane

TABLE II
CONFIGURATION OF THE TP-FPU AS IMPLEMENTED INTO THE ARIANE CORE. THE FPU WIDTH IS $w_{\text{fpu}} = 64$ bit

Format	**Implementation** (number of cycles, number of lanes)			
	ADDMUL	DIVSQRT	COMP	CONV
FP64	parallel (4,1)	merged (21,1*)	parallel (1,1)	merged (2,2*)
FP32	parallel (3,2)	merged (11,0)	parallel (1,2)	merged (2,0)
FP16	parallel (3,4)	merged (7,0)	parallel (1,4)	merged (2,2*)
FP16alt	parallel (3,4)	merged (6,0)	parallel (1,4)	merged (2,0)
FP8	parallel (2,8)	merged (4,0)	parallel (1,8)	merged (2,4)

* Merged lane with support for all formats of equal width and narrower.

file of the core 64-bit wide. We add support for operations on FP16, FP16alt, and FP8, as well as SIMD operations for all these formats, including FP32. While we support the flexible cast-and-pack operations, this version of the core is not equipped with expanding FMA operations.

2) Core Modifications: We replace the core's FPU with our design, extend the processor's decoder with the new operations and the load/store circuitry of the core to also allow for one-extending narrower FP data for proper NaN-boxing. These additional core control circuitry changes are not timing-critical, and their cost is negligible concerning the rest of the core resources.

3) FPU Configuration: We configure the TP-FPU to include the aforementioned formats and add format-specific pipeline depths as shown in Table II. The number of pipeline registers is set so that the processor core can achieve a clock frequency of roughly 1 GHz. w_{fpu} is set to the FP register file width of 64 bit, hence there are no SIMD vectors for the FP64 format.

We choose a parallel implementation of the ADDMUL block to vary the latency of operations on different formats, and not incur unnecessary energy and latency overheads for narrow FP formats. Latency is format-dependent for DIVSQRT due to the iterative nature of the divider hardware used and not available on SIMD data to conserve area. Three mantissa bits are produced every clock cycle in addition to a constant three cycles for pre- and post-processing. Divisions take between 4 (FP8) and 21 (FP64) cycles, which is acceptable due to the relative rarity of divide and square-root operations in performance-optimized code. Conversions are again implemented using a merged slice, where two lanes are 64 bit wide for cast-and-pack operations using two FP64 values. Additionally, there are two and four 16-bit and 8-bit lanes, respectively, to cover all possible conversions.

IV. IMPLEMENTATION RESULTS

A. PULPissimo: RI5CY With Transprecision FPU

To benchmark applications on the TP-enabled RI5CY core, we perform a full place and route implementation of a platform containing the core. This section presents the implementation results, while Section IV-C shows an application case study on the implemented design.

1) Implementation: We make use of PULPissimo[8] to implement a complete system. PULPissimo is a single-core SoC platform based on the RI5CY core, including 512 kB of memory as well as many standard peripherals such as UART,

[8]https://github.com/pulp-platform/pulpissimo

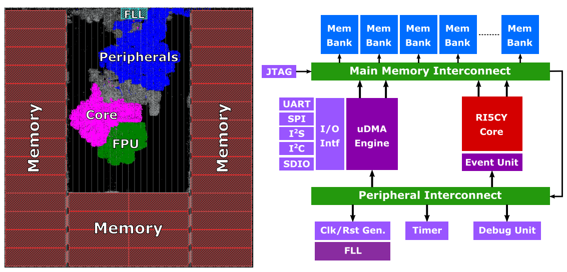


Fig. 3. Floorplan and block diagram of the PULPissimo SoC (without pad frame) using RI5CY Core with TP-FPU.

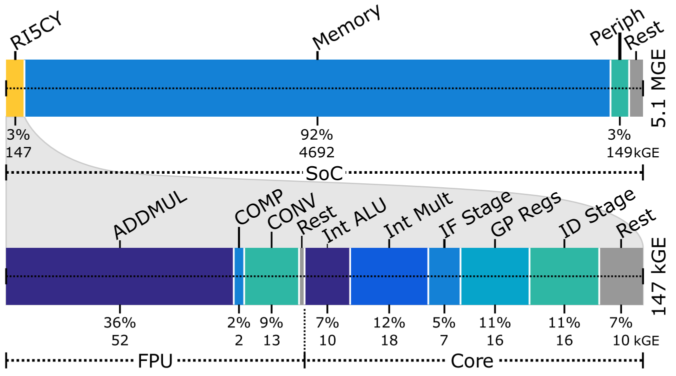


Fig. 4. Area distribution of the PULPissimo SoC and the RI5CY core (in kGE, 1 GE ≈ 0.199 μm^2).

I^{2}C, and SPI. We use our extended RI5CY core as described in Section III-B, including the single-cycle TP-FPU configuration shown in Table I.

The system has been fully synthesized, placed, and routed in GLOBALFOUNDRIES 22FDX technology, a 22-nm FD-SOI node, using a low-threshold 8-track cell library at low voltage. The resulting layout of the entire SoC (sans I/O pads) is shown in Fig. 3. Synthesis and place & route were performed using Synopsys Design Compiler and Cadence Innovus, respectively, using worst case low-voltage constraints (SSG, 0.59 V, 125 °C), targeting 150 MHz on the final design, with additional 20% of clock uncertainty in synthesis. Under nominal low-voltage conditions (TT, 0.65 V, 25 °C), the system runs at 370 MHz. The critical path of the design is between the memories and the core, involving the SoC interconnect.

2) Impact of the TP-FPU: The total area of the RI5CY core with TP-FPU is 147 kGE, of which the FPU occupies 69 kGE (47%), while the entire PULPissimo system including memories is 5.1 MGE, see Fig. 4. The ADDMUL block hosting the merged multiformat FMA units for all formats occupies 76% of the FPU area, while the COMP and CONV blocks use 4% and 18%, respectively.

Compared to a standard RI5CY core with support for only FP32, area increases by 29% and static energy by 37%. The higher increase in energy with respect to the added area stems from the FPU utilizing relatively more high-drive, short-gate cells than the rest of the processor: While the TP-FPU is not timing-critical, it uses 2.4× more 20 nm transistors over 28 nm ones compared the rest of the core, due to the long paths through the single-cycle unit. On the system scale, the added area and static energy account for only 0.7% and 0.9%, respectively, due to the impact of memories (92% and 96% of system area and leakage, respectively).

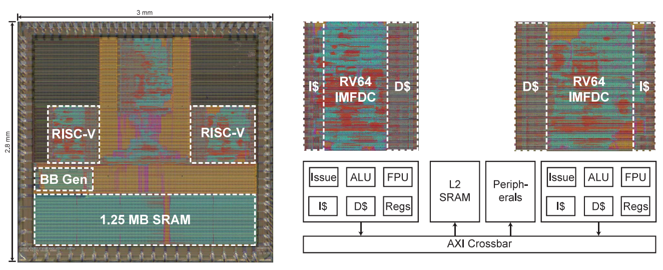


Fig. 5. Die micrograph and block diagram of entire Kosmodrom chip showing the placement of the two Ariane core macros with TP-FPU.

From an energy-per-operation point of view, it is interesting to compare FP32 FMA instructions with the 32-bit integer multiply-accumulate (MAC) instructions available in RI5CY. Under nominal low-voltage conditions at 370 MHz, these FP and integer instructions consume 3.9 pJ and 1.0 pJ in their respective execution units on average. Considering the system-level energy consumption, operating on FP32 data averages 22.2 pJ per cycle while the integer variant would require 21.2 pJ for running a filtering kernel (see Section IV-C), achieving equal performance. These small system-level differences in area and static and dynamic energy imply that FP computations are affordable even in an MCU context.

B. Kosmodrom: Ariane With Transprecision FPU

We implement a full test system with the TP-enabled Ariane core in silicon and perform a detailed analysis of the per-operation energy efficiency of FP instructions.

1) Silicon Implementation: We implement a full test system in GLOBALFOUNDRIES 22FDX technology called Kosmodrom [17]. Fig. 5 contains a silicon micrograph as well as an architectural overview of the main blocks in the design. Two functionally identical Ariane cores with TP-FPU have been fabricated using different cell technologies and target frequencies, and share a 1.25 MB L2 memory and common periphery like interrupt and debug infrastructure. We support five FP formats with dedicated datapaths for each one, leveraging the format-specific latencies shown in Table II.

Synthesized using Synopsys Design Compiler, the faster, higher performance core uses a low threshold eight-track cell-library while the slower, low-power core features a 7.5-track library. We will solely focus on the high-performance core for the subsequent performance and efficiency analysis as the cores can be individually clocked and powered. In synthesis, a 1-GHz worst case constraint (SSG, 0.72 V, 125 °C) with a 20% clock uncertainty was set. We use automated clock gate insertion extensively during synthesis (>96% of FPU registers are gated). Ungated registers comprise only the handshaking tokens and the finite-state machine controlling division and square root. The locations of pipeline registers in the entire FPU were optimized using the register retiming functionality of the synthesis tool.

Placement and routing are done in Cadence Innovus with a 1 GHz constraint in a multimode multicorner flow that includes all temperature and mask misalignment corners, eight in total. As part of corner trimming evaluations, we assume forward biasing voltage in the worst and typical corners

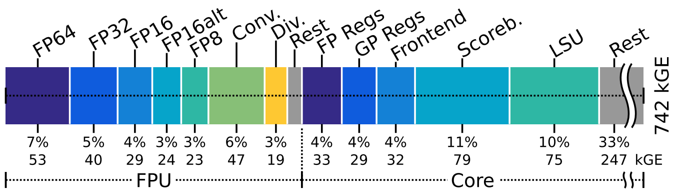


Fig. 6. Area distribution of the entire Ariane RISC-V core, excluding cache memories (in kGE, 1 GE ≈ 0.199 μm^2).

to relax pressure on critical paths to reduce leakage. The finalized backend design reaches 0.96 GHz under worst case conditions (SSG, 0.72 V, ±0.8 V bias, −40/125 °C), 1.29 GHz under nominal conditions (TT, 0.8 V, ±0.45 V bias, 25 °C), and 1.76 GHz assuming best case conditions (FFG, 0.88 V, 0 V bias, −40/125 °C).

We have also performed a substantial exploration of logic cell mixes (threshold voltage and transistor length) to maximize energy efficiency. The design contains 74% LVT and 26% SLVT cells; and 86% 28 nm, 8% 24 nm, and 6% 20-nm transistors.

2) Static Impact of the TP-FPU: The total area of the Ariane core with TP-FPU is 742 kGE (1.4 MGE including caches). The area breakdown is shown in Fig. 6. The total size of the FPU is 247 kGE, of which 160 kGE make up the various FMA units, 13 kGE are comparison and bit manipulation circuitry, 19 kGE for the iterative divider and square root unit, and 47 kGE are spent on the conversion units. Compared to a complete Ariane core (including caches) with support for only scalar FP32 and FP64 ("F" and "D" extensions), area and static energy are increased by 9.3% and 11.1%, respectively. The added area and energy cost in the processor are moderate, considering that FP operations on three new formats were added, along with SIMD support, which improves FP operation throughput by up to 8× when using FP8 vectors.

3) Silicon Measurements:

a) Evaluation methodology: We extract a detailed breakdown of energy consumption within the FPU by stressing individual operations using synthetic applications on Ariane. Each instruction is fed with randomly distributed normal FP values constrained such that the operations do not encounter overflow, creating a worst case scenario for power dissipation by providing high switching activity inside the datapath of the TP-FPU. Measurements are taken with full pipelines and the FPU operating at peak performance to provide a fair comparison.

Silicon measurements of the core and memory power consumption are done with the processor and FPU performing a matrix–matrix multiplication. Using a calibrated post-place-and-route simulation with full hierarchical visibility allows us to determine the relative energy cost contribution of individual hardware blocks. Post-layout power simulations are performed using typical corner libraries at nominal conditions (TT, VDD = 0.8 V, 25 °C). Silicon measurements are performed under unbiased nominal (0.8 V, 0 V bias, 25 °C) conditions where 923 MHz are reached, unless noted otherwise. The impact of voltage scaling on performance and energy efficiency is obtained through measurements of the manufactured silicon.

b) FPU instruction energy efficiency and performance: The top of Fig. 7 shows the average *per-instruction*[9] energy cost within the FPU for arithmetic scalar operations. The energy proportionality of smaller formats is especially pronounced in the ADDMUL block due to the high impact of the multiplier (first three groups of bars). For example, the FP64 FMA *fmadd.d* consumes 26.7 pJ, while performing the same operation on FP32 requires 65% less energy. Reducing the FP format width further costs 48%, 54%, and 49% of energy when compared to the next larger format for FP16, FP16alt, and FP8, respectively. Using FP16alt instead of FP16 consumes is energetically 12% cheaper due to the smaller mantissa multiplier needed for FP16. Similarly, reducing the FP format width leads to relative energy gains when compared to the next-larger format of 65%, 47%, 52%, 47% for FP multiplication, 53%, 47%, 57%, 47% for FP addition, and 38%, 34%, 35%, 22% for FP comparisons using FP32, FP16, FP16alt, and FP8, respectively. As such, scalar operations on smaller formats are energetically at least directly proportionally advantageous.

[9]One FPU instruction may perform multiple flops on multiple data items.

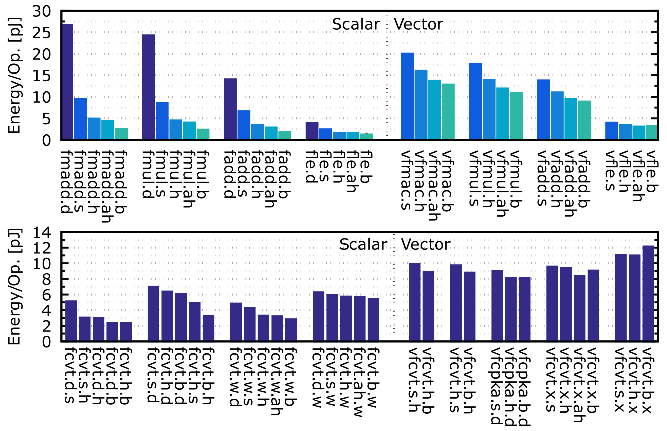


Fig. 7. FPU energy cost per instruction for the fully pipelined scalar operations (top left), vectorial operations (top right), grouped by FMA, multiply, add, and comparison; and scalar conversion operations (bottom left) and vectorial conversion operations (bottom right).

Intuition would suggest that SIMD instructions on all formats would require very similar amounts of energy due to the full utilization of the 64-bit datapath. However, we find that vectorial operations are also progressively energy-proportional, amplifying the energy savings even further. Starting from an energy cost of 20.0 pJ for an FP32 SIMD FMA *vfmac.s*, the *per-instruction* energy gains to the next-larger format for FP16, FP16alt, and FP8 are 20%, 31%, and 20%. Similarly, they are 21%, 32%, and 21% for multiplication, 20%, 31%, and 19% for addition, and 14%, 23%, and 8% for comparisons. Despite the full datapath utilization, packed operations using more narrow FP formats offer super-proportional energy gains while simultaneously increasing the throughput per instruction. This favorable scaling is owed to the separation in execution units for the individual formats where idle slices are clock-gated, which would be harder to attain using a conventional shared-datapath approach. By accounting for the increased throughput, the *per-datum* energy gains to the next larger format become 60%, 66%, and 58%, for the SIMD FMA, which is better than direct proportionality.

Conversion instructions that share a merged slice for all formats in the architecture's CONV block are an example of less pronounced energy scaling. The bottom of Fig. 7 shows the average *per-instruction* energy consumption of conversions on scalars and vectors. Energy consumption of instructions is influenced by both the source and destination formats in use.

For scalar FP-FP casts, converting to a larger format is energetically cheaper as only part of the input datapath toggles, and most of the output mantissa is padded with constant zeroes. Casts to a smaller format are more expensive as the wide input value causes dynamic switching within the conversion unit to produce the output. To contrast with the scaling results obtained above, we compare conversions where both the input and output formats are halved, such as `fcvt.s.d`, `fcvt.h.s`, and `fcvt.b.h`. Starting from 7.0 pJ for the FP64/FP32 cast, we find a reduction in energy of merely 30% and 35% when halving the format widths. Compared to the energy scaling results from above, the scaling is worse due to the use of one merged unit where unused portions of the datapath are much harder to turn off.

For SIMD vectors, the effect of *per-instruction* energy proportionality is visible again; going from the FP32/FP16 cast to the FP16/FP8 cast is 9.5% cheaper. While not as significant as for vectorial FMA, this gain is due to the additional vector lanes for casting small formats being narrower and supporting fewer formats.

The flexible cast-and-pack instructions allow the conversion of two FP64 values and pack them into two elements of the destination vector for only roughly 30% more energy than performing one scalar conversion from FP64 to the target format. It should be noted that two scalar casts and an additional packing operation, which are not directly available in the ISA, would be required without this functionality.

Measuring scalar FP-integer conversions where the integer width is fixed also show the relatively small relative gains, up to only 25% for FP16alt/int32 versus FP32/int32 conversions, much worse than direct proportionality. Vectorial FP-integer casts operate on integers of the same width as the FP format. Here, the impact of sharing vectorial lanes with other formats can make SIMD cast *instructions* on many narrow values cost more energy than on the larger formats, such as for the FP8/int8 cast, diminishing *per-datum* energy scaling when compared to the parallel slices.

Under nominal conditions, our TP-FPU thus achieves scalar FMA in 2.5 to 26.7 pJ, SIMD FMA in 1.6 to 10.0 pJ per data item, over our supported formats. FP-FP casts cost 7.0 to 26.7 pJ for scalar, and 2.2 to 4.9 pJ for vectorial data, respectively. Our approach of dividing the unit into parallel slices has proven to be effective at achieving high energy proportionality on scalar and SIMD data. At the silicon's measured nominal frequency of 923 MHz, this corresponds to a performance and energy efficiency of 1.85 Gflop/sW to 14.83 Gflop/sW and 75 Gflop/sW to 1245 Gflop/sW for the FMA across formats.

c) Impact of voltage scaling: Fig. 8 shows the impact of voltage and frequency scaling on the manufactured silicon. We measure the highest possible frequency and corresponding power consumption for supply voltages between 0.425 and 1.2 V. We observe peak compute and efficiency numbers of 3.17 Gflop/s and 178 Gflop/sW for FP64, 6.33 Gflop/s and 473 Gflop/sW for FP32, 12.67 Gflop/s and 1.18 Tflop/sW for FP16, 12.67 Gflop/s and 1.38 Tflop/sW for FP16alt, and 25.33 Gflop/s and 2.95 Tflop/sW for FP8.

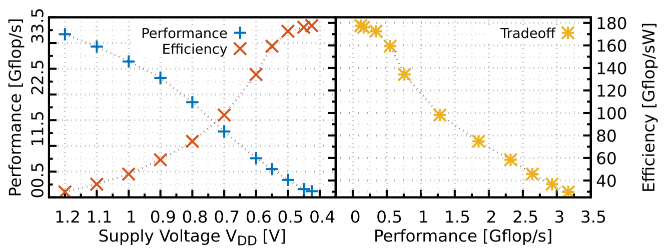


Fig. 8. Compute performance and energy efficiency of FP64 FMA versus supply voltage (left), tradeoff between compute performance and energy efficiency, achieved by adjusting supply voltage and operating frequency. Measured on manufactured silicon.

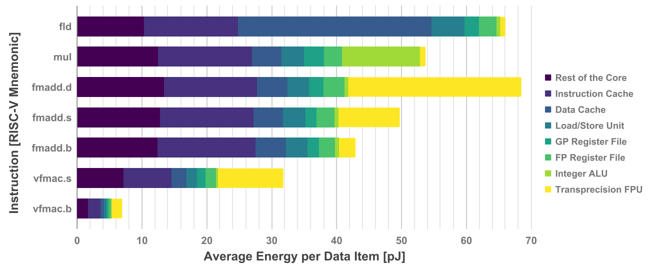


Fig. 9. Energy cost per data item of operations in the entire Ariane core.

d) Core-level energy efficiency: As the TP-FPU is merely one part of the entire processor system, we now briefly consider the energy spent during operations within the entire core. Fig. 9 shows the per-data energy consumption of the processor blocks performing various operations in the Ariane core. During an FP64 FMA—energetically the most expensive FP operation—the FPU accounts for 39% of the total Ariane core energy, with energy consumption of memory operations being comparable with that of the FP64 FMA. Although thanks to formidable energy proportionality, the FP8 FMA consumes 10.5× less FPU energy than the same operation on FP64; overall core energy consumption is decreased by only 38%. While for small and embedded applications, scalar FPU-level energy savings might be sufficient, they are not enough to bring maximum savings in energy efficiency through transprecision in application-class cores such as Ariane due to the relatively large core-side overheads.

Employing SIMD vectorization strongly mitigates this core overhead's impact on the energy cost per item. For example, the FP8 FMA requires another 6.2× less total core energy when executed as part of a vectorial FMA.

C. Performance and Programming of Transprecision Kernels

To visualize some challenges and benefits of transprecision applications, we showcase a multiformat application kernel running on the transprecision-enabled RI5CY core. Furthermore, we touch on the considerations to make when programming for transprecision-enabled platforms.

1) Transprecision Application Case Study: We consider the accumulation of element-wise products of two input streams, commonly found in many applications such as signal processing or SVM.

a)
```
float16 *a, *b;
float16 sum = 0;

for (int i = 0; i < n; ++i)
  sum += a[i] * b[i];
```

b)
```
float16 *a, *b;
float   sum = 0;

for (int i = 0; i < n; ++i)
  sum += a[i] * b[i];
```

c)
```
float16 *a, *b;
float   sum = 0;

for (int i = 0; i < n; ++i)
  __macex_f16(sum, a[i], b[i]);
```

Fig. 10. Accumulation of element-wise products from two input streams a and b. Inputs are in FP16, the result is accumulated using FP16 in (a), and using FP32 otherwise. Code c) uses compiler intrinsic functions to invoke transprecision instructions.

TABLE III
APPLICATION METRICS CORRESPONDING TO ASSEMBLY FROM FIG. 11

	# Bits	**Rel. Error of Result vs.**		**Rel. Energy**	
	correct	Exact	Exact FP16*	Core	System
Exact Result	37	0.0			
Cast to FP16	12	1.9×10^{-4}	0.0		
Result 11 **a)**	9	2.7×10^{-3}	2.9×10^{-3}	0.60	0.63
Result 11 **b)**	22	2.0×10^{-7}	0.0	1.00	1.00
Result 11 **c)**	19	1.6×10^{-6}	0.0	1.16	1.03
Result 11 **d)**	19	1.6×10^{-6}	0.0	0.97	0.75
Result 11 **e)**	22	2.0×10^{-7}	0.0	0.63	0.63

* The final result is converted to FP16 and compared to the exact result converted to FP16

a) Approach: Fig. 10 shows the C representation of the workload relevant for our evaluation. The input streams reside in memory as FP16 values, and the accumulation result uses FP16 or FP32. We use our transprecision ISA extensions to obtain the assembly in Fig. 11 as follows: Fig. 11(a) is the FP16-only workload in Fig. 10(a) requiring an ideal three instructions per input pair. Fig. 11(b) performs all operations on FP32 to achieve the most precise results but requires casts in a total of five instructions. Fig. 11(c) tries to save energy by performing the multiplication in FP16 to replace the FP32 FMA with additions. Fig. 11(d) accelerates the FP16 portion of the previous code using SIMD in 3.5 instructions. Fig. 11(e) makes use of expanding multiformat FMA instructions to combine computation and conversion in three instructions again.

The complete application repeats these actions over the entire input data using the zero-overhead hardware loops and post-incrementing load instructions available in RI5CY. Further manual loop unrolling can only be used to hide instruction latency overheads due to the low data intensity of this workload.

b) Performance and energy results: We collect the final result accuracy and energy use of these programs in Table III. Energy results have been obtained from a post-layout simulation of the RI5CY + TP-FPU design presented in Section IV-A.

The accuracy of the result from Fig. 11(a) is relatively low with 9 bit of precision correct (about three decimal digits), while the exact result of the operation would require 37 bit of precision. Due to the accumulation of rounding errors, the result strays far from the possibly most accurate representation of the exact result in FP16 (12 bit correct). The code in Fig. 11(b) offers 22 bit of precision but increases energy cost by 66% and 59% on core and system level, respectively, due to the increased execution time and higher per-instruction energy spent on FP32 operations. Fig. 11(c) suffers from decreased accuracy (19 bit) and even requires 16% more core energy (+3% system energy) with respect to the FP32 code, as the FP16 multiplications are energetically much more expensive than the casts they replace. Compared to the FP32 case, the use of SIMD in Fig. 11(d) reduces core energy by 3% and total system energy by even 25%. In the core, the increased performance slightly outweighs the increased FPU energy, where on the system level, the lower number of memory operations has a significant effect. Using the expanding multiply-accumulate operations in Fig. 11(e) offers the best of both worlds: the same performance as the naïve FP16-only version and the same precision as if performed entirely on FP32. Converted to FP16, this yields a value 14.6× more accurate than using FP16 only, reducing core and system power by 37% when compared to the FP32 case. These results highlight the energy savings potential of transprecision computing when paired with flexible hardware implementations.

2) Compiler Support: We make the low-level transprecision instructions available as a set of compiler intrinsic functions that allow full use of the transprecision hardware by the compiler and the programmer. Scalar types and basic operations are transparently handled by the compiler through the usage of the appropriate types (`float16`, `float16alt`, `float8`), and operators (+, *, etc.). Vectorial operations on custom FP formats are inferred using GCC vector extensions. In fact, the compiler can generate programs such as in Fig. 11(d) from the code in Fig. 10(b).

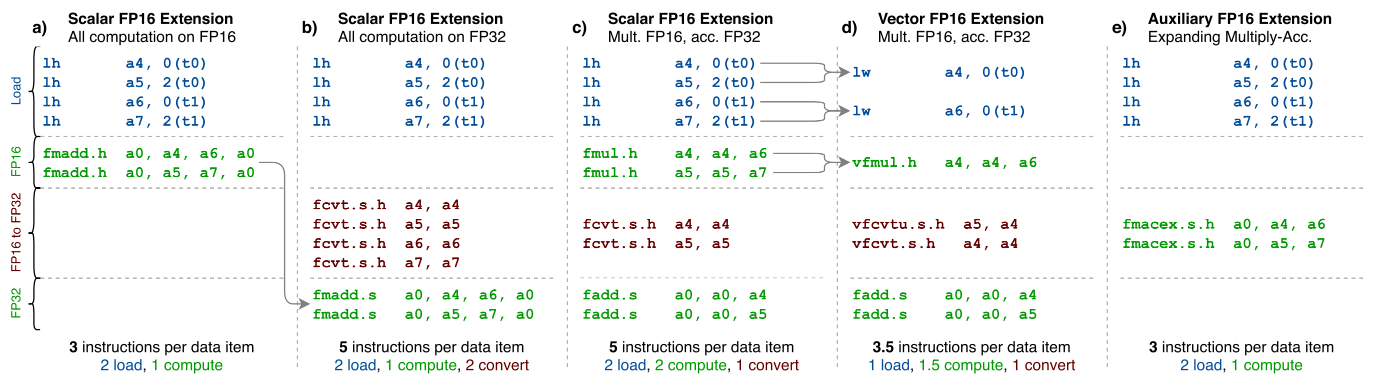
However, operations such as the FMA as well as optimized access and conversion patterns using SIMD vectors often do not cleanly map to high-level programming language operators and semantics. It is prevalent that performance-optimized FP code requires low-level manual tuning to make full use of the available hardware, even in nontransprecision code. We can and should make use of the noninferrable operations such as cast-and-pack or expanding FMA through calls to intrinsics, as seen in Fig. 10(c), which can produce assembly Fig. 11(e). The benefits and limits of the compiler-based approach are further investigated in [8].

V. RELATED WORK

A. SIMD and Transprecision in Commercial ISAs

Intel's x86-64 SSE/AVX extensions offer very wide SIMD operations (up to 512 bit in AVX-512) on FP32 and FP64. They include an FP dot-product instruction that operates on vectors of 2× FP64 or 4× FP32, respectively, producing a scalar result. Currently, no nonstandard FP formats are supported, but future CPUs with the AVX-512 extension (Cooper Lake) will include the *BF16* format (FP16alt) with support for cast-and-pack, as well as an expanding SIMD dot-product on value pairs.

The ARM NEON extension optionally supports FP16 and contains a separate register file for SIMD operations that support register fusion through different addressing views depending on the FP format used. The addressing mode is implicit in the use of formats within an instruction, enabling very consistent handling of multiformat (expanding, shrinking) operations that always operate on entire registers. This



a) **Scalar FP16 Extension** All computation on FP16

```
lh        a4, 0(t0)
lh        a5, 2(t0)
lh        a6, 0(t1)
lh        a7, 2(t1)
fmadd.h   a0, a4, a6, a0
fmadd.h   a0, a5, a7, a0
```

3 instructions per data item
2 load, 1 compute

b) **Scalar FP16 Extension** All computation on FP32

```
lh        a4, 0(t0)
lh        a5, 2(t0)
lh        a6, 0(t1)
lh        a7, 2(t1)
fcvt.s.h  a4, a4
fcvt.s.h  a5, a5
fcvt.s.h  a6, a6
fcvt.s.h  a7, a7
fmadd.s   a0, a4, a6, a0
fmadd.s   a0, a5, a7, a0
```

5 instructions per data item
2 load, 1 compute, 2 convert

c) **Scalar FP16 Extension** Mult. FP16, acc. FP32

```
lh        a4, 0(t0)
lh        a5, 2(t0)
lh        a6, 0(t1)
lh        a7, 2(t1)
fmul.h    a4, a4, a6
fmul.h    a5, a5, a7
fcvt.s.h  a4, a4
fcvt.s.h  a5, a5
fadd.s    a0, a0, a4
fadd.s    a0, a0, a5
```

5 instructions per data item
2 load, 2 compute, 1 convert

d) **Vector FP16 Extension** Mult. FP16, acc. FP32

```
lw          a4, 0(t0)
lw          a6, 0(t1)
vfmul.h     a4, a4, a6
vfcvtu.s.h  a5, a4
vfcvt.s.h   a4, a4
fadd.s      a0, a0, a4
fadd.s      a0, a0, a5
```

3.5 instructions per data item
1 load, 1.5 compute, 1 convert

e) **Auxiliary FP16 Extension** Expanding Multiply-Acc.

```
lh          a4, 0(t0)
lh          a5, 2(t0)
lh          a6, 0(t1)
lh          a7, 2(t1)
fmacex.s.h  a0, a4, a6
fmacex.s.h  a0, a5, a7
```

3 instructions per data item
2 load, 1 compute

Fig. 11. RISC-V assembly implementations of two iterations of the loop body in Fig. 10 (gray).

approach contrasts with our ISA extension, which requires multiple encodings to slice input or output vectors during vectorial conversions. A supplement to the ARM ISA is the scalable vector extension (SVE), targeting high-performance 64-bit architectures only, providing scaling to vector lengths far beyond 128 bit. SVE contains optional support for BF16 as a storage format, implicitly converting all BF16 input data to FP32 when used in computations, producing FP32 results. Converting FP data to BF16 for storage is also possible.

A new ISA extension for ARM M-class processors is called MVE. It reconfigures the FP register file to act as a bank of eight 128-bit vector registers, each divided into four "beats" of 32 bit. While vector instructions always operate on the entire vector register (fixed vector length of 128 bit), implementations are free to compute one, two, or all four beats per clock cycle—essentially allowing serializing execution on lower-end hardware. The floating-point variant of this ISA extension can operate on FP16 and FP32. Multiformat operations are not supported. An execution scheme in the spirit of MVE would apply to processors using our ISA extension with very little implementation overhead. For a single vector instruction, emitting a sequence of four or two SIMD FP operations recreates the behavior of a single-beat or dual-beat system for a FP register width of 32 and 64 bit, respectively. Furthermore, MVE also supports predication on individual vector lanes, interleaving, and scatter-gather operations not available in our extension.

There exists a working draft for the RISC-V "V" standard vector extension. The "V" extension adds a separate vector register file with Cray-style vector operation semantics and variable vector lengths. Multiple registers can also be fused to increase the vector length per instruction effectively. The standard vector extension includes widening and narrowing operations that fuse registers on one end of the operation, allowing consistent handling of data without the need for addressing individual register portions. It supports FP16, FP32, FP64, and FP128, as well as widening FMA operations. They operate in the same manner as our implementation of the `fmacex` operation, with the limitation that the target format must be exactly 2× as wide as the source. Furthermore, reduction operations for the inner sum of a vector exist.

B. Open-Source Configurable FPU Blocks

Most open-source FPU designs implement a fixed implementation in a specific format, targeting a specific system or technology[10], [11]; however, there are some notable configurable works available.

For example, FloPoCo [18] is a FP function generator targeted mainly at FPGAs implementations, producing individual functions as VHDL entities. FP formats are freely configurable in terms of exponent and mantissa widths; the resulting hardware blocks are not IEEE 754-compliant, however. Namely, infinity and NaN values are not encoded in the FP operands themselves, and subnormals are not supported as a tradeoff for a slightly higher dynamic range present in FloPoCo FP formats. The FMA operation is not available.

Hardfloat[12] on the other hand provides parametric FP functions that are IEEE 754 compliant. It is a collection of hardware modules written in Chisel with parametric FP format and includes the FMA operator. While Chisel is not widely adopted in commercial EDA tool flows, a generated standard Verilog version is also available. Hardfloat internally operates on a nonstandard recoded representation of FP values. However, the operations are carried out following IEEE 754, and conversion blocks are provided to the standard interchange encoding, which is used in the TP-FPU.

Both of these works offer individual function blocks instead of fully featured FPUs. However, thanks to the hierarchical architecture of the TP-FPU, it would be easily possible to replace its functional units with implementations from external libraries.

C. FPUs for RISC-V

Some vagueness exists in IEEE 754 concerning so-called *implementation-defined behavior*, leading to problems with portability and reproducibility of FP code across software and hardware platforms. FP behavior can be vastly different depending on both the processor model and compiler version used. To avoid at least the hardware-related issues, RISC-V specifies precisely how the open points of IEEE 754 are to be implemented, including the exact bit patterns of NaN results and when values are rounded. As such, FPUs intended for use in RISC-V processors (such as this work), are usually consistent in their behavior.

The FPUs used in the RISC-V cores originating from UCB, Rocket, and BOOM [19], [20], are based on Hardfloat components in specific configurations for RISC-V.

[10]https://opencores.org/projects/fpu
[11]https://opencores.org/projects/fpu100
[12]https://github.com/ucb-bar/berkeley-hardfloat/

Kaiser *et al.* [21] have published a RISC-V-specific implementation of the FMA operations on FP64 in the same GLOBALFOUNDRIES 22FDX technology as this work. We compare our TP-FPU to their implementation and others toward the end of this section.

D. Novel Arithmetics/Transprecision FP Accelerators

Nonstandard FP systems are becoming ever more popular in recent years, driven mainly by the requirements of dominant machine learning algorithms.

For example, both the Google TPU [5] and NVIDIA's Tensor Cores [6] provide high throughput of optimized operations on reduced-precision FP formats for fast neural network inference. Both offer a custom format termed *bfloat16* (BF16), using the same encoding as FP16alt in this work. The latter furthermore supports FP16 as well as a new 19-bit FP format called *TensorFloat32*, which is formed from the 19 most significant bits of an FP32 input value, producing results in FP32. However, both architectures omit certain features mandated in the standard, such as denormal numbers or faithful rounding, to pursue higher throughput and lower circuit area. The uplift in performance and efficiency on BF16 is directly actionable, as the format has been popularized for DL applications with negligible accuracy loss [22].

Dedicated accelerators geared toward neural network training such as NTX [23], for example, employ nonstandard multiply-accumulate circuits using fast internal fixed-point accumulation. While not compliant to IEEE 754, they can offer higher precision and dynamic range for accumulations.

FP-related number systems are also being employed, such as universal numbers (UNUMs) [24], Posits [25], or logarithmic number systems (LNSs). UNUM-based hardware implementations were proposed in [26] and [27]. Fast multiplication and transcendental functions were implemented into a RISC-V core in [28]. While the focus of our TP-FPU is to provide FPSTD-like FP capabilities, our work could be leveraged in several orthogonal ways to combine with these more exotic number systems. For example, dedicated functional units for these formats could be included in the TP-FPU as new operation groups alongside the current FP functions to accelerate specific workloads. Furthermore, our functional unit implementations can be utilized as a starting point to implement some of these novel arithmetic functions. The datapath necessary for posit arithmetic is very similar to a merged functional unit with a large number of possible input formats. Lastly, the TP-FPU could be used as an architectural blueprint and filled with arbitrary functional units, leveraging our architecture's energy proportionality.

E. Multimode Arithmetic Blocks

To our knowledge, no fully featured TP-FPUs with support for multiple formats have been published so far. However, multimode FMA architectures have been proposed recently, usually featuring computations on two or three FP formats [29]–[32], or a combination of integer and FP support [34]. Table IV compares the proposed architectures with our implementation under nominal conditions. It is noted that results for our work measure the entire TP-FPU energy while performing the FMA operation, not just the FMA block in isolation as in the related works.

TABLE IV
FMA COMPARISON OF THIS WORK (COMPLETE FPU) WITH OTHER STANDALONE [MULTIMODE] ARCHITECTURES AT NOMINAL CONDITIONS

Format		L/T[*]	Perf.[†] [Gflop/s]	Energy [pJ/flop]	Energy Efficiency [Gflop/s W]	rel.
This Work, 22 nm, 0.8 V[a], 0.049 mm^2 (entire FPU), 923 MHz						
FP64	scalar	4/1	1.85	13.36	74.83	1.0×
FP32	scalar	3/1	1.85	4.72	211.66	2.8×
FP16	scalar	3/1	1.85	2.48	403.08	5.4×
FP16alt	scalar	3/1	1.85	2.18	458.56	6.1×
FP8	scalar	3/1	1.85	1.27	786.30	10.5×
FP32	vector	3/2	3.71	5.01	199.70	2.7×
FP16	vector	3/4	7.42	2.01	497.67	6.7×
FP16alt	vector	3/4	7.42	1.72	581.96	7.8×
FP8	vector	2/8	14.83	0.80	1244.78	16.6×
Kaiser *et al.* [21], 22 nm, 0.8 V[c], 0.019 mm^2, 1.8 GHz						
FP64	scalar	3/1	3.60	26.40	37.88	
Manolopoulos *et al.* [29], 130 nm, 1.2 V[b], 0.287 mm^2, 291 MHz						
FP64	scalar	3/1	0.58	60.53	16.52	1.0×
FP32	vector	3/2	1.16	30.26	33.05	2.0×
Arunachalam *et al.* [30], 130 nm, 1.2 V[b], 0.149 mm^2, 308 MHz						
FP64	scalar	8/1	0.62	28.86	34.64	1.0×
FP32	vector	8/2	1.23	14.43	69.29	2.0×
Zhang *et al.* [31], 90 nm, 1 V[c], 0.181 mm^2, 667 MHz						
FP64	scalar	3/1	1.33	32.85	30.44	1.0×
FP32	vector	3/2	2.67	16.43	60.88	2.0×
FP16	vector	3/4	5.33	8.21	121.76	4.0×
Kaul *et al.* [32], 32 nm, 1.05 V[a], 0.045 mm^2, 1.45 GHz						
FP32	scalar	3/1	2.90	19.4	52.00	1.0×
FP20[‡]	vector	3/2	5.80	10.34	96.67	1.9×
FP14[‡]	vector	3/4	11.60	6.2	161.11	3.1×
Pu *et al.* [33], 28 nm, 0.8 V[a], 0.024[§]/0.018[|] mm^2, 910[§]/1360[|] MHz						
FP64	scalar	6/1	1.82	45.05	43.70	1.0×
FP32	scalar	6/1	2.72	18.38	110.00	2.5×

[*] Latency [cycle] / Throughput [operation/cycle] [†] 1 FMA = 2 flops
[a] Silicon measurements [b] Post-layout results [c] Post-synthesis results
[‡] FP20 = FP32 using only 12 bit of precision, FP14 = FP32 using only 6 bit of precision [§] FP64 FMA design [|] FP32 CMA design

The RISC-V-compatible FMA unit from [21] supports only FP64 with no support for FP32 even though required by RISC-V. Synthesized in the same 22 nm technology as our implementation, it achieves a 49% lower energy efficiency than our FPU performing the same FMA operation.

The architectures in [29]–[31] focus heavily on hardware sharing inside the FMA datapath, which forces all formats to use the same latency, no support for scalars in smaller formats, as well as lack of substantial energy proportionality. These architectures only achieve directly proportional energy cost, while our energy efficiency gains become subsequently better with smaller formats—reaching 16.6× lower energy for operations on FP8 with respect toFP64 (width reduction of 8×). By using the voltage scaling knob, this efficiency gain can again be increased by 2.3×, allowing for an over-proportional benefit to using the narrow FP formats in our implementation rather than a simple 2:1 tradeoff.

The FMA implementation in [32] uses a vectorization scheme where the FP32 mantissa datapath is divided by 2 or 4 employing very fine-grained gating techniques while keeping the exponent at a constant 8-bit width. This architecture's use is to attempt a bulk of FP32 computations at 4× throughput using the packed narrow datapath, costing 3.1× less energy. By tracking uncertainty, imprecise results are recomputed using the 2× reduced datapath before reverting the operation in full FP32. As such, the intermediate formats used in this unit do not correspond to any standard IEEE 754 formats.

FPMax [33] features separate implementations of the FMA operation for FP32 and FP64 without any datapath sharing, targeting high-speed ASICs. Comparing the energy cost of their two most efficient instances (using different internal architectures) yields energy proportionality slightly lower than our full FPU implementation. This result further compounds the value in offering separate datapaths for different formats on the scale of the entire FPU. It prompts us to explore the suitability of specific FMA architectures for different formats in the future.

F. Other Uses of Our TP-FPU

The open-source nature of FPnew, as well as the fact that it is written in synthesizable SystemVerilog, lower the burden of implementing FP functionality into new systems without the need for extra IP licenses or changes to standard design flow. FPnew or subcomponents of it have found use under the hood of some recent works.

Ara [35], a scalable RISC-V vector processor implementing a draft version of the “V” extension, makes use of FPnew instances to perform efficient matrix operations on 16 64-bit vector lanes. Snitch [36] is a tiny pseudo dual-issue RISC-V processor paired with a powerful double-precision FPU. It employs an instance of FPnew to provide FP64 and SIMD FP32 compute capabilities. Montagna *et al.* [37] make use of the flexibility of FPnew to explore different pipelining and sharing configurations in a multicore CPU cluster. GAP9,[13] a commercial IoT application processor announced by GreenWaves Technologies contains FPnew to enable sub-32-bit transprecision FP computation. Much of the FP hardware found in the European Processor Initiative (EPI) project [38] is based on the open-source FPnew design.

VI. Conclusion

We have presented FPnew, a configurable open-source transprecision floating-point unit capable of supporting arbitrary FP formats. It offers FP arithmetic and efficient casting and packing operations, in both scalar and SIMD-vectorized variants, with high energy efficiency and proportionality. We implemented the TP-FPU into an embedded RISC-V processor core to show the potential of transprecision computing, using our transprecision RISC-V ISA extension. In our case study, we achieve FP32 precision without incurring any performance overhead compared to an optimal scalar FP16 baseline, reducing system energy by 34% with respect tothe FP32 implementation. Furthermore, we implement the unit as part of a RISC-V application-class core into the first full TP-FPU silicon implementation with support for five FP formats, in GlobalFoundries 22FDX. Adaptive voltage and frequency scaling allows for energy efficiencies up to 2.95 Tflop/sW and compute performance up to 25.33 Gflop/s for 8 × FP8 SIMD operation. The cost in the additional area (9.3%) and static energy (11.1%) in the processor are tolerable in light of the significant gains in performance and efficiency possible with the TP-FPU.

Our design achieves better energy efficiency scaling than other multimode FMA designs thanks to the parallel datapaths approach taken in our architecture. Thanks to its open nature, FPnew can be utilized in many different application scenarios, having found use both in embedded IoT applications and high-performance vector processing accelerators.

[13]https://greenwaves-technologies.com/gap9iotapplicationprocessor

References

[1] A. C. I. Malossi *et al.*, “The transprecision computing paradigm: Concept, design, and applications,” in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1105–1110.

[2] W. Kahan, J. D. Darcy, E. Eng, and H.-P. N. Computing, “How Java’s floating-point hurts everyone everywhere,” in *Proc. ACM Workshop Java High-Performance Netw. Comput.*, Stanford, CA, USA: Stanford Univ., 1998, p. 81.

[3] U. Köster *et al.*, “Flexpoint: An adaptive numerical format for efficient training of deep neural networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1742–1752.

[4] E. Chung *et al.*, “Serving DNNs in real time at datacenter scale with project brainwave,” *IEEE Micro*, vol. 38, no. 2, pp. 8–20, Mar. 2018.

[5] N. P. Jouppi *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.

[6] S. Xie *et al.*, “Extreme datacenter specialization for planet-scale computing: ASIC clouds,” *ACM SIGOPS Operating Syst. Rev.*, vol. 52, no. 1, pp. 96–108, Aug. 2018.

[7] A. Waterman and K. Asanovic, Eds. *The RISC-V Instruction Set Manual: User-Level ISA, Document Version 20191213*, vol. 1. Cham, Switzerland: RISC-V Foundation, Dec. 2019.

[8] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, “Design and evaluation of SmallFloat SIMD extensions to the RISC-V ISA,” in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 654–657.

[9] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, “A 0.80pJ/flop, 1.24Tflop/sW 8-to-64 bit transprecision floating-point unit for a 64 bit RISC-V processor in 22nm FD-SOI,” in *Proc. IFIP/IEEE 27th Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2019, pp. 95–98.

[10] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benin, “A transprecision floating-point platform for ultra-low power computing,” in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1051–1056.

[11] M. Gautschi *et al.*, “Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.

[12] F. Zaruba and L. Benini, “The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

[13] R. K. Montoye, E. Hokenek, and S. L. Runyon, “Design of the IBM RISC System/6000 floating-point execution unit,” *IBM J. Res. Develop.*, vol. 34, no. 1, pp. 59–70, Jan. 1990.

[14] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini, “A transprecision floating-point architecture for energy-efficient embedded computing,” in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[15] J. Bachrach *et al.*, “Chisel: Constructing hardware in a scala embedded language,” in *Proc. 49th Annu. Design Autom. Conf. DAC*, 2012, pp. 1212–1221.

[16] J.-M. Müller *et al.*, *Handbook of Floating-Point Arithmetic*, 2nd ed. Boston, MA, USA: Birkhäuser, 2018.

[17] F. Zaruba, F. Schuiki, S. Mach, and L. Benini, “The floating point trinity: A multi-modal approach to extreme energy-efficiency and performance,” in *Proc. 26th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Nov. 2019, pp. 767–770.

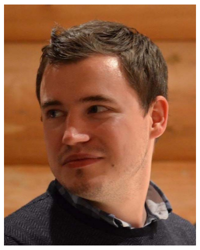
[18] F. de Dinechin and B. Pasca, “Designing custom arithmetic data paths with FloPoCo,” *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[19] K. Asanovic *et al.*, “The rocket chip generator,” EECS Dept., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, 2016.

[20] K. Asanovic, D. A. Patterson, and C. Celio, "The Berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized RISC-V processor," Univ. California at Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-167, 2015.
[21] F. Kaiser, S. Kosnac, and U. Brüning, "Development of a RISC-V-conform fused multiply-add floating-point unit," *Supercomputing Frontiers Innov.*, vol. 6, no. 2, pp. 64–74, 2019.
[22] P. Zamirai, J. Zhang, C. R. Aberger, and C. De Sa, "Revisiting BFloat16 training," 2020, *arXiv:2010.06192*. [Online]. Available: http://arxiv.org/abs/2010.06192
[23] F. Schuiki, M. Schaffner, and L. Benini, "NTX: An energy-efficient streaming accelerator for floating-point generalized reduction workloads in 22 nm FD-SOI," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 662–667.
[24] J. L. Gustafson, *The End of Error: Unum Computing*. Boca Raton, FL, USA: CRC Press, 2017.
[25] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers Innov.*, vol. 4, no. 2, pp. 71–86, 2017.
[26] F. Glaser, S. Mach, A. Rahimi, F. K. Gurkaynak, Q. Huang, and L. Benini, "An 826 MOPS, 210uW/MHz unum ALU in 65 nm," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.
[27] A. Bocco, Y. Durand, and F. de Dinechin, "Hardware support for UNUM floating point arithmetic," in *Proc. 13th Conf. Ph.D. Res. Microelectron. Electron. (PRIME)*, Jun. 2017, pp. 93–96.
[28] M. Gautschi, M. Schaffner, F. K. Gurkaynak, and L. Benini, "4.6 A 65nm CMOS 6.4-to-29.2pJ/FLOP@0.8 V shared logarithmic floating point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Jan. 2016, pp. 82–83.
[29] K. Manolopoulos, D. Reisis, and V. A. Chouliaras, "An efficient dual-mode floating-point multiply-add fused unit," in *Proc. 17th IEEE Int. Conf. Electron., Circuits Syst.*, Dec. 2010, pp. 5–8.
[30] V. Arunachalam, A. N. Joseph Raj, N. Hampannavar, and C. B. Bidul, "Efficient dual-precision floating-point fused-multiply-add architecture," *Microprocessors Microsyst.*, vol. 57, pp. 23–31, Mar. 2018.
[31] H. Zhang, D. Chen, and S.-B. Ko, "Efficient multiple-precision floating-point fused multiply-add with mixed-precision support," *IEEE Trans. Comput.*, vol. 68, no. 7, pp. 1035–1048, Jul. 2019.
[32] H. Kaul *et al.*, "A 1.45GHz 52-to-162GFLOPS/W variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm CMOS," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2012, pp. 182–184.
[33] J. Pu, S. Galal, X. Yang, O. Shacham, and M. Horowitz, "FPMax: A 106GFLOPS/W at 217GFLOPS/mm^2 single-precision FPU, and a 43.7GFLOPS/W at 74.6GFLOPS/mm^2 double-precision FPU, in 28nm UTBB FDSOI," 2016, *arXiv:1606.07852*. [Online]. Available: http://arxiv.org/abs/1606.07852
[34] T. M. Bruintjes, K. H. G. Walters, S. H. Gerez, B. Molenkamp, and G. J. M. Smit, "Sabrewing: A lightweight architecture for combined floating-point and integer arithmetic," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–22, Jan. 2012.
[35] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 2, pp. 530–543, Feb. 2020.
[36] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Trans. Comput.*, early access, Oct. 7, 2020, doi: 10.1109/TC.2020.3027900.
[37] F. Montagna *et al.*, "A transprecision floating-point cluster for efficient near-sensor data analytics," 2020, *arXiv:2008.12243*. [Online]. Available: http://arxiv.org/abs/2008.12243
[38] M. Kovač, D. Reinhardt, O. Jesorsky, M. Traub, J.-M. Denis, and P. Notton, "European processor initiative (EPI)—An approach for a future automotive eHPC semiconductor platform," in *Electronic Components and Systems for Automotive Applications*. Cham, Switzerland: Springer, 2019, pp. 185–195.

Stefan Mach received the B.Sc. and M.Sc. degrees from the Swiss Federal Institute of Technology Zürich (ETHZ), Zürich, Switzerland, where he is currently pursuing the Ph.D. degree.

Since 2017, he has been a Research Assistant with the Integrated Systems Laboratory, ETHZ. His research interests include transprecision computing, computer arithmetics, and energy-efficient processor architectures.

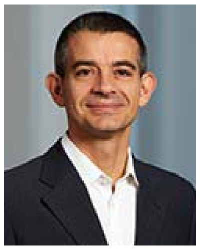


Fabian Schuiki received the B.Sc. and M.Sc. degrees in electrical engineering from the Swiss Federal Institute of Technology Zürich (ETHZ), Zürich, Switzerland, in 2014 and 2016, respectively, where he is currently pursuing the Ph.D. degree with the Digital Circuits and Systems Group, Luca Benini.

His research interests include transprecision computing as well as near- and in-memory processing.

Florian Zaruba (Graduate Student Member, IEEE) received the B.Sc. degree from TU Wien, Vienna, Austria, in 2014, and the M.Sc. degree from the Swiss Federal Institute of Technology Zürich, Zürich, Switzerland, in 2017, where he is currently pursuing the Ph.D. degree with the Integrated Systems Laboratory.

His research interests include the design of very large-scale integrated circuits and high-performance computer architectures.



Luca Benini (Fellow, IEEE) holds the Chair of digital circuits and systems with the Swiss Federal Institute of Technology Zürich (ETHZ), Zürich, Switzerland. He is a Full Professor with the Universita di Bologna, Bologna, Italy.

He has authored or coauthored more than 1000 peer-reviewed articles and five books. His research interests are in energy-efficient computing systems design, from embedded to high-performance.

Mr. Benini is a fellow of the ACM and a member of the Academia Europaea. He is the recipient of the 2016 IEEE CAS Mac Van Valkenburg Award and the 2020 EDAA Achievement Award.