Floating-Point Architectures for Energy-Efficient Transprecision Computing

Diss. ETH No. 27611

Floating-Point Architectures for Energy-Efficient Transprecision Computing

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
STEFAN MACH
MSc ETH EEIT
born September 26th, 1990
citizen of Luzern, Switzerland

accepted on the recommendation of
Prof. Dr. Luca Benini, examiner
Prof. Dr. Alberto Nannarelli, co-examiner

2021

Acknowledgments

When starting my Ph.D., I was not fully aware of what to expect from this unique and exciting journey and what lay ahead. Little did I know how much I would learn and be allowed to achieve during the time that followed. Now, some four and a half years later, I have arrived at the end of a genuinely formative experience that has helped me grow and helped me to learn so much, also about myself. As such a journey could never be undertaken alone, I am obliged to thank the many wonderful people that enabled, supported, and accompanied me through it all.

Firstly, I want to thank Luca for supervising and guiding me during my Ph.D. studies. Having access to your vast expertise and experience, which you use to support and guide – but also to question and challenge your protégés is hugely valuable. I highly appreciate the unequaled research and technical opportunities you enable for our entire institute. Furthermore, I am infinitely grateful for the patience you have afforded me during the more challenging parts of our shared journey. You were always an open, honest, and caring mentor, and I thank you for that. I would also like to thank Alberto for co-refereeing my thesis and contributing valuable insights and feedback.

Life as a Ph.D. student can be a crazy roller coaster ride with ups and downs where joy and struggle are often close together and rapidly changing. I want to thank my officemates Fabian, Florian, and Coco for an enjoyable working atmosphere and many unforgettable days (and night shifts) together. We have endured tape-outs and deadlines together and enjoyed many fruitful discussions, collaborations, as well as trips to the coffee machine. I hope you will remember our time

together as fondly as I will. Thank you also to Michael, who left me his desk at the window with an expertly organized collection of pens in its drawers.

I want to thank all my fellow "inmates" at the institute for our frequent chats, the coffee breaks, and the overall great atmosphere. I will fondly remember helping Daniele with his crazy drone projects. Thank you, Nils, for taking part in an unforgettable conference trip to Istanbul with me. But mainly, I would like to thank Gianna for being a valued friend and picking me up when I was down.

The great people at the Microelectronics Design Center and our IT group deserve praise for keeping our institute up and running. I especially thank Frank for his technical support and shielding us Ph.D. students by taking care of any administrative issues we or our projects might encounter. What you do is insanely valuable to all of us. Furthermore, I am convinced you are the best tour guide for cities that are Istanbul, and I believe you managed to spoil us for any future conference trips.

I would also like to thank everyone at the University of Bologna with whom I have had the pleasure of meeting and collaborating. Primarily, I want to thank Davide for his guidance and support, especially during the early stages of my Ph.D studies. Thank you very much, Davide, Beppe, and Antonio (whom I will include here by association), for the valuable collaborations, your excellent hospitality, and for teaching me the ways of the Guanciale.

I also want to thank all partners of the OPRECOMP project, which had run parallel to my entire time as a Ph.D. student. Thanks to all of you, we have managed to deliver the (allegedly) best-reviewed Horizon 2020 project ever.

Finally, I can hardly put into words my gratitude towards my friends and family, who have patiently supported me at all times. I am deeply grateful for my parents always encouraging and supporting me to pursue what I am fascinated by. I thank my sister Martina for always being there for me and understanding my struggles. Also, thank you, Jan, for putting up with me in our whacky shared flat. Ultimately, I thank Céline for her truly limitless love, support, understanding, and patience throughout all these years.

Abstract

An era of exponentially improving computing efficiency is coming to an end as Moore's law falters and Dennard scaling seems to have broken down. The power-wall obstacle fuels a push towards computing paradigms that put energy efficiency as the ultimate figure of merit for any hardware design. However, demand for more powerful computers is ever-growing, as rapidly evolving workloads such as ML always demand higher compute performance at constant or decreasing power budgets.

Floating-point (FP) has become truly ubiquitous in most computing domains, including general-purpose processors, GPUs, embedded systems, and ultra-low-power microcontrollers. While fixed-point arithmetic is a well-established paradigm in embedded systems optimization using a simplified numerical representation for real numbers, it is not necessarily the most energy-efficient solution and is unsuitable for many applications.

Significant advances in research have relaxed the "always maximum precision" approach to standard FP usage and exploit approximation more aggressively, as many algorithms are highly resilient to approximation. The emerging "transprecision computing" paradigm requires fine-grained control over numerical precision to tease out more performance and efficiency from hardware and software. Traditionally, however, only the standard "double" and "float" formats were available in CPUs and GPUs.

Recently, a slew of specialized FP formats and accelerators have been introduced, featuring custom reduced-width formats as a means of reducing hardware complexity and improving performance. A fur-

ther optimization enabled by reduced bit widths consists of leveraging SIMD execution to improve performance, energy efficiency, and memory footprint. Thus, there is a strong need for utmost flexibility and customizability in FP computations to generalize transprecision computing.

This thesis develops the concept of flexible transprecision computing for general-purpose systems by following a multi-pronged approach. It explores extensions to the FP type system, the addition of new types and operations to compilation toolchains, processor ISAs, and processor cores while also developing the necessary hardware for a fully configurable transprecision floating-point unit. These developments spawn a range of implementations spanning both the low-power embedded and application-class high-performance domains, delivering improvements in performance and energy efficiency across the board.

Zusammenfassung

Eine Ära exponentiell steigender Computereffizienz geht zu Ende, da das Mooresche Gesetz ins Wanken gerät und die Dennard-Skalierung zusammengebrochen zu sein scheint. Das Hindernis der Verlustleistungsgrenze treibt einen Schub in Richtung von Computer-Paradigmen, die Energieeffizienz als ultimative Kennzahl für jedes Hardwaredesign setzen. Die Nachfrage nach leistungsfähigeren Computern wächst jedoch ständig, da sich schnell entwickelnde Arbeitslasten wie ML immer höhere Rechenleistung bei konstantem oder sinkendem Energiebudget erfordern.

Gleitgkommazahlen sind in den meisten Bereichen der Datenverarbeitung allgegenwärtig, einschliesslich Mehrzweckprozessoren, GPUs, eingebetteten Systemen und Mikrocontrollern mit extrem niedrigem Stromverbrauch. Während die Festkommaarithmetik ein gut etabliertes Paradigma in der Optimierung eingebetteter Systeme ist, das eine vereinfachte numerische Darstellung für reelle Zahlen verwendet, ist sie nicht unbedingt die energieeffizienteste Lösung und für viele Anwendungen ungeeignet.

Bedeutende Fortschritte in der Forschung haben den “immer maximale Genauigkeit”-Ansatz bei der Nutzung von Gleitkommazahlen gelockert und setzen Näherungsberechnung aggressiver ein, da viele Algorithmen sehr widerstandsfähig gegen Näherung sind. Das aufkommende “Transprecision Computing”-Paradigma erfordert eine feinkörnige Kontrolle über die numerische Genauigkeit, um mehr Leistung und Effizienz aus Hardware und Software herauszuholen. Traditionell waren jedoch nur die Standardformate “double” und “float” in CPUs und GPUs verfügbar.

In jüngster Zeit wurden eine Reihe von spezialisierten Gleitkommaformaten und Beschleunigern eingeführt, die spezielle Formate mit reduzierter Bitbreite verwenden, um die Komplexität der Hardware zu reduzieren und die Leistung zu verbessern. Eine weitere Optimierung, die durch reduzierte Bitbreiten ermöglicht wird, besteht in der Nutzung der SIMD-Ausführung, um die Leistung, die Energieeffizienz und den Speicherbedarf zu verbessern. Es besteht also ein grosser Bedarf an grösstmöglicher Flexibilität und Anpassbarkeit bei Gleitkommaberechnungen, um Transprecision Computing zu verallgemeinern.

In dieser Arbeit wird das Konzept des flexiblen Transprecision Computing für Mehrzwecksysteme entwickelt, indem ein mehrgleisiger Ansatz verfolgt wird. Sie untersucht Erweiterungen des Gleitkommatypensystems, das Hinzufügen neuer Typen und Operationen zu Kompilierungstoolchains, Befehlssatzarchitekturen und Prozessorkernen und entwickelt gleichzeitig die notwendige Hardware für eine vollständig konfigurierbare Transprecision-Gleikommaeinheit. Diese Entwicklungen führen zu einer Reihe von Implementierungen, welche sowohl die Domäne der eingebetteten Systeme mit extrem niedrigem Stromverbrauch als auch die der Hochleistungsklasse abdecken und auf breiter Front Verbesserungen bei Leistung und Energieeffizienz liefern.

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays, computing advances all areas of science and engineering [1]. However, an era of exponentially improving computing efficiency, driven mainly through CMOS technology scaling, is coming to an end as Moore's law falters, and Dennard scaling seems to have broken down [2, 3]. The so-called thermal- or power-wall obstacle fuels a push towards computing paradigms that put energy efficiency as the ultimate figure of merit for any hardware design. This trend can already be observed in the industry [4, 5]. Higher energy efficiency will overcome the limits of Thermal Design Power (TDP), allowing further integration of more computational power into the same chip. Furthermore, as the amount of heat dissipated is reduced, less power for cooling will need to be expended, reducing one of the dominating cost factors in data centers [6].

At the same time, the demand for more powerful computers is ever-growing, as the High Performance Computing (HPC) community is currently targeting exascale machines (1 Eflop/s) [7]. Rapidly evolving workloads such as machine learning (ML) are the focus of the computing industry and always demand higher compute performance at constant or decreasing power budgets, ranging from the data-center and HPC scale down to the Internet of Things (IoT) domain. For

domains such as big-data, ML and scientific computing, the dominant HPC workloads are IEEE 754 floating-point (FP) operations. Emerging trends, like ML, exhibit a very regular and dense compute pattern that can be efficiently accelerated. Commercial vendors are building specialized hardware accelerators which are solely dedicated to accelerating the regular parts of these payloads in data centers [8, 9].

This model's main drawback is a high specialization to the underlying problem, nowadays usually ML, which makes the accelerator unsuitable for other tasks. In this environment, achieving high energy efficiency in general numerical computations requires architectures and circuits that are fine-tunable in precision and performance.

1.2 Transprecision Computing

The most flexible and dynamic way of performing numerical computations on modern systems is 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 HPC supercomputers, but also increasingly in high-performance embedded systems and ultra-low-power microcontrollers. 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 [10]. Furthermore, many applications such as scientific computing with physical and chemical simulations require the dynamic range which FP offers and are often infeasible using other approaches.

Floating-Point vs. Fixed-Point Computing Efficiency

While fixed-point computation, which usually uses integer data paths, sometimes offers an efficient alternative to FP, it is not nearly as flexible and universal. However, fixed-point arithmetic is a well-established paradigm in embedded systems optimization since it allows a simplified numerical representation for real numbers at high energy efficiency [11]. Nevertheless, many applications require high precision results characterized by a wide dynamic range (e.g., the accumulation stage of support vectors or feed-forward inference for deep neural

networks). In these cases, fixed-point implementations may suffer from numerical instability, requiring an in-depth analysis to make the result reliable. This methodology implies additional code sections to normalize and adjust the dynamic range avoiding saturation (e.g., the fixed-point implementation of linear time-invariant digital filters described in [12]). As a result, fixed-point arithmetic is not necessarily the most energy-efficient solution since the code requires real-time adaptations of the dynamic range that affect performance significantly and increase the time-to-market [11].

The adoption of single-precision FP arithmetic is a well-established paradigm to cope with these issues for embedded low-power systems. For example, such as ARM Cortex M4, a microcontroller unit (MCU) architecture that is the *de facto* standard for FP-capable low-power edge nodes. Combining FP and fixed-point arithmetic, depending on the computational requirements, is the typical approach for optimizing Cortex M4 applications. This approach's main shortcomings are the manual analysis for the format selection (float vs. fixed), the tuning required for adjusting the fixed-point dynamic range, and the software overhead to make the format conversions.

Furthermore, the usage of mixed (i.e., floating-fixed point) representations introduces several architectural bottlenecks in managing the pipelines and the register file, such as flushing or stalls that reduce these approaches' computational efficiency. Finally, at least in commercial architectures, the floating-point unit (FPU) cannot be turned off while the core executes fixed-point operations, resulting in a further reduction of energy efficiency.

The Push for Reduced Precision Arithmetic

In recent years, significant advances in research have been made to exploit approximation even more aggressively, aiming at relaxing the "always maximum precision" abstraction [13, 14], mainly thanks to the ever-growing interest in ML algorithms which are highly resilient to approximation. For example, Temporal Convolutional Networks (TCN) [15], Deep Learning (DL) algorithms [16], Convolutional Neural Networks (CNN) [17], Quantized Neural Networks (QNNs) [18] all tolerate lower precision arithmetic without losing their accuracy[19].

In this scenario, *transprecision (TP) computing* [20] is emerging

as a successful paradigm for embedded computing systems. This paradigm is an evolution of approximate computing, and it aims at tuning approximation at a fine grain during the computation progress through hardware and software control mechanisms. In the context of FP computations, this approach requires the availability of hardware units providing efficient support for multiple FP formats.

FP precision modulation as required for efficient TP computing has been limited to the standard "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 [21], Microsoft Brainwave's 9-bit floats [22], the Google TPU's 16-bit "bfloats" [23], or NVIDIA's 19-bit TF32, implemented in dedicated accelerators such as Tensor Cores [24], shows that new architectures with extreme TP flexibility are needed for FP computation, strongly driven by machine learning algorithms and applications.

A significant optimization enabled by FP bitwidth reduction relies on applying the single instruction multiple data (SIMD) approach on multiple sub-word elements simultaneously and is a common feature of such architectures [25]. These data types are known as *packed-SIMD vectors*. As a clear benefit, SIMD operations act on multiple data elements of the same size and type simultaneously, offering a theoretical $n\times$ speed-up for n SIMD lanes. Moreover, bitwidth reduction enables an equivalent reduction of the memory footprint, allowing to store larger problems in the same memory amount. This approach also enables more effective data movements, as multiple elements can be concurrently transferred when packed into machine words.

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, with a strong focus on enabling energy-proportional TP computing.

1.3 Open-Source Instruction-Set Architecture

In order to leverage such TP-enabled hardware, there must, of course, also be support and awareness across the entire software stack. The

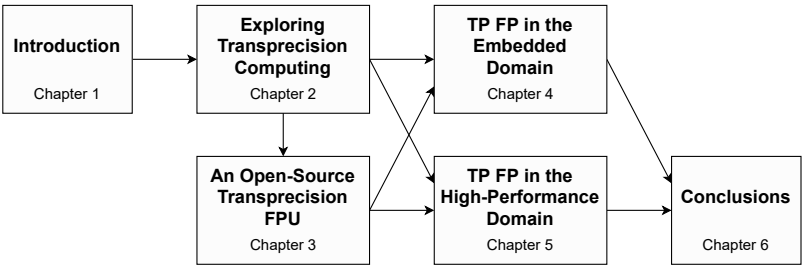


Figure 1.1: Structure and dependencies of chapters within this thesis.

interface between hardware and software is formed by the instruction set architecture (ISA).

RISC-V [26] is an open-source ISA which natively supports computation on the common "double" and "float" formats. Furthermore, the ISA explicitly allows non-standard extensions where architects are free to add new functionality of their own. Lately, RISC-V has gained traction in 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.

Paired with an open-source ISA, many projects now deliver open-source hardware [27, 28, 29, 30]. Open-source hardware offers the advantage of full access to architectures and implementation, which can be extended, adapted, and verified by many independent contributors. Frequently, licensing is permissive and allows the creation of new projects and products on top of high-quality, open systems backed by a large community [31].

We use open-source RISC-V processor cores to embed and demonstrate our work in real, complete computing systems. In turn, we also choose to make our contributions available open-source [32] to enable their inclusion in many different systems in the future, helping with the dissemination of the transprecision computing paradigm.

1.4 Outline

This section provides a brief outline of this thesis. A significant portion of the material presented herein has been previously published in journals and conference proceedings. Figure 1.1 shows the structure of this thesis and the dependencies between chapters.

Chapter 2 First, we investigate how the typical approach to numerical computations using FP arithmetic can be improved in terms of performance and energy efficiency. We focus on expanding the traditional FP type system with new sub-32-bit types (SmallFloat) to leverage the TP computing paradigm in general-purpose computing. We develop an FP emulation software library to assist with the precision tuning of applications and leverage it to evaluate the viability of TP computing on various applications. We contrast our choice of FP arithmetic for TP computing by exploring alternative number representations and their implications on hardware implementation.

Chapter 3 In a second step, we focus on supporting SmallFloat and energy-proportional FP computing in hardware, enabling the TP computing concept in real-world systems. We design FPnew, an open-source transprecision floating-point unit (TP-FPU) which provides energy-proportional hardware support for any custom FP format and is suited for a wide variety of target systems. To enable integration into general-purpose processor cores, we furthermore develop an extension to the open-source RISC-V ISA which supports our sub-32-bit types and implement it into the GCC RISC-V compiler suite.

Chapter 4 Focussing on embedded platforms, we introduce TP FP into an open-source RISC-V processor in a single-core system on a chip (SoC). We benchmark applications to confirm the performance and energy efficiency gains achievable with the introduction of TP on the processor level. Then, we scale up to introduce TP FP in an ultra-low-power embedded multi-core processor, exploring different architectures by sharing FPUs among multiple cores.

Chapter 5 Armed with an energy-proportional hardware TP-FPU, an ISA extension, and compiler integration, we also set our sights on implementing the unit into high-performance processor systems. We implement FPnew into an application-class RISC-V processor and validate the system on manufactured silicon. Lastly, we show how field-programmable gate arrays (FPGAs) could be employed to scale out TP computing into data centers and explore some of the many systems that have made good use of our TP-FPU.

Chapter 6 The final chapter summarizes the findings presented in the thesis, draws conclusions, and provides an outlook on future work.

1.5 Contributions

The key contributions of this thesis and related publications are summarized below. Refer to the respective chapter or section for a detailed introduction and treatment of the subject.

1. Evaluation and extension of the FP type system in the context of transprecision computing. Includes the development of a software library that enables explorations of FP formats and an energy-efficient prototype hardware design with support for multiple FP types. (Chapter 2, [33])

2. The design of *FPnew*, a highly configurable architecture for a transprecision floating-point unit. All standard RISC-V operations are supported along with various additions. The unit is fully open-source and thus extensible to support even more functions. (Chapter 3, [34])

3. Extensions to the RISC-V ISA to support TP FP operations on existing and new FP formats. Includes extensions to the standard RISC-V GCC compiler. (Chapter 3, [35])

4. A full architecture for ultra-low-power TP computing, based on PULPino SoC with a prototype TP-FPU integrated into the RI5CY processor core. (Chapter 4, [36])

5. Integration of FPnew into RI5CY RISC-V core and the PULPissimo SoC. Based on that, the architectural design of an embedded multi-core TP cluster and its software infrastructure. Includes architectural design space explorations considering: the number of cores, the number of shared FPUs, and the number of pipeline stages in the FPUs. (Chapter 4, [34, 37])

6. Integration of FPnew into the Ariane RISC-V application-class processor, introducing the first full TP-FPU silicon implementation in 22 nm. (Chapter 5, [34, 38])

7. Extension of the embedded TP cluster with the modifications necessary to be deployed as co-processors within large-scale data centers in conjunction with IBM POWER8™ host machines. (Chapter 5, [39])

1.6 List of Publications

Most of the material covered in this thesis has been published in the following conference and journal papers, and has been adapted for use in Chapters 2 to 5:

[33] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1051–1056

[34] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2020

[35] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "Design and evaluation of smallFloat SIMD extensions to the RISC-V ISA," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 654–657

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

[**37**] F. Montagna, S. Mach, S. Benatti, A. Garofalo, G. Ottavi, L. Benini, D. Rossi, and G. Tagliavini, "A transprecision floating-point cluster for efficient near-sensor data analytics," *arXiv preprint arXiv:2008.12243*, 2020

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

[**39**] D. Diamantopoulos, F. Scheidegger, S. Mach, F. Schuiki, G. Haugou, M. Schaffner, F. K. Gürkaynak, C. Hagleitner, A. C. I. Malossi, and L. Benini, "Xwattpilot: A full-stack cloud system enabling agile development of transprecision software for low-power SoCs," in *2020 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*. IEEE, 2020, pp. 1–3

The following publications with contributions by the author provide additional evidence and insights, and are covered in part by this thesis:

[**40**] F. Glaser, S. Mach, A. Rahimi, F. K. Gürkaynak, Q. Huang, and L. Benini, "An 826 Mops, 210 uW/MHz Unum ALU in 65 nm," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5

[**41**] D. Rossi, F. Conti, M. Eggiman, S. Mach, A. Di Mauro, M. Guermandi, G. Tagliavini, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "A 1.3 Tops/W @ 32 Gops fully integrated 10-core SoC for IoT end-nodes with 1.7 uW cognitive wake-up from MRAM-based state-retentive sleep mode," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. IEEE, Feb. 2021

[**42**] F. Schuiki, F. Zaruba, S. Mach, and L. Benini, "Kosmodrom: Energy-efficient ariane cores with transprecision FPU in 22nm," in *RISC-V Workshop Zurich*, 2019

[**43**] F. Zaruba, F. Schuiki, S. Mach, and L. Benini, "The floating point trinity: A multi-modal approach to extreme energyefficiency and performance," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2019, pp. 767–770

Further publications by the author which are not covered explicitly in this thesis:

[**44**] A. Pullini, M. Gautschi, F. K. Gürkaynak, F. Glaser, S. Mach, G. Rovere, D. Schiavone, G. Haugou, D. Palossi, A. Marongiu *et al.*, "KISS PULPino – updates on PULPino," in *5th RISC-V Workshop*. ETH Zürich, 2016

[**45**] A. Pullini, S. Mach, M. Magno, and L. Benini, "A dual processor energy-efficient platform with multi-core accelerator for smart sensing," in *International Conference on Sensor Systems and Software*. Springer, 2016, pp. 29–40

[**46**] M. Eggimann, S. Mach, M. Magno, and L. Benini, "A RISC-V based open hardware platform for always-on wearable smart sensing," in *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*. IEEE, 2019, pp. 169–174

[**47**] A. Di Mauro, F. Zaruba, F. Schuiki, S. Mach, and L. Benini, "Live demonstration: Exploiting body-biasing for static corner trimming and maximum energy efficiency operation in 22nm FDX technology," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–1

[**48**] H. Müller, D. Palossi, S. Mach, F. Conti, and L. Benini, "Fünfliber-drone: A modular open-platform 18-grams autonomous nano-drone," in *Design, Automation Test in Europe Conference Exhibition (DATE 2021)*, 2021, pp. 12–11

Chapter 2

Exploring Transprecision Computing

2.1 Introduction

In this starting chapter, we investigate how the classic approach to numerical computations using floating-point (FP) arithmetic can be improved upon to achieve the energy-proportional computation required for flexible TP computing. Focussing on performance and energy efficiency, we first perform a case study on embedded systems applications by expanding upon the commonly used FP type system. To contrast, we furthermore explore some alternative number systems and their implications on hardware design.

Nowadays most embedded applications involving numerical computations with large dynamic range are performed using *binary64* (double-precision) or *binary32* (single-precision) FP formats, described by the IEEE 754 standard [49]. In these applications, the execution of FP operations emerges as a significant contributor to energy consumption. To provide experimental evidence of this insight, we have executed a set of FP-intensive applications on PULPino [50], an open-

source ultra-low power (ULP) microcontroller. Results show that 30% of the energy consumption of the core is actually due to FP operations. Moreover, an additional 20% is spent moving FP operands from data memory to registers and vice versa.

To provide a compromise between energy cost and dynamic range, IEEE 754 introduces a 16-bit format referred to as *binary16* (half-precision). The introduction of *binary16* represents the first step to increase the energy efficiency of FP computations. However, software development flows for ULP systems still lack a methodology to evaluate the effect of reduced-precision FP variables on application requirements. In practice, programmers often use the maximum precision provided by target platforms, following the most conservative principle: guaranteeing each elementary step's numerical precision also guarantees final results precision.

In recent years, significant advances in the field of approximate computing have been made aimed at relaxing this precise-computing abstraction [51, 52, 53, 54]. The most promising research trends are stepping beyond the concept of approximation itself towards a novel paradigm, *transprecision computing*. In TP, rather than *tolerating* errors implied by imprecise HW or SW computations, systems are explicitly designed to deliver just the required precision for intermediate computations. In other words, the specified constraints on the precision of final results are always met, i.e., results are not generically "approximated"; they match a required precision. However, intermediate operations can be moved to custom, lower-precision computation units to save energy.

While approximate computing shows flexibility with low precision arithmetic and aggressive bit width reduction [53], much of the application spectrum adheres to the IEEE 754 standard for FP arithmetic despite its possible adverse side effects. The standard formats mainly suffer from rigid allocation of bits to their sign, exponent, and mantissa fields. They lack robustness to rounding errors [55, 56], caused by the implicit rounding rules defined in the standard. When a value lies in between two representable FP values, it will be rounded, producing an inevitable rounding error; across multiple calculations, such rounding error can be accumulated without allowing the application a direct observation or control over the error.

TP computing, with its aim at fine-tunable control over the preci-

sion of computations, thus implies a wide range of demands that make efficient hardware solutions that retain as much flexibility as possible highly desirable.

The main contributions of this chapter are:

1. The introduction of a software library that enables explorations of FP types. We present a methodology to integrate our library with external tools for precision tuning. We also explore an energy-efficient hardware design with support for multiple FP types (Section 2.3).

2. A brief overview of the unum format and considerations for implementation in hardware (Section 2.4).

3. A brief overview of the posit format and considerations for implementation in hardware (Section 2.5).

The remainder of this chapter is organized as follows: Section 2.2 gives an overview of FP arithmetic. Section 2.3 covers the extension of the FP type system for TP computing, and Sections 2.4 and 2.5 analyze unum and posit, respectively. The final section provides a conclusion.

2.2 Primer on Floating-Point Arithmetic

Computer number formats are ways of representing numerical data using a string of binary digits. For integer data, either a plain unsigned binary encoding or the two's complement formats are commonly used. As accurately representing real-valued data generally is not possible using a limited number of bits, several encodings exist to approximate these values with different aims and trade-offs.

Fixed-point representations interpret a string of bits as binary fractions, statically assigning a portion of bits – hence the name *fixed*-point – to represent either integer or fractional components of the number. As such, fixed-point representations are barely more complex to handle in hardware than integers. However, the representation suffers from a small range of representable values (called *dynamic range*) and a relatively low resolution of encodable values (called *precision*).

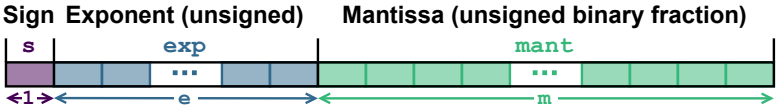


Figure 2.1: Encoding used for IEEE 754 binary FP formats.

A more complex encoding scheme is floating-point (FP) representations, which are the most widely used method of encoding real values in computer systems. In FP, a fractional *mantissa* is scaled using a separately encoded *exponent*, allowing to cover a very large dynamic range while offering reasonable precision [57].

2.2.1 IEEE 754 Floating-Point

FP arithmetic was standardized in the IEEE 754 standard. The latest major revision of this standard, IEEE 754-2008, was published in August 2008 and is widely used [49]. A minor revision containing mainly clarifications was published in 2019. While the standard specifies encodings for both binary (base-2) and decimal (base-10) FP, this work focuses exclusively on binary formats.

Binary Encoding

The binary representation of an IEEE 754 FP value consists of three components:

Sign (`s`) One sign bit to indicate a positive or negative value;
Exponent (`exp`) e bits representing the biased exponent;
Mantissa (`mant`) m bits representing the fractional mantissa.

These three components are appended to form a binary FP datum as shown in Fig. 2.1. The numerical interpretation of this binary representation is defined in Eq. (2.1), where $(\dots)_2$ denotes interpretation as an unsigned binary value. Note that the leading `1` bit of the mantissa is not encoded in the binary representation and that the exponent field is biased to center the dynamic range around ± 1.0. All values interpreted according to Eq. (2.1) are considered *normal numbers* in IEEE 754 parlance.

Table 2.1: The IEEE 754 encodings with special interpretations.

Exponent	Mantissa	Encoded Value
$(\texttt{00}\ldots\texttt{0})_2$	$= 0$	± 0
$(\texttt{00}\ldots\texttt{0})_2$	$\neq 0$	Subnormal numbers
$(\texttt{11}\ldots\texttt{1})_2$	$= 0$	$\pm\infty$
$(\texttt{11}\ldots\texttt{1})_2$	$\neq 0$	not a number (NaN)

$$\begin{gathered}(-1)^{\texttt{s}} \times 2^{(\texttt{exp})_2 - \mathrm{BIAS}} \times 1.(\texttt{mant})_2 \\ \mathrm{BIAS} = 2^{m-1} - 1\end{gathered} \tag{2.1}$$

Special Cases

IEEE 754 defines exceptional cases in which the bit patterns are interpreted differently, as shown in Table 2.1. A group of numbers with small magnitude, signified by an all-zero exponent field and non-zero mantissa, is called *subnormal numbers*. Their value is interpreted according to Eq. (2.2). The difference is that the implied integer bit of the mantissa is now `0`, and the exponent is set to the minimum normal exponent. This special handling creates a linear space of values between ± 0 and the smallest normal numbers and is referred to as *gradual underflow*.

$$(-1)^{\texttt{s}} \times 2^{1-\mathrm{BIAS}} \times 0.(\texttt{mant})_2 \tag{2.2}$$

Furthermore, a group of bit patterns is reserved to signify non-numerical data called *NaN*. These particular values are used for invalid operations and when the result of an operation would be mathematically undefined.

Rounding Modes and Exception Flags

As FP representations attempt to map the infinitely many real values to a limited number of available bit patterns, quantization error can be introduced when operating on FP data. This mapping of real values to representable FP values is shown in Fig. 2.2. In order to control

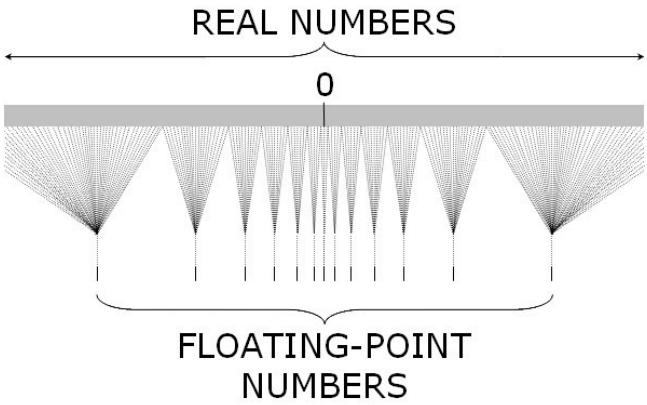


Figure 2.2: Mapping of real numbers to FP values. Taken from [58].

the error incurred in FP operations, the standard defines multiple rounding modes to select from the two closest representable FP values:

To nearest, ties to even The default rounding mode where the closest representable FP value is chosen, and halfway cases are mapped to the representation with an even mantissa;

To nearest, ties away from 0 The closest representable FP value is chosen, and halfway cases are mapped to the larger magnitude value;

Toward zero Always chooses the value closest to 0;

Downward Always rounds towards negative infinity;

Upward Always rounds towards positive infinity.

The standard defines an exception flag for inexact results to signal whether an operation has incurred a rounding error. Note that non-representable real values can become mapped to FP representations of the exact zero and infinity values due to rounding. To this end, IEEE 754 includes exception flags for cases of overflow and underflow. Furthermore, invalid operations such as division by zero are also signaled.

Table 2.2: The binary FP interchange formats defined in IEEE 754.

Format	Common Name	# bits	e	m
binary16	Half Precision	16	5	10
binary32	Single Precision	32	8	23
binary64	Double Precision	64	11	52
binary128	Quadruple Precision	128	15	112
binary256	Octuple Precision	256	19	237

Table 2.3: Dynamic range of the FP formats from Table 2.2.

Format	Minimum Value (Subnormal)	Minimum Value (Normal)	Maximum Value
binary16	$\approx 5.96 \times 10^{-8}$	$\approx 6.10 \times 10^{-5}$	65504
binary32	$\approx 1.40 \times 10^{-45}$	$\approx 1.18 \times 10^{-38}$	$\approx 3.40 \times 10^{38}$
binary64	$\approx 4.94 \times 10^{-324}$	$\approx 2.23 \times 10^{-308}$	$\approx 1.80 \times 10^{308}$
binary128	$\approx 6.48 \times 10^{-4966}$	$\approx 3.36 \times 10^{-4932}$	$\approx 1.19 \times 10^{4932}$
binary256	$\approx 2.25 \times 10^{-78984}$	$\approx 2.48 \times 10^{-78913}$	$\approx 1.61 \times 10^{78913}$

Official Formats

The standard defines five binary interchange formats with fixed exponent and mantissa lengths as shown in Table 2.2, and the range of the numerical values they cover is given in Table 2.3. Of these formats, only *binary32* and *binary64* are considered ubiquitous in general-purpose computing. The precision and range offered by *binary64* are adequate for most problems; thus, the 64-bit format is by default used in most high-performance and scientific computing workloads. The larger formats find use only rarely as their increased range is often unnecessary, and implementations suffer from drastically higher performance and power penalties and are seldom supported in hardware. Higher performance and energy efficiency are usually achieved using the 32-bit FP format, which is sufficient for most general-purpose processing tasks. The *binary16* format, initially introduced for image processing applications, lacks the dynamic range to be broadly supported; however it has found increased use for ML inference in re-

cent years. Performance-constrained and low-power embedded devices usually only support the *binary32* format if FP support is present at all.

2.2.2 Considerations for FP in Hardware

Hardware implementations of arithmetic operations on FP values are significantly more complex than their integer counterparts, as computations usually cannot occur directly on the bit-patterns used by the IEEE 754 formats. Essentially, the values encoded by the FP data first require manipulation into more suitable internal representations before they can be processed using simple arithmetic circuits. Even basic arithmetic operations like addition and multiplication become significant contributors to circuit complexity when implemented in hardware.

Operation Primitives

As FP values are essentially represented by a fractional mantissa scaled by the exponent, the general approach uses separate data paths for the exponent and mantissa computations.

In order to add or subtract two FP values, the mantissae must be appropriately scaled before the corresponding bits can be added together. As such, exponents are compared, and shifters are required to align the mantissa bits before a regular adder circuit can be used to compute the resulting mantissa. For multiplications, thanks to the similarity of FP encoding to scientific notation, mantissae are multiplied directly, and the result exponent is computed from the argument exponents.

Rounding is necessary for all FP computations and boils down to the choice between two consecutive FP values as the result of an operation. Proper rounding decisions require information about the actual (infinitely-precise) result of the computation, which usually necessitates computing more bits of the result mantissa than just what is needed for the representation of the FP value. Many methods exist to obtain the rounded FP result of an operation, such as multiplexing between the two possible results or conditionally incrementing the

appropriate mantissa bits, and their use depends on the operation carried out [57].

Intricacies of IEEE 754

Everyday computational operations such as addition and multiplication must be provided with any compliant FP implementation, as well as comparisons amongst FP values. Additionally, conversions between FP and integers and between different FP formats have to be implemented. IEEE 754 furthermore mandates the fused multiply-add (FMA) operation as a standard arithmetic operation. The FMA operation computes $(a \times b) + c$ and applies only one final rounding step to the operation's infinitely precise result. It is usually used to reduce the error introduced by repeated rounding in applications that perform many accumulation-type operations, such as matrix-multiplications or convolutions [55].

Hardware implementations of FMA usually are the largest and most timing-critical. The requirement for accurate rounding necessitates keeping many more precision bits until the end of the operation, resulting in very wide internal data paths. Due to the significant overhead of hardware FMA, regular addition and multiplication operations often reuse the FMA unit to reduce additional overheads. Using hardware FMA can improve code size and performance compared to issuing separate multiply and add instructions.

Implementing the different rounding modes in hardware does not incur as much overhead as the standard "round to nearest, ties to even" rounding mode is the most complex to construct. Also, producing exception flags alongside the operation results can usually be done using information already available somewhere in the computation unit's datapath.

2.3 Extensions to the FP Type System

In this section, we propose an extended FP type system with complete hardware support to enable TP computing on ULP embedded platforms. We propose the introduction of two additional formats, namely *binary8* and *binary16alt*. Specifically, *binary8* is a 8-bit format with

low precision (3-bit mantissa), while *binary16alt* is a 16-bit format complementary to the IEEE 754 one and featuring a higher dynamic range (8-bit exponent)[1]. To assess the benefits of this extended FP type system, we performed a precision analysis supported by additional considerations on the hardware design. As a first step, we designed a C++ library to explore the effects on application behavior when varying the dynamic range and precision of program variables. Then we modified a set of applications representative of FP-intensive computations in the embedded domain, adopting emulated FP types and providing an interface with an external tool for precision analysis [53].

Our results show that the introduction of *binary8* guarantees the best trade-off between precision and dynamic range for applications that match minimum precision requirements. Moreover, the introduction of *binary16alt* extends (up to 50%) the number of variables that can be safety scaled from a 32-bit representation to a 16-bit one.

To provide support at the hardware level, we designed a dedicated *TP-FPU*. Our design also enables vectorial operations on sub-32-bit formats, further increasing the core's energy efficiency and performance and reducing data memory pressure.

Experimental results show that up to 90% of FP operations can be safely scaled down to 8-bit or 16-bit formats. Thanks to precision tuning and vectorization, execution time is decreased by 12%, and memory accesses are reduced by 27% on average. As a significant outcome, energy consumption is reduced up to 30%.

2.3.1 Floating-Point Types And Programming Flow

Exploration Of Floating-Point Formats

Applications that operate on real-valued data most commonly use IEEE 754-compliant FP formats [49]. Of the standard formats, *binary32* and *binary64* enjoy the most widespread use and are also available on general-purpose consumer platforms. While even larger formats are commonly used for scientific computations, reducing the

[1]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, NaN, and support all rounding modes.

amount of data to process. Hence, the width of FP formats is more suitable for power-constrained and embedded platforms. While sub-32-bit FP formats (also called *minifloat*s) have been use in computer graphics applications [59], their relevance is rising with the spread of energy-constrained computing platforms, such as near-sensor processing nodes and Internet-of-things endpoints. IEEE 754 formats are packed as the sign bit, e bits for the exponent, and m bits for the significand (or mantissa).

By choosing a specific format, programmers enforce a trade-off between dynamic range and precision. The dynamic range is the ratio between the largest and smallest representable values, conditioned by e. Conversely, the precision is the number of digits of the represented number preserved in FP representation, and it is uniquely defined by m.

As discussed in Section 2.3.4, available tools are not flexible enough to simulate arbitrary FP formats by tuning both precision and dynamic range. To enable exploration of arbitrary FP types, we designed a dedicated C++ library, called *FlexFloat*. This library provides a generic FP type by defining a template class (`flexfloat<e,m>`) and a set of auxiliary functions for debugging and collecting statistics.

Using *FlexFloat*, all FP types used in the source files can be safely replaced with instantiations of this template class without changing any other part of the program since class methods include operator overloading. The template parameters include the number of bits used for the exponent (e) and the number of bits used for the mantissa (m), which must be specified as positive integer values. For instance, `flexfloat<7, 12>` is a FP type including the sign bit, 7 bit in the exponent field and 13 bit in the mantissa field. The *FlexFloat* library also supports the encoding of subnormal numbers, infinities, and NaN values. Arithmetic operations are performed converting the number representation to a native type (e.g., `double`) and then *sanitizing* the result, that is, adjusting exponent and mantissa to obtain the exact binary representation of the original type. This methodology guarantees shorter execution times w.r.t. emulation approaches (e.g., *SoftFloat*), and it also produces the same results of a dedicated hardware unit (i.e., precise at bit level).

An automatic cast between different template instances is not allowed, so standard arithmetic operations must involve variables of the

same instance. This design choice enables a fine-grain control on the intermediate results since the compiler notifies an error for each operator involving a type mismatch. Consequently, programmers can choose to match the variable types to the same template instance or insert an explicit cast. A constructor supporting explicit conversions is provided, and it can be used to cast a *FlexFloat* variable to a different template instance (e.g., from `flexfloat<e1,m1>` to `flexfloat<e2,m2>`). Constructors with implicit semantics are provided for standard FP types (`float`, `double` and `long double`) to simplify the usage of FP literal values. Vice versa, an automatic cast from a FlexFloat template instance to a standard FP type is not allowed, but it can be performed by invoking an explicit cast operator. This feature can be used to interface sections of source code that use FlexFloat and sections strictly bound to standard types (e.g., a call to an external library function whose source code is not available).

The main benefits of *FlexFloat* are:

- It produces binaries that are fast to execute since its computations rely on native types;
- It reduces the debugging effort, as the compiler performs early check upon template instantiation;
- It is quite intuitive to use since it provides the usual infix notation for arithmetic operations;
- It can be easily integrated with external tools, having no specific requirements w.r.t. the source code.

To simplify the interaction between a *FlexFloat*-based program and any external tool, we designed a *FlexFloat wrapper*, that is a support tool performing three steps: (i) it reads a file specifying a required precision for each program variable, then (ii) it extracts the dynamic range from a configuration file providing the map indexed by precision intervals, and finally (iii) it compiles the program sources providing the derived values for precision and dynamic range as actual parameters in the template instantiations.

To perform an exploration of FP types, we used the *DistributedSearch* tool introduced in Section 2.3.4. Since this tool performs precision tuning without considering the dynamic range of variables,

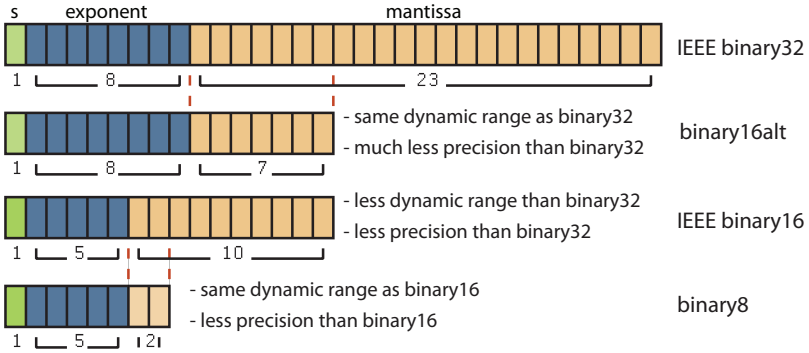


Figure 2.3: Overview of FP formats used throughout this work.

we assumed a limited set of initial hypotheses to fix the dynamic range associated with specific intervals of precision bits. Considering our target on ULP systems, we restricted our investigation to 8-bit and 16-bit formats.

Among potential 8-bit formats, we chose the mapping $(0, 3] \mapsto 5$, calling this type *binary8*. Thus, any variable associated with precision between 1 and 5 bit will be provided with an exponent of 5 bit. This format was conceived to mirror the dynamic range of *binary16* variables. Adopting this convention, conversions between *binary8* and *binary16* only affect precision but do not saturate for values of large magnitudes. Additionally, operations on *binary8* become very cheap in hardware since it contains only two explicit mantissa bits.

As regards 16-bit formats, we considered the mapping corresponding to *binary16*, that is $(0, 11] \mapsto 5$, and an alternative mapping that we called *binary16alt*, corresponding to $(0, 8] \mapsto 8$. 8 is the number of bits used for the exponent field in *binary32*, so this value is a reasonable upper bound for any 16-bit format. Again, using the same number of exponent bits of the *binary32* format makes conversions much cheaper. Figure 2.3 summarizes the FP formats used throughout this work.

Table 2.4 shows the results of our preliminary analysis, reporting the total number of variables associated with each type. These values are obtained executing *DistributedSearch* on our set of benchmarks constrained with a precision of 10^{-1}. We considered two different

Table 2.4: Variables classified by type type using V1 and V2 type systems.

	binary8	**binary16**	**binary16alt**	**binary32**
V1	10	29	-	72
V2	19	10	41	41

configurations of the FP type system, namely V1 (including *binary8*, *binary16*, *binary32*) and V2 (adding *binary16alt* to V1).

As a first consideration, *binary8* is used for 17% of the variables in the best case. This format is highly beneficial in reducing energy consumption since it simplifies circuitry complexity and enables vectorization. It is noteworthy that supporting both 16-bit formats contributes to decreasing the number of 32-bit variables in the program w.r.t. the usage of a single 16-bit format. A drawback of *binary16* is that both dynamic range and precision are diminished compared to *binary32*. Sometimes, this leads to saturation when converting values with high dynamics from *binary32*, disqualifying the 16-bit format from being used for TP tuning in these cases. Conversely, *binary16alt* features the same dynamic range as *binary32*, allowing the whole range of values to be converted - albeit with a much larger granularity. In some cases, applications do not exploit the dynamic range provided by *binary16alt*, and at the same time, they require higher precision, so our intuition is that we need both types. A further evaluation is provided in Section 2.3.3.

Transprecision Programming Flow

Figure 2.4 depicts the TP programming flow that we adopted throughout this work. As a first step, application sources are modified to replace standard FP types with multiple instances of `flexfloat<ex,mx>`, where *ex* and *mx* are variable-specific parameters. A tool for precision tuning is invoked (step 2), and different values for *ex* and *mx* are explored using the *FlexFloat wrapper*. After this tuning, program variables are uniquely mapped to supported FP types (step 3). Using this mapping, *FlexFloat* can provide statistics on the number of operations and casts performed for each FP type, which is instantiated

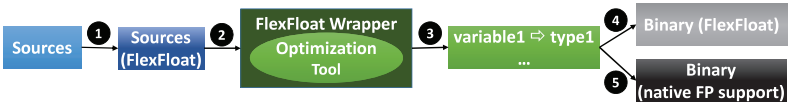


Figure 2.4: Overview of the programming flow.

(step 4). Moreover, a version of FlexFloat providing explicit template specialization is provided to replace simulated operations with native ones (step 5). This step requires that the target platform's compiler supports all the FP types provided by the mapping.

2.3.2 Transprecision Floating Point Unit Prototype

To evaluate the potential of the FP formats introduced in Section 2.3.1, we designed a prototype TP-FPU supporting vectorization of reduced-precision operations. The hardware unit is built up from three types of slices, each with a fixed width of 32-bit, 16-bit and 8-bit, respectively. Each slice hosts operations on the FP formats that match the slice width and conversion operations. The supported arithmetic operations are addition, subtraction, and multiplication. The conversion operations include casts to and from integers (both signed and unsigned) and casts among the FP formats. Moreover, the narrower slices are replicated in order to enable sub-word parallelism inside the unit. Thus, two 16-bit or four 8-bit FP operations can be executed simultaneously. Following the SIMD paradigm, the proposed unit can run scalar operations when only one slice for a given precision is active and vectorial operation when all the slices of a given precision are active.

The various individual operation blocks are instances of Synopsys DesignWare FP Datapath components. As a power-saving feature, the unit employs operand silencing to all unused operations and formats by forcing zero to prevent transistor switching. Arithmetic operations in *binary32*, as well as both 16-bit formats, are pipelined with one stage to meet the container core's timing requirements featuring a bandwidth of one operation per cycle and a latency of two clock cycles. Arithmetic operations in *binary8* and all conversion operations have a one-cycle

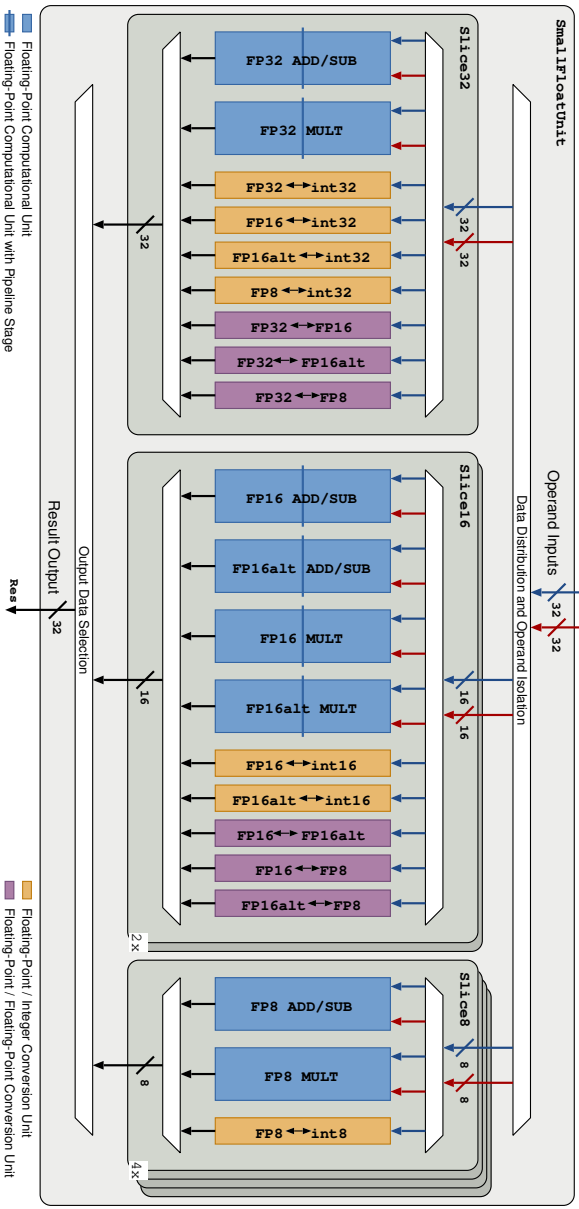


Figure 2.5: Simplified block diagram of the designed hardware unit datapath. Control logic as well as data to and from duplicated slices is omitted.

latency. Area optimization of the TP-FPU and its integration into the core will be completed as future work.

2.3.3 Experimental Results

Evaluation Methodology

Experiments have been performed on a set of applications that implement key algorithms for two domains of ULP systems, near-sensor computing, and embedded machine learning:

- JACOBI applies the Jacobi method to a 2D heat grid:
- KNN computes the k-nearest neighbors of an input value using euclidean distance;
- PCA performs the principal component analysis;
- DWT computes the discrete wavelet transform;
- SVM is the prediction stage of a support vector machine;
- CONV implements a 5×5 convolution kernel.

Precision tuning has been performed using the *fpPrecisionTuning* toolsuite on an x86 workstation, adopting the programming flow described in Section 2.3.1. Since sub-word vectorization is not supported by the current *FlexFloat* implementation, we manually tagged vectorizable sections in the source code. The library provides a separate report for vectorial operations and casts. The application sources have been compiled using the GCC compiler with a RISC-V back-end optimized for PULPino, which supports the single-precision FP type defined in the RISC-V instruction set architecture (ISA). Binaries have been executed on the PULPino virtual platform, which is cycle-accurate and provides detailed statistics. The virtual platform reports the number of cycles required to execute each instruction used in the binary file, targeting the whole program or delimited code regions. The current version of GCC does not include a set of instructions to handle *binary16*, *binary16alt* and *binary8* formats. Since the latency of *binary16* operations is the same as *binary32* ones, we have used the *binary32* type to measure the exact number of cycles required by

each instruction to execute. This value depends on the ability of the compiler to schedule other classes of operations (non-FP, *binary8* or casts) to fill latency cycles and avoid stalls in the core pipeline, so it is strictly dependent on both application and compiler back-end. *Binary8* operations and FP conversions always require a single cycle, so their contribution to execution time has been accumulated analytically.

For evaluation of the hardware architecture, the design unit was synthesized for the UMC 65 nm technology using worst case libraries (1.08 V, 125 °C). To have an accurate estimation of the power consumption of the TP-FPU, we performed post-place-&-route power simulations. The target frequency for the post-layout design was set to 350 MHz, using worst-case conditions. Results take into account the switching activity of input and output registers, added at the unit's boundaries to evaluate their performance, negligible to the power of the arithmetic units themselves. Energy costs of FP operations were obtained through simulation of the post-layout design in all modes of operation, again using worst-case conditions. Values provided to the unit were generated randomly, making sure that no invalid values were generated. Namely, no NaN or infinity values were applied, and operands were chosen sufficiently close to each other such that operand cancellation would not occur during addition or subtraction. For conversions, only mappable values to the target type were used to eliminate over- and underflow, simulating normal operations on meaningful data. We prevent operand cancellation or invalid inputs, which would lead to significantly diminished switching activity inside the operational units. The energy cost of each non-FP instruction executed by the core includes core logic, instruction memory, and data memory. Even if the TP-FPU has not been integrated into PULPino's core yet to collect energy measurements, we have considered the contribution of moving data to/from the input and output registers of the FPU. Furthermore, we also consider the cost of idle cycles due to pipeline stalls (for both 16-bit and 32-bit instructions).

Precision Tuning

The table in Fig. 2.6 shows the results of the precision tuning, performed for three precision requirements (signal-to-quantization-noise ratio (SQNR) = 10^{-3}, 10^{-2}, 10^{-1}). Rows correspond to applications

		binary8	binary16alt					binary16			binary32	
SQNR	Application	Precision (mantissa bits)										
		3	4	5	6	7	8	9	10	11	12	13-24
10^{-3}	JACOBI	1164	2	0	1	2	1	0	0	0	0	1165
	KNN	0	2005	0	0	30001	0	0	0	0	0	0
	PCA	0	5	5	0	11	13	5896	1	7	1	574
	DWT	0	0	0	1	0	1	5849	0	0	0	1
	SVM	11025	0	0	0	2	2	37	3	0	0	1
	CONV	0	0	0	0	0	1586	1	0	0	0	0
10^{-2}	JACOBI	0	1165	4	1	0	0	0	0	0	0	1165
	KNN	32006	0	0	0	0	0	0	0	0	0	0
	PCA	5	12	14	12	6424	4	4	0	0	37	1
	DWT	0	1	0	1	0	0	5849	0	0	1	0
	SVM	11025	1	2	1	2	38	0	0	0	0	1
	CONV	0	0	0	0	0	1586	1	0	0	0	0
10^{-1}	JACOBI	1167	3	0	0	0	0	0	0	0	0	1165
	KNN	32006	0	0	0	0	0	0	0	0	0	0
	PCA	6	5911	12	11	537	24	2	0	0	10	0
	DWT	0	2	0	5849	0	1	0	0	0	0	0
	SVM	11025	4	39	1	0	0	0	0	0	0	1
	CONV	1586	0	0	0	0	0	1	0	0	0	0

Figure 2.6: Precision tuning of program variables for three precision requirements.

and columns to precision bits. The reported values represent the number of memory locations (scalar variables or array elements) requiring the minimum number of bits in their column to meet precision constraints. Color bands show the mapping between precision bits and the FP type system introduced in Section 2.3.1.

KNN and SVM make wide use of *binary8* data, while other applications do not. *binary8* emerges a format that is profitable in specific application domains, while *binary16* is a good candidate for broader use. Moreover, most of the interval $[9, 11]$ are concentrated in column 9, which is the minimum number of precision bits required for a *binary16* type. These elements strictly require the additional precision provided by *binary16* w.r *binary16alt*, which means that both types are useful in different contexts. For the same reason, there are more variables in column 4 than in column 5, since they include all cases that do not require a wider dynamic range w.r.t. *binary8* (regardless of the precision). Conversely, variables that require high precision usually require more than 12 precision bits, and they are concentrated in the last column.

Execution Time And Memory Accesses

Figure 2.7 shows a breakdown of the FP operations performed by each application, taking into account the previous section's precision requirements. Pictured is a dynamic view of the FP type system at run-time (whereas Fig. 2.6 provides a static view after precision tuning). Each bar segment quantifies the contribution of a specific type to the total number of FP operations, discriminating scalar and vectorial operations.

In JACOBI and PCA, there is a significant contribution of 32-bit operations, which is a first trait that adversely affects a potential reduction of energy consumption. Another negative aspect is the lack of vectorial operations, which is pathological in JACOBI. We have not considered any advanced coding techniques (e.g., manual code vectorization). However, we have based our analysis on off-the-shelf versions of applications that could be further optimized to follow the guidelines derived from our considerations.

Figure 2.8 depicts two groups of bars for each application, reporting memory accesses and execution cycles. Values are normalized to the

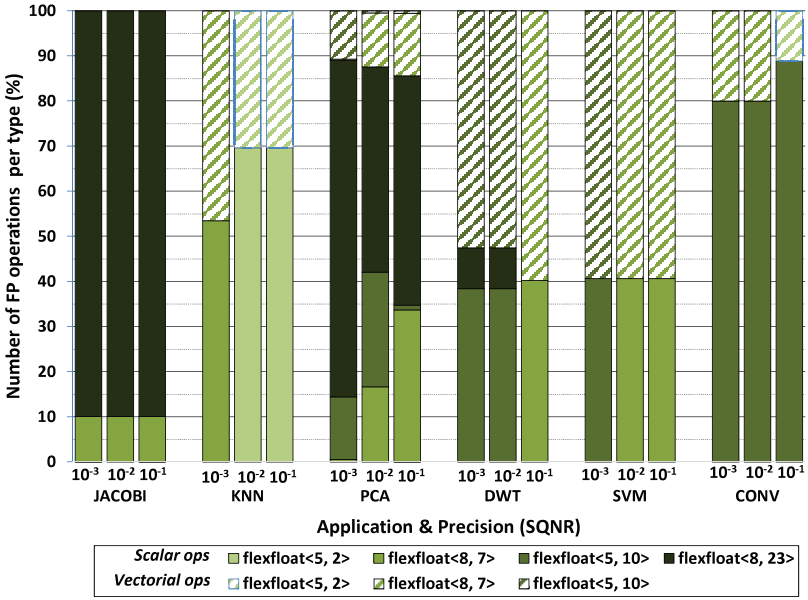


Figure 2.7: Breakdown of FP operations for three precision requirements.

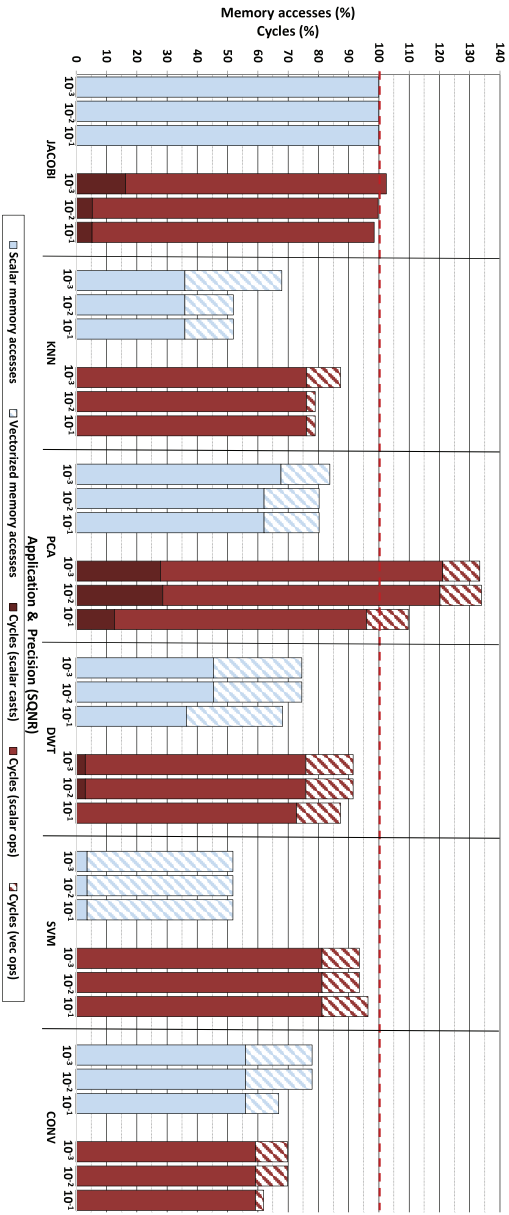


Figure 2.8: Memory access and cycles for three precision requirements, normalized to *binary32* baseline.

binary32 version of the application, which acts as a baseline. Vectorial memory accesses, cycles spent in vectorial operations, and cycles spent in cast operations are highlighted differently.

As shown in the previous section, JACOBI does not perform any vectorial operation. Moreover, the number of cycles is equivalent to the original version since this application only uses a limited number of *binary16alt* variables without exploiting vectorization. In the most general case, the number of cycles can even exceed the baseline since cast operations between different FP types are introduced (e.g., JACOBI when SQNR = 10^{-3}). As a significant limitation, current tools for precision tuning do not consider the cost of casts since they aim to minimize the number of precision bits used by any variable. No other optimization goal can be specified here. This effect is further exacerbated in PCA, where the number of casts required after the tuning process exceeds 10% (SQNR = 10^{-1}) and 20% (SQNR = 10^{-2} and 10^{-3}).

We can observe evident benefits in memory accesses and cycles in other benchmarks, mainly due to vectorization, while the overhead of cast operations is not relevant. SVM shows the maximum reduction of memory accesses, that is 48% since 60% of FP operations are vectorizable (for any precision requirement). On average, the execution time is decreased by 12%, and memory accesses are reduced by 27%. Considering JACOBI and PCA as outliers, these values turn into 17% and 36%.

Energy Consumption

Figure 2.9 shows the energy consumption of each application, normalized to the *binary32* baseline. Each bar contains three contributions, the FP operations (FP ops), the memory accesses (Memory ops), and all the other instructions that are executed by the core (Other ops).

These numbers can be easily justified by the considerations in the previous section. On average, JACOBI's energy consumption is 97% since this application makes limited use of sub-32-bit types and does not exploit the benefits of vectorization. The energy consumption of PCA is 7% and 8% greater than the baseline for two precision requirements (SQNR = 10^{-3} and 10^{-2}), due to the high number of casts coupled with a predominant number of scalar operations on *binary32* values.

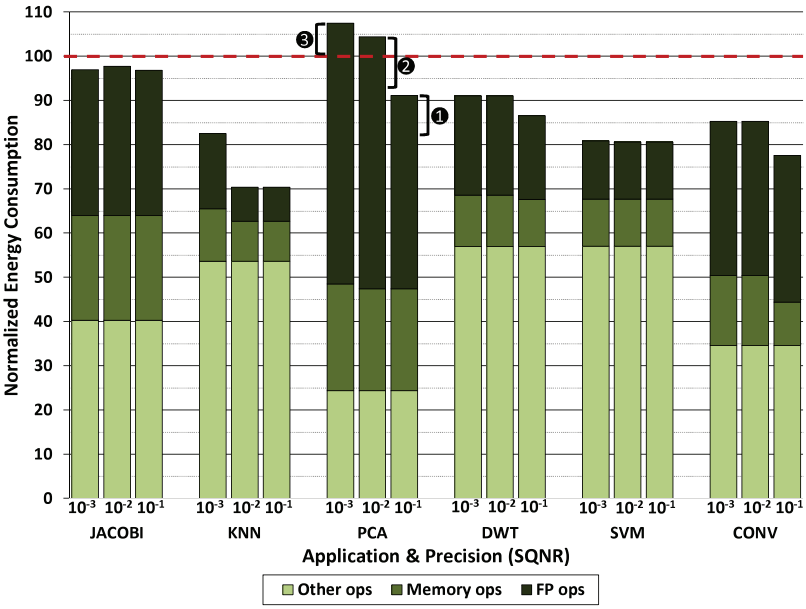


Figure 2.9: Energy consumption normalized to *binary32* baseline.

The other applications have average energy savings of around 18% compared to the baseline, with a maximum of 30% measured for KNN. Considering the results of Section 2.3.3, the behavior of KNN is related to three main characteristics, (i) it uses the *binary8* type for all program variables, (ii) it exploits vectorization, and finally (iii) it requires a limited number of non-vectorized memory accesses.

Advanced vectorization techniques can provide tremendous benefits whenever an application provides a relevant percentage of sub-32-bit operations after the tuning process. We applied manual vectorization to PCA to demonstrate this assumption, thus reducing the energy consumption to lower values (101%, 96%, and 85%). These gains are marked on Fig. 2.9 by labels 1, 2, and 3. Further energy savings can only be achieved by reducing casts' contribution by using more intelligent tools for precision tuning.

2.3.4 Related Work

To overcome the limitations of fixed-format FP types, researchers have proposed multiple-precision arithmetic libraries that perform calculations on numbers with arbitrary precision. ARPREC [60] and MPFR [61] are two widely used libraries that provide support to multiple-precision arithmetic. These libraries are mainly used in contexts where a high dynamic range is required and higher computation time is considered an unavoidable side-effect, such as scientific computing. However, they are not suitable to perform explorations of sub-32-bit FP types since they represent exponents using an entire machine word. This approach prevents simulation of the behavior of FP types with a reduced number of bits since tuning of dynamic range is not possible.

SoftFloat [62] is a library that implements standard IEEE 754 formats, enabling a bit-accurate emulation of the FP operations performed by FP hardware units. *Softfloat* can be easily extended to support additional formats, including the ones introduced in this work. However, program executions are enormously prolonged since the library executes all the computations in software. Moreover, any change to a FP format requires manually modifying code in several source files. Overall, the solutions mentioned above require a complete refactoring of the source code; in some cases, additional software layers have been introduced to perform this task (e.g., the *Boost* interval arithmetic

library [63]).

Many research tools are available to perform automatic or semi-automatic precision tuning of program variables. In this chapter we use *DistributedSearch*, a tool provided by the *fpPrecisionTuning* [53] toolchain that finds a near-optimal solution. Its main configuration parameter is the result's precision, expressed as a value of the SQNR that program outputs must satisfy. This tool requires a binary version of the target program, a target output (i.e., a sequence of FP numbers that are the exact result), and a configuration file. The configuration file should include a list of numbers corresponding to the precision bits used for program variables. *DistributedSearch* requires that the target executable read the configuration file to tune the precision of its variables accordingly. Also, the program must provide its output results on the standard output. The tool runs the program multiple times on this premise, performing a heuristic search of each variable's minimum precision (for a fixed input set). A second phase performs a statistical refinement to join the precision bindings derived from different input sets.

Other tools adopt more advanced techniques but their search space is restricted to standard FP types (e.g., PROMISE [64] and *Precimonious* [65]), or in other cases they are limited to the analysis of functional expressions (e.g., *FPTuner* [54] and PRECISA [66]). As a final consideration, all these tools do not enable the analysis of the dynamic range associated with a fixed-format FP type.

On the hardware side, several recent works proposed the design of energy-efficient FPUs. Kaul et al. [67] implement a variable-precision multiply-and-add FPU supporting vectorization. Its configurations use an 8-bit exponent field. Each operand carries a 5-bit certainty field, which is processed in parallel with the exponent logic, indicating the number of bits for the mantissa. The certainty field is used to implement automatic precision tracking, which raises precision where it does not meet specified requirements. Considering an energy consumption of 19.4 pJ/flop, this solution seems to perform similarly to our hardware design. However, the memory overhead due to precision tracking and fixed-size exponents is relevant since the memory transfers are a significant contributor to the total energy consumption. Moreover, applications that require 32-bit variables are very inefficient due to repeated operations at lower precision that are performed until a final

retry at single precision is executed.

Tong et al. [68] explore an iterative (digit-serial) multiplier that can be used inside a FP multiplier. Their design processes 8 bit per cycle, thus operands with up to 8-bits use one cycle, operands with up to 16-bit use 2 cycles, and finally operands up to 24 bit use 3 cycles. Power is reduced by 66% when using the one-cycle configuration and by 30% when using the two-cycles one. Again, single-precision operations become slower, and memory effects are not considered.

Rzayev et al. [69] explore various sub-32-bit formats for deep learning applications. They introduce a 8-bit FP format that is identical to *binary8*, showing that vectorization enables higher performance and reduces memory energy used per operation. However, they do not propose a mixed-precision FP type system for TP computing.

Gautschi et al. [70] propose a shared FPU adopting the logarithmic number system (LNU), which is up to 4.1× more energy-efficient than standard FPU in non-linear processing kernels. However, LNU is a domain-specific approach, and not all FP operations can be implemented.

2.4 Alternatives to FP Arithmetics: Unum

In order to better control precision loss, the universal number (unum) format was proposed by Gustafson [71] as an alternative and direct competitor to IEEE 754 FP. It attempts to do so by introducing a variable-width storage format and a flag that determines whether a unum corresponds to an exact number or an interval between exact unums. This flag, called *ubit*, hence explicitly represents when a calculation produces a value that is not exactly representable in the number system. In contrast to IEEE 754 FP, this information is encoded within the value and does not rely on a separate exception side-channel. The unum format additionally defines two fields that make the number self-descriptive.

The unum format, so far, has been supported in various programming environments, including Julia [72], Matlab [73], Python [74], J, and Mathematica [71] languages. Initial efforts on hardware with unum

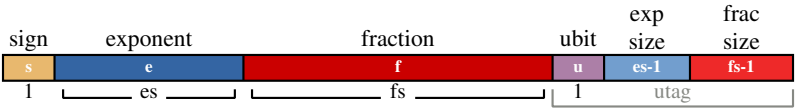


Figure 2.10: Encoding of the unum format, extending IEEE 754 floats with self-descriptive fields in the *utag*.

support focus on early synthesis [75] of three operators (i.e., addition, multiplication, and comparison), and FPGA implementation [76] of four operators (i.e., addition, subtraction, multiplication, and division). To evaluate the benefits and challenges of unum hardware design in silicon, we have presented the first ASIC as a fully operational unum processor capable of performing additions and subtractions as well as format-specific functions for lossless and lossy compressions [40]. This section presents an overview of the unum format and some critical insights for hardware implementations we have gained from said work.

2.4.1 Properties of Unums

Binary Encoding

The unum encoding, depicted in Fig. 2.10 bears similarity to the IEEE 754 FP representation for real numbers with its *sign-exponent-mantissa* notation. The unum format extends this representation by adding three new fields that allow for the inclusion of self-descriptive information about the represented value. These additional fields are summarized under the name *utag*.

The last two fields in the utag denote the exponent size es and fraction size fs of the unum, making unum a variable-size format. Hence, FP values that can be represented with a small number of bits require fewer storage bits than a large fixed-size FP environment thanks to the self-descriptive nature of the utag.

Since it is infeasible to allow unlimited exponent and mantissa sizes in practice, the widths of the exponent size and fraction size fields in the utag are fixed, defining the maximum range of possible unum values. The chosen widths for the exponent size and fraction size fields then define a so-called unum *environment*. For example,

setting the exponent size width to 4 bits and the fraction size width to 5 bits, the resulting environment can represent unums with *up to* 16 exponent and *up to* 32 fraction bits. Such unums are defined in a {4,5}-environment – the maximum possible size of a unum in an {a,b}-environment is given as maxubits $= 2 + 2^a + 2^b + a + b$.

The first field in the utag, called the *ubit*, can be set to denote that the represented value x is not an exact point on the real number line, but rather an open interval $(x, x + ulp)$ with ulp being the unit in the last place for the current unum format. Explicitly encoding that the exact value cannot be represented in the current format sets unum apart from regular FP representations where all encoded values are considered exact, and approximation is entirely implicit.

For describing general intervals more than one *ulp* apart, two unums can be connected to create a so-called *ubound*[2], each denoting one endpoint of an interval. In a ubound, each of the two ubits indicates whether the respective endpoint is part of the interval (closed) or not (open).

Unum Operations

Unum addition is similar to FP addition, with more complex exceptional cases involving infinities being dependent on both values and bound types. The left and right bound of ubounds can be handled independently, however.

One complexity of FP arithmetic, namely rounding, is greatly simplified in unum. Whenever the result of an operation on two exact values requires more precision than available in the unum environment, the ubit is set to mark the value as inexact. When handling bounds, the bound type of the result bound corresponds to the logical-OR of its operand ubits.

Since the bit-pattern representation of a value is not unique within a unum environment, there are additional unum-specific operations to be considered. Since implementations should strive to utilize as few bits as possible for a given value, we also define the lossless *optimize* operation, calculating the representation of a ubound with the smallest number of bits. Furthermore, Gustafson [77] specifies the *unify* operation that

[2]This definition deviates from Gustafson's definition in [77], where the term *ubound* can also denote a single unum with the *ubit* set.

attempts to merge a ubound consisting of two unums into the smallest single unum that fully includes the interval. This operation can incur a loss of precision, namely if the resulting inexact unum covers a larger interval than the initial ubound.

Dynamic Behavior

In order to illustrate the dynamic behavior of unum during calculations, *axpy* was run with input coefficients of rising complexity, calculating and accumulating the result using either floats or unum environments. The relative error change compared to a double-precision reference and the bitwidth over the iterations are shown in Fig. 2.11.

During phase I, only small coefficients produce results that can be exactly represented in all evaluated formats. The size of unum results is made up of the fixed size of the utag – 8-bit and 10-bit, respectively, for the {3,4} and {4,5} environments – and the dynamic number of bits needed to store the actual value.

Phase II applies large coefficients, significantly increasing the accumulated values. Unum formats start increasing in size to store the result still accurately. Once the exact value requires more fraction bits than available in the format, error proportional to the format-specific minimal *ulp*-width appears, and unum starts using ubounds to represent the uncertainty of the results accurately.

In phase III, more error is introduced by using random floats as coefficients, also causing {4,5}-unum's 32 fraction bits to be insufficient for exact results.

The ubounds used for unum results would require significantly more storage space than floats; thus, they should stay contained within the processing unit registers if possible. Before storing to main memory, *unify* can be used to reduce storage size at the cost of increasing the error bound. Unifying excessively, after each iteration as shown in Fig. 2.11, causes the additional error introduced by each unification to accumulate rapidly.

2.4.2 Considerations for Unum in Hardware

The interchange format for unums as shown in Fig. 2.10 is specified in [77]. Unum values reside in memory in this format, using only

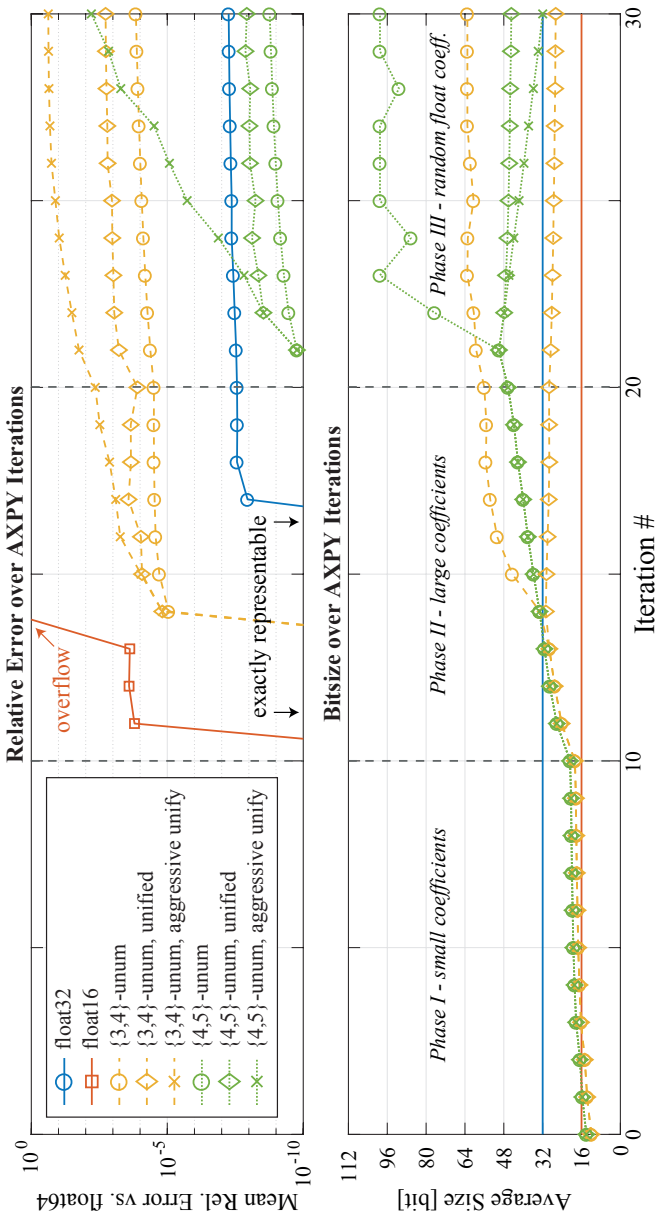


Figure 2.11: Relative error of *axpy* iterations using FP and unum formats (top) and the bit-size of the results (bottom).

as much storage as mandated by the exponent size and fraction size fields – which can be drastically less than using a fixed-width FP representation. However, this departure from using uniformly sized and aligned operands requires additional effort when handling unums in the memory system.

Since arithmetic units and register files must be provisioned for handling all possible unums in a given environment, this incurs a relative hardware overhead for those unums that do not use the maximum width of the environment. Additionally, the constant overhead for storing the utag with each value is especially noticeable in small environments. Unpacking of unum values in the register file and the storage of additional meta-information, called *summary bits* in [77], can simplify the implementation of unum operations, especially the handling of bounds and exceptional cases such as NaN and infinity operands.

Comparison with IEEE 754

In the example from Fig. 2.11, there is a range where unum provides lower memory footprints than IEEE 754 binary32 with equivalent accuracy, while binary16 error already grows rampant. Unified {3,4}-unums require 7% less memory than float32 at the price of a significant error increase similar to float16 – while remaining usable long after float16 overflows due to small range. Unified {4,5}-unums require roughly 45% more storage than single-precision values primarily due to utag overhead – albeit at around 5× lower error and explicitly denoting this error. Using binary32 interval arithmetic to store the error bound would cost 39% more memory than unum in this example.

Figure 2.12 shows synthesis experiments in 65 nm, comparing different unum-enabled arithmetic units with an IEEE 754 compliant FP adder with corresponding exponent and fraction sizes. A first observation is a modest area increase (27% or 1.08 kGE with a 4 ns period constraint) when only considering the unum adder.

However, complementing the adder with the expand and *optimize* units to take advantage of on-the-fly data compression comes with an area increase of more than 3.5×. The implemented, fully parallel ubound adder adds roughly another factor of two while also doubling the throughput. The second important observation is the limitation in

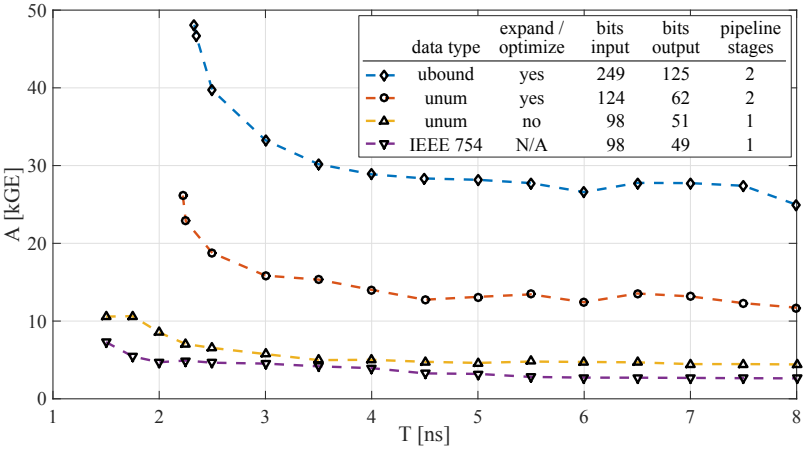


Figure 2.12: Area and timing comparison of our ubound adder in [40] and its sub-parts against an IEEE 754 compliant adder.

terms of the minimum clock period for the compression-enabled unum units, even with additional pipeline stages.

2.5 Alternatives to FP Arithmetics: Posit

Posits are the third version proposed under the name unum by Gustafson [78]. Proponents claim that posits can be used as a direct drop-in replacement for IEEE 754 FP numbers with more advantageous properties than FP. These include a more extensive dynamic range, higher accuracy, better closure, bitwise identical results across systems, simpler hardware, and more straightforward exception handling.

2.5.1 Properties of Posits

Posits were developed as an evolution of the unums format outlined in Section 2.4, and intend to address the shortcomings that format brings

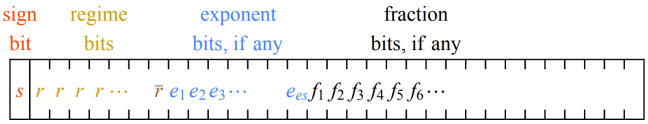


Figure 2.13: Binary representation of an n-bit Posit value with es exponent bits [78].

Table 2.5: The binary encoding of the regime value `k`.

Regime	$(0001)_2$	$(001)_2$	$(01)_2$	$(10)_2$	$(100)_2$	$(1000)_2$
k	-3	-2	-1	0	1	2

in terms of implementation in hardware. Most notably, the interchange format is no longer variable in length, although the encoding retains some aspects of variable-length encoding.

Binary Encoding

Figure 2.13 shows the binary representation of a posit which is built from the same components as FP numbers, plus an additional field called the *regime*. The regime value `k` serves as an exponentiation factor for the exponent itself, allowing posits to cover an extensive dynamic range. It is encoded after the sign bit and has a variable size, determined by the run-length of equal bits terminated by an opposite bit. As in FP, the exponent field `exp` is fixed in length and defined by es, but can be defined to have zero length. The mantissa (fraction) `mant` occupies the remaining bits of the number if any space is left.

The encoded value of a posit is interpreted as given in Eq. (2.3). The main difference is the additional scaling using the exponent size es and the regime value `k`, which is obtained by counting the run-length m of bits equal to the first bit `r` of the regime, and applying Eq. (2.4). Some example encodings are listed in Table 2.5. The exponent is unbiased, as centering around ± 1.0 is achieved using the regime. In case the regime starts displacing the exponent, all lower-order bits are assumed to be `0`. As in FP, the mantissa has a hidden bit, which is always `1`; thus, no subnormals exist.

Table 2.6: Posit encodings of special cases.

Sign	Remainig Bits	Encoded Value
0	$(\texttt{00}\ldots\texttt{0})_2$	0
1	$(\texttt{00}\ldots\texttt{0})_2$	$\pm\infty$ / NaR

$$(-1)^{\mathtt{s}} \times \mathrm{useed}^{\mathtt{k}} \times 2^{(\mathtt{exp})_2} \times 1.(\mathtt{mant})_2 \qquad \mathrm{useed} = 2^{2^{es}} \tag{2.3}$$

$$k = \begin{cases} -m & \text{if } \mathtt{r} = 0 \\ m - 1 & \text{if } \mathtt{r} = 1 \end{cases} \tag{2.4}$$

Special Cases

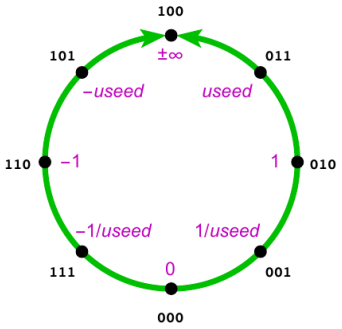
Table 2.6 shows the only two special encodings for posits, and there are neither signed zeroes nor infinities. In contrast to FP, exceptional computations like division by zero do not trigger an exception but will simply result in "$\pm\infty$," which is considered *not a real (NaR)*.

While the sign bit is used in the same way as for FP, there is one caveat. If the sign is negative, the two's complement of the entire binary representation is taken and used to decode the regime, exponent, and fraction fields. As a result, exciting properties emerge, as outlined further in this section.

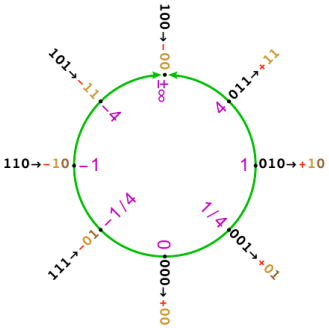
Rounding Modes

Posit knows only one rounding mode, *round to nearest, ties to even*, which also serves as the default in IEEE 754. As posit values can be encoded without any mantissa bits, rounding can also occur in exponent bits. The midpoint of two such bit patterns corresponds to the geometric mean of the choices.

Posits are never rounded down to zero nor rounded up to infinity, as this would destroy all information about a result or introduce infinite error, respectively.



(a) Projective reals using 3 bit.



(b) Applied posit encoding for $es = 1$.

Figure 2.14: A binary string of 3 bit projected onto so-called projective reals. (a) shows the general assignment of ± 1, 0, *useed*, and NaR. (b) includes the numerical interpretations for $es = 1$. [78]

Numerical Properties

Posits can be interpreted as binary integers mapped to so-called *projective reals*, as shown in Section 2.5.1. This mapping is highly symmetrical, and reflection along the vertical and horizontal axes points at the negated and reciprocal values, respectively. Notably, every representable number in posit has an exactly representable reciprocal. Consequently, the two special encodings of zero and $\pm\infty$/NaR are reciprocals of each other.

2.5.2 Considerations for Posit in Hardware

Due to the symmetry of the "projective real" mapping, certain operations are elementary to achieve using posit. Reciprocals are obtained by flipping the sign bit of the representation. Simultaneously, the negation of a number corresponds to taking the two's complement of the representation, both of which can be performed using an integer adder.

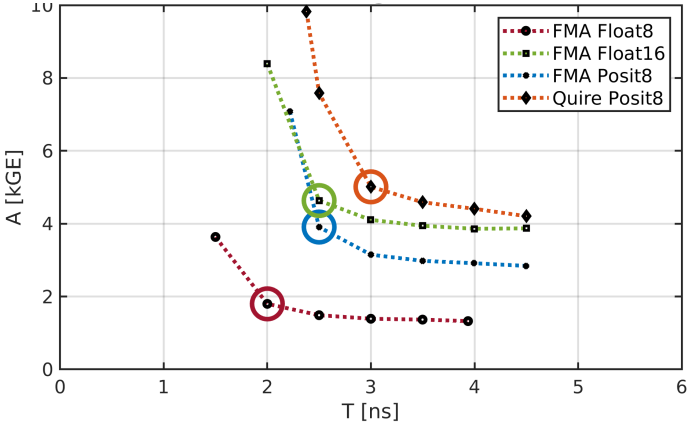


Figure 2.15: AT plot of 8-bit and 16-bit FP FMA, 8-bit posit FMA and 8-bit posit with 32-bit quire accumulation.

Comparison with IEEE 754

Proposed posit configurations follow the power-of-two format sizes used for FP. While posit can be configured to cover a similar dynamic range of regular FP, its creators suggest reducing the number of exponent bits for better precision, deeming the extensive dynamic range of *binary64* already too large for practical needs [78]. We focus on minimal configurations to evaluate posit's potential to replace FP for TP purposes.

While IEEE 754 mandates the FMA operation as a means of combatting accumulation of rounding error, authors of posit propose the use of a wide fixed-point accumulator called *quire* instead [78]. This concept, also known as a Kulisch-Accumulator [79], could also be applied to FP accumulations.

We compare a 8-bit posit FMA unit and an 8-bit posit unit containing 32-bit quire functionality with IEEE 754 FMA implementations in a 65 nm technology. The trade-offs in area and timing obtained from synthesis are shown in Fig. 2.15. Notably, the 8-bit posit FMA requires 2× the area of the 8-bit FP FMA, with a longest path similiar to a 16-bit FMA. The stark increase in area compared to 8-bit FP stems

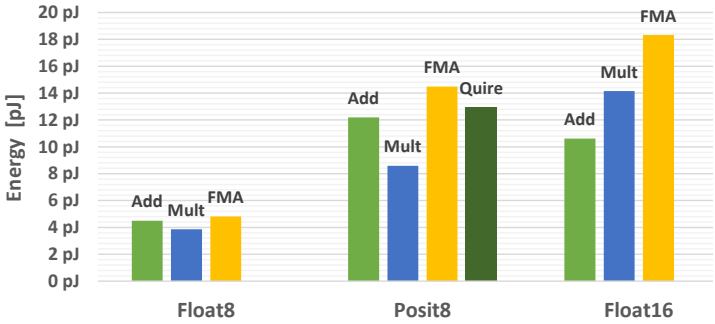


Figure 2.16: Energy consumption per operation for implementations circled in Fig. 2.15.

due to accounting for the variable lengths of the regime and mantissa in posit, requiring an over-provisioned datapath. Furthermore, the area and critical path are inflated by 20% when adding the quire functionality to the posit implementation due to the quire register's impact.

Power measurements from simulations on a post-place-and-route database are shown in Fig. 2.16. We note that operations on 8-bit posit are over 200% more expensive than 8-bit FP. Quire operation slightly reduces energy cost compared to FMA, as fixed-point accumulation has much lower complexity than posit addition despite the higher bit-width.

2.6 Summary and Conclusion

In this chapter we have explored the possiblities to introduce TP into FP-based computing. Notably:

- We designed a software library, *FlexFloat*, in order to perform explorations of arbitrary FP types using precision tuning frameworks.
- We introduced an extended FP type system with two additional FP formats, *binary16alt* and *binary8*, which are closely based on

IEEE 754 principles.

- We designed a first TP-capable FPU built from priprietary FP blocks as hardware support to enable TP computing on ULP embedded platforms.

- Experimental results show that our approach is effective in reducing energy consumption by leveraging the knobs provided by the extended FP type system and thanks to vectorization support. The energy consumption is reduced on average by 18%, and up to 30% for specific applications. Simultaneously, execution time is decreased by 12%, and memory accesses are reduced by 27%.

- We discussed two alternative number formats, *unum* and *posits* that try to expand upon concepts from standard FP formats from a hardware perspective to gauge their potential for use in TP implementations.

- We show synthesis experiments to compare unum arithmetics with their IEEE 754 counterparts. We conclude that it must be carefully analyzed whether memory accesses are expensive enough for the significant (de)compression overhead linked to variable-width number formats to pay off. While unum formats could provide a moderate memory footprint advantage (7%) to the standard IEEE 754 binary32 and broader range than binary16, this comes at a significant increase in datapath complexity and requires special care in avoiding aggressive unification to prevent error blow-up.

- We find that using posit hardware implementations introduces high additional costs in both area and timing. Area, longest path, and energy increase by over 100%, 25%, and 100%, respectively, when comparing an 8-bit posit FMA with an FP FMA of the same width. The reason is the need to cover the regime's worst-case sizes and the mantissa, significantly inflating the datapath.

We conclude that extending the FP type system with sub-32-bit formats is a viable approach to improving performance and energy efficiency through TP computing methods. For the time being, we shall

forego alternatives to traditional FP formats due to large overheads in the circuit area, speed, and energy costs, especially for small formats.

To make further explorations more accessible, we require a more flexible and configurable TP-FPU with a design focus on energy-proportional FP computing. Besides a dedicated hardware unit, exploitation of TP techniques will require integration of a processing system and support at the ISA and compiler levels of the computing stack.

Chapter 3

An Open-Source Transprecision FPU

3.1 Introduction

In the previous chapter, we have shown the promising potential of TP computing in terms of performance and energy efficiency improvements of applications. We achieved this by extending the FP type system with custom types – collectively referred to as "SmallFloat" – which allow TP-aware applications to fine-tune their precision requirements during computation, enabling faster and more efficient execution. However, such gains are only attainable when the hardware performing such computations is equally fine-tunable and offers energy-proportional execution.

Traditionally, FP precision modulation has been limited to only two standard formats in CPUs and GPUs. However, recent research in various domains, but strongly driven by machine learning algorithms and applications, shows that new architectures with extreme TP flexibility in terms of formats supported are needed for FP computation. Thus, in this second chapter, 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.

In order to leverage such TP-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 [26] is an open-source ISA which natively supports computation on the common "double" and "float" formats. Furthermore, the ISA explicitly allows non-standard extensions where architects can add new instructions. Lately, RISC-V has gained traction in industry and academia due to its open and extensible nature with growing support from hardware and software projects.

The main contributions of this chapter are:

1. The design of *FPnew*, 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, multi-format 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 (Section 3.2).

2. Extensions to the RISC-V ISA to support TP FP operations on existing and new FP formats. Scalar operations are supported by a set of ISA extensions corresponding to the new formats, moreover the complementary "Xfvec" extension defines SIMD sub-word parallelism for all operations in the scalar FP extensions. Operations that do not fit these classes have been added in an additional extension set, called "Xfaux", which include advanced arithmetic (average, dot-product) and expanding operations (that take SmallFloat type operands but return a single-precision result). We furthermore discuss the modifications to the standard RISC-V GCC compiler, which are required to enable the adoption of SmallFloat types (Section 3.3).

3. A case study for showcasing the programming possibilities of our SmallFloat ISA and compiler extensions. We achieve a significant speedup compared to an FP32 baseline without sacrificing any precision in the result (Section 3.4).

The rest of this chapter is organized as follows: Section 3.2 describes in-depth the requirements and architecture of the proposed TP-FPU. Section 3.3 outlines our work on TP ISA extensions and implementation into the GCC compiler. Section 3.4 contains a TP case study performed on a RISC-V core system. The last sections of this chapter contrast our work with related works and provide a concluding summary.

3.2 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 and domain-specific accelerators.

3.2.1 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 was also a guiding principle for the design. We present some key requirements that we considered during the design of the unit:

FP Format Encoding

As outlined in Chapters 1 and 2, 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. 3.1.

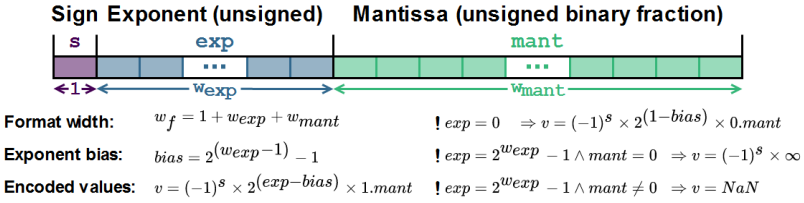


Figure 3.1: FP format encoding as specified by IEEE 754 and its interpretation.

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 [80] has since been added to IEEE 754-2008 and is nowadays ubiquitously used for 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. Non-standard multi-format arithmetic is also becoming more common, such as performing the multiplication and accumulation in an FMA using two different formats in tensor accelerators [23, 24].

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 TP applications (see Chapter 4) and require support for vectored data. The architecture of the TP-FPU thus must be able to support this kind of vectorization to support multi-format operations on SIMD vectors.

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 (see Chapter 4).

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.

Design Tool Flow

The TP-FPU is written in the industry-standard SystemVerilog HDL (IEEE 1800), supported by every commonly used design flow, to ensure interoperability and accessibility. Novel hardware construction languages such as Chisel [81] were not considered as the lack of native tool support considerably adds complexity w.r.t. standard design and verification flows.

Target-specific synthesis flows (e.g. for application-specific integrated circuit (ASIC) or FPGA technologies) differ in available optimized blocks, favoring inferrable 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 to enable automatic optimizations, including clock-gating and pipelining, wherever possible.

3.2.2 Building Blocks

In the following we present a general architectural description of our TP-FPU, shown in Fig. 3.2. Concrete configurations chosen for the integration into processor cores and the implementation in silicon are discussed in Chapters 4 and 5.

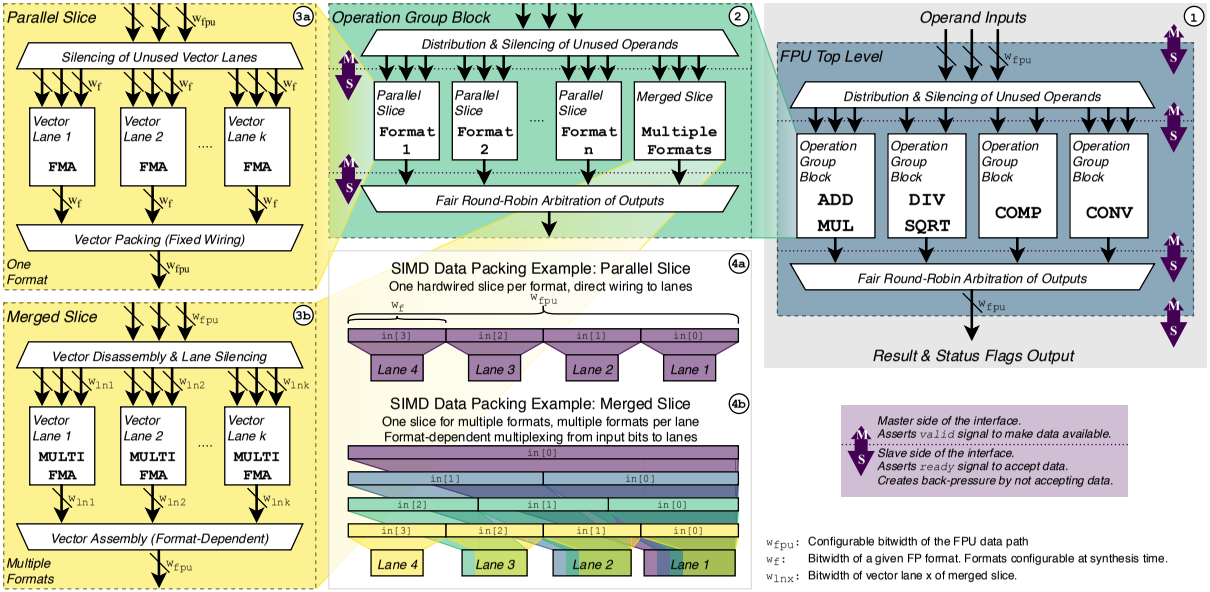


Figure 3.2: Datapath block diagram of the TP-FPU with its underlying levels of hierarchy. It supports multiple FP formats, and the datapath width is configurable.

FPU Top Level

At the top level of the TP-FPU (see Fig. 3.2-1), 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 input and output operands' width is parametric and will henceforth be referred to as the *unit width* ($\mathtt{w}_{fpu}$).

Input operands are routed towards one of four operation group blocks, each dedicated to a class of instructions. Arbiters feed the operation group outputs towards 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.

Operation Group Blocks

The four operation group blocks making up the TP-FPU are as follows:

- *ADDMUL*: addition, multiplication and FMA
- *DIVSQRT*: division and square root
- *COMP*: comparisons and bit manipulations
- *CONV*: conversions among FP formats, to/from integers

Each of these blocks forms an independent datapath for operations to flow through (see Fig. 3.2-2). When multiple FP formats are present in the unit, the blocks can host several slices that are either implemented as format-specific (parallel) or multi-format (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.

Format Slices

Format slices host the functional units which perform the operations in which the block is specialized.

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

In the parallel case (see Fig. 3.2-3a), the lanes are duplicate instances of the same functional unit, and the number and width of lanes are determined as given in Eq. (3.1).

$$\begin{aligned} \mathtt{k}_{parallel} &= \left\lfloor \frac{\mathtt{w}_{fpu}}{\mathtt{w}_f} \right\rfloor \\ \mathtt{w}_{lane,parallel} &= \mathtt{w}_f \end{aligned} \tag{3.1}$$

In the merged case (see Fig. 3.2-3b), the total number of lanes is determined by the smallest supported format; the width of each lane depends on the containing formats. Individual lanes within merged slices have the peculiar property of differing in bit width (see Eq. (3.2)), and each lane needs support for a different set of formats (see Fig. 3.2-4b).

$$\begin{aligned} \mathtt{k}_{merged} &= \left\lfloor \frac{\mathtt{w}_{fpu}}{\min\limits_{\forall format \in slice} \mathtt{w}_f} \right\rfloor \\ \mathtt{w}_{lane[i],merged} &= \max_{\forall format \in slice} \mathtt{w}_f \big|_{\mathtt{w}_f \leq \frac{\mathtt{w}_{fpu}}{i}} \end{aligned} \tag{3.2}$$

As the current operation is either scalar or vectorized, 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 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.

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 [80, 57], providing results within 1/2 ulp. Subnormals and all rounding modes mandated by IEEE 754 are implemented. Due to the single-path architecture employed, support for subnormal numbers does not significantly impact the data path; the internal exponents are widened by one bit to treat subnormals as normal values. The default RNE rounding mode is the most complex to implement, adding a carry chain to the datapath. A multi-format 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 divider. A radix-2 non-restoring division iteration is unrolled three times. Thus, 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 TP computing. The COMP block's operational unit consists of a comparator with additional selection and bit manipulation logic to perform comparisons, sign manipulation, and FP classification. Lastly, the CONV block features multi-format casting units that can convert between any two FP formats, from integer to FP formats, and from FP to integer formats.

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. A simple synchronous *valid-ready* handshaking protocol is used within

the internal hierarchies and on the unit's outside interface to stall the internal pipelines.

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 is present in the pipeline.

3.2.3 Configuration, Parametrization, and Usage

The TP-FPU offers great 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. 3.1) and support all standard rounding modes. Operations on defined formats can be implemented as either a parallel or a merged slice or disabled entirely by using parameters. Furthermore, SIMD vectors can be enabled globally for all formats with $\mathtt{w}_f \leq \mathtt{w}_{fpu}/2$, and $\mathtt{w}_{fpu}$ 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.

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. All implementations in this thesis were parametrized from the default package. As the unit is open-source, designers are free to extract or replace functional units or add other operation groups of their own.

[1] In order 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.

3.3 Enabling FPnew in the RISC-V ISA

To enable TP computing and fully leverage our TP-FPU in general-purpose processors, we require support for non-standard FP formats and instructions within an ISA. The open-source RISC-V ISA offers ample opportunities for extensions with custom operations, thus we have extended the ISA with special instructions. Our RISC-V SmallFloat extensions comprise a set of non-standard extensions to the RISC-V ISA which enable operations on *sub-32*-bit FP formats.

As a base, scalar extensions are provided that match the "F" and "D" standard FP extensions in terms of available operations. Furthermore, optional vectorial extensions are specified which make use of SIMD sub-word parallelism by packing multiple *sub-32*-bit elements into one FP register. Lastly, we add an optional extension for auxiliary operations, which are not available in the standard FP extensions.

3.3.1 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 the sub-32 bit formats proposed in Chapter 2. The available FP formats in our implementations are:

- **binary64** (FP64): IEEE 754 double-precision (11, 52)
- **binary32** (FP32): IEEE 754 single-precision (8, 23)
- **binary16** (FP16): IEEE 754 half-precision (5, 10)
- **binary16alt** (FP16alt): custom half-precision (8, 7)
- **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 subnormals, NaN and the NaN-boxing of narrow values inside wide FP registers.

Table 3.1: Summary of available FP operations with SmallFloat extensions to the RISC-V ISA. Instruction examples listed for operations on *binary16*.

Operation Type	Instruction Example	Semantics	ISA Extension
Scalar Arithmetic	`fadd.h`	`rd = rs1 + rs2`	Xf16
Scalar Conversions	`fcvt.h.s`	`rd = (fp32)rs1`	Xf16
Vector Arithmetic	`vfadd.h`	`rd[] = rs1[] + rs2[]`	Xfvec
Vector Conversions	`vfcvt.x.h`	`rd[] = (int16v)rs1[]`	Xfvec
Cast-and-Pack	`vfcpk.h.s`	`rd[] = {(fp16)rs1, (fp16)rs2}`	Xfvec
Expanding	`fmacex.s.h`	`rd = (fp32)(rs1 * rs2 + rd)`	Xfaux
Other	`vfdopex.h`	`rd[] = (fp32)dotp(rs1[],rs2[])`	Xfaux

3.3.2 Operations

The SmallFloat extensions have no collisions with typical RISC-V implementations and can thus be included in any implementation without loss of RISC-V standard compliance. Table 3.1 gives a summary of available operation types with SmallFloat extensions active. We exemplify a representative instruction for each operation type, and we specify its semantics with a C-like pseudocode notation. For brevity, we have limited the list to *binary16* instructions, however operations related to the other types are analogously defined by changing the opcode suffixes (`.b` is used for *binary8* and `.ah` for *binary16alt*). More details are provided in the SmallFloat ISA manual [82].

The new operations can be roughly grouped into three parts, namely *scalar*, *vectorial*, and *auxiliary* extensions.

3.3.3 Scalar Extensions

The scalar SmallFloat extensions provide support for the *binary16* and custom *binary16alt* FP formats (both 16-bit wide), as well as the custom *binary8* format (8-bit wide). Each of these formats is contained in its own respective ISA extension "Xf16", "Xf16alt" and "Xf8", all of which are a repurposed version of the single-precision "F" standard RISC-V extension. An example of scalar operation is `fadd.h` in Table 3.1.

The set of operations on SmallFloat formats is equivalent to their single-precision counterparts; thus, their encoding closely matches the standard FP operations. A previously unused configuration of the FP format field in the instruction word has been chosen to signify 16-bit FP types, while the patterns representing quad-precision FP operations (128-bit) have been repurposed to now denote the 8-bit FP type. While this poses a collision with the "Q" RISC-V standard extension, it is highly unlikely embedded implementations targeted towards low precision FP will also implement 128-bit floats. The two 16-bit formats (*binary16* and *binary16alt*) are differentiated using unused states of the rounding-mode fields in the instruction word.

Table 3.2: *FLEN*, given by the widest supported floating-point format (rows) defines the vector dimension for all narrower supported formats (columns).

		Vector length `n` if supported			
		F	Xf16	Xf16alt	Xf8
FLEN	64	2	4	4	8
	32	×	2	2	4
	16	×	×	×	2

3.3.4 Vectorial Extension

The vectorial SmallFloat extension "Xfvec" is encoded in its own encoding space, utilizing a previously unused prefix in the RISC-V "OP" opcode. This extension defines SIMD sub-word parallelism for all operations in the scalar FP extensions, such as the `vfadd.h` operation in Table 3.1. If "Xfvec" is supported, vectorial FP operations are added for all supported FP formats that are narrower than the width of the FP register file (*FLEN*). Thus, the vectorial extension depends on some or all the following: the "D" and "F" standard extensions as well as the "Xf16" and "Xf16alt" SmallFloat extensions, which also defines the format-specific vector length `n` as shown in Table 3.2. Notably, if the "D" standard extension is enabled (and hence *FLEN*=64), vectorial operations on single-precision FP data are also enabled with "Xfvec".

In addition to computational and comparison operations already present in the scalar extensions, "Xfvec" adds vector-specific conversion operations. Vectorial FP - Integer conversions will convert a vector of packed FP values to a packed vector of integer values of the same width (the `vfcvt.x.h` operation in Table 3.1), and vice versa. Vectorial FP - FP conversions pose the challenge of mismatched vector lengths for different FP formats. Thus instructions are also provided to convert half of the more populous FP vector into a complete vector of a wider format and vice versa. For example, the lower or upper two entries of a *binary8* vector (containing four entries) can be converted to a vector of two *binary16*[2].

[2]Assuming *FLEN*=32 here. Analogous behavior for *FLEN*=64, where four entries are converted concurrently.

Lastly, cast-and-pack instructions were added that convert two scalar single- or double-precision (if supported) FP operands and insert them into two adjacent entries of a packed vector in the destination register. These operations were added in addition to the vectorial conversion instructions since "convert scalars and assemble vectors" operations emerged as the main bottleneck of TP computing (see Chapter 4). In Table 3.1 the `vfcpk.h.s` operation converts two single-precision values to half-precision and insert them into a half-precision vector.

3.3.5 Auxiliary Operations Extension

Finally, additional operations not included in the standard FP extensions such as averaging two numbers or dot-product of vectors have been added in their own extension set "Xfaux". These operations have been encoded in unused regions of the scalar or vectorial extensions, depending on the operation's nature.

"Xfaux" most notably includes expanding operations that take SmallFloat type operands but return a single-precision result, making explicit conversion instruction cycles unnecessary in many applications where the dynamic range of operands increases over the execution. These instructions include expanding multiplication, multiply-accumulate of SmallFloats on a *binary32* accumulator, as well as expanding dot-products. A typical example of expanding operation is the `fmacex.s.h` operation in Table 3.1, which performs a multiplication between two half-precision operands and accumulates the result into a single-precision register.

3.3.6 Encoding

The encoding of these new instructions was implemented as RISC-V brown-field non-standard ISA extensions [82]. 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. 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, similarly to the encoding of DSP extensions realized for the RI5CY core [50]. Auxiliary instructions are encoded either in unused FP or integer operation opcode space, depending on whether they operate on scalars or vectors.

3.3.7 Compiler Support

This section describes the main modifications to the baseline GCC RISC-V compiler, which are required to support SmallFloat types.

As a preliminary step, we have extended the *real* interface – used as an internal representation for all the FP types supported by the programming language – with callback functions that enable to convert data from the internal format to the SmallFloat one (and vice versa). Accordingly, we have extended the RISC-V back-end to include new *machine modes* which describe the size and representation of the SmallFloat formats (scalar types and their vectorial counterparts).

We have added a new set of rules to the machine description, which provides all the information required to perform the lowering (i) from the middle-end (*gimple*) to the back-end (RTL) representation, and (ii) from RTL expressions to assembly opcodes (see Table 3.1).

The coexistence of two FP formats with the same size (i.e., *binary16* and *binary16alt*) has required modifications to the pass performing the RTL lowering. It has been necessary to correlate *gimple* expressions involving these types to the related back-end rules to guarantee a correct behavior, in particular, to enable casts between *binary16* and *binary16alt* formats.

GCC provides a mechanism to access at the back-end level a set of language-specific front-end hooks. We have used this mechanism to extend the standard C/C++ type system to provide access to the SmallFloat types by introducing a new set of keywords (`float8`, `float16` and `float16alt`). We have also augmented the set of implicit conversion rules to integrate the new format into the C/C++ type system fully. Each operand is automatically converted to the wider one in an expression involving different FP types and no explicit casts. In case of expressions using both `float16` and `float16alt`, `float16` is prioritized.

GCC includes an automatic vectorization pass that operates on the middle-end intermediate representation [83]. This pass analyzes loops to replace the scalar operations with the vectorial ones reducing the loop's trip-count by the vectorization factor. In our work we have extended the GCC auto-vectorizer to enable the adoption of SmallFloat types.

- We have modified the RTL lowering pass (i) to distinguish between `float16` and `float16alt` targets when performing a vector unpacking operation from a `float8` vector and (ii) to support vectorial casts between `float16` and `float16alt`.

- We have introduced a virtual vector type (i.e. a one-element vector) for single-precision operands. We enable the support to the cast-and-pack operations, which involve a SmallFloat vector as destination and single-precision registers as sources. Simultaneously, introducing a vectorial machine mode prevents the compiler from aborting the vectorization analysis for loops that include single-precision arithmetic at an early stage.

- We have tuned the cost model provided to the auto-vectorizer by the RISC-V back-end, intending to allow the presence of single-precision operations when the following ones can be vectorized. In this case, cast-and-pack operations are inserted after the single-precision operations, which are unrolled by the vectorization factor[3].

In addition to the compiler's automatic vectorization techniques, programmers can manually vectorize their code. GCC supports the definition of vectorial data types by means of a `typedef` declaration coupled with a `vector_size` attribute; standard arithmetic operations using these types are automatically lowered into their vectorial counterpart. Moreover we have provided a set of compiler intrinsics which provide access to the operations that are included in the "Xfvec" and "Xfaux" ISA extensions (see Table 3.1). An example of manual vectorization using vectorial types and intrinsics is explained in [35].

[3]GCC already considered vectorization patterns involving multiple vectorial types with different widths through unrolling and vectorial casts, but cast-and-pack semantic was not supported.

GCC handles widening variants of addition, multiplication, and multiply-and-add operations, whose operands have a smaller size than results. Since these operations were supported only for integer data types, we have extended the middle-end pass that generates intermediate code to consider widening expression with SmallFloat operands. Adopting this approach, GCC can use the expanding operations of the "Xfaux" extension without ad-hoc code modifications in the target program. Also, in this case, we have provided intrinsics for advanced programmers who perform fine code tuning.

3.4 Programming of Transprecision Application Kernels

To visualize some challenges and benefits of TP applications, we showcase a simple multi-format application kernel running on the TP-enabled RI5CY core (see Chapter 4). Furthermore, we touch on the considerations to make when programming for TP-enabled platforms.

3.4.1 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.

Approach

Figure 3.3 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 TP ISA extensions to obtain the assembly in Fig. 3.4 as follows:

- Fig. 3.4 a) is the FP16-only workload in Fig. 3.3 a) requiring an ideal 3 instructions per input pair.
- Fig. 3.4 b) performs all operations on FP32 to achieve the most precise results but requires casts in a total of 5 instructions.

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]);
```

Figure 3.3: 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 TP instructions.

Table 3.3: Metrics corresponding to assembly from Fig. 3.4

	# Bits correct	**Rel. Error of Result vs.** Exact	Exact FP16*	**Rel. Energy** Core	System
Exact Result	37	0.0			
Cast to FP16	12	1.9×10^{-4}	0.0		
Result 3.4 **a)**	9	2.7×10^{-3}	2.9×10^{-3}	0.60	0.63
Result 3.4 **b)**	22	2.0×10^{-7}	0.0	1.00	1.00
Result 3.4 **c)**	19	1.6×10^{-6}	0.0	1.16	1.03
Result 3.4 **d)**	19	1.6×10^{-6}	0.0	0.97	0.75
Result 3.4 **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

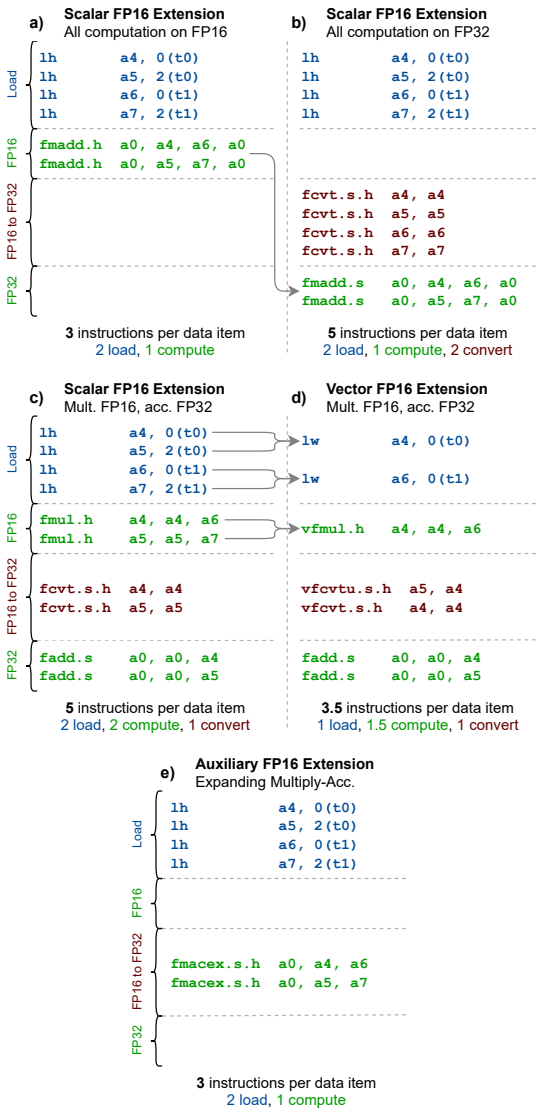


Figure 3.4: RISC-V assembly implementations of two iterations of the loop body in Fig. 3.3 (grey).

- Fig. 3.4 c) tries to save energy by performing the multiplication in FP16 to replace the FP32 FMA with additions.
- Fig. 3.4 d) accelerates the FP16 portion of the previous code by using SIMD in 3.5 instructions.
- Fig. 3.4 e) makes use of expanding multi-format FMA instructions to combine computation and conversion in 3 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.

Performance and Energy Results

We collect the final result accuracy and energy use of these programs in Table 3.3. Energy results have been obtained from a post-layout simulation of the RI5CY + TP-FPU design presented in Section 4.3.2.

The accuracy of the result from Fig. 3.4a) 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. 3.4 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.

Figure 3.4 c) suffers from decreased accuracy (19 bit) and even requires 16% more core energy (+3% system energy) w.r.t. 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. 3.4 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. 3.4 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% compared to the FP32 case. These results highlight the energy savings potential of TP computing when paired with flexible hardware implementations.

3.4.2 Compiler Support

We make the low-level TP instructions available as a set of compiler intrinsic functions that allow full use of the TP hardware by the compiler and 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 by using GCC vector extensions. In fact, the compiler can generate programs such as in Fig. 3.4 d) from the code in Fig. 3.3 b).

However, operations such as the FMA and optimized access and conversion patterns using SIMD vectors often do not cleanly map to high-level programming language operator semantics. It is prevalent that performance-optimized FP code requires low-level manual tuning to make full use of the available hardware, even in non-TP code. We can and should make use of the non-inferrable operations such as cast-and-pack or expanding FMA through calls to intrinsics, as seen in Fig. 3.3 c), which can produce assembly Fig. 3.4 e). The benefits and limits of the compiler-based approach are further investigated in [35].

3.5 Related Work

3.5.1 SIMD and TP 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 non-standard 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 using formats within an instruction, enabling very consistent handling of multi-format (expanding, shrinking) operations that always operate on entire registers. This 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. Multi-format 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 bit 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 data handling without addressing individual register portions. It supports FP16, FP32, FP64, and FP128 and 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.

3.5.2 Open-Source Configurable FPU Blocks

Most open-source FPU designs are implementing a fixed implementation in a specific format, targeting a specific system or technology [84, 85]; however, there are some notable configurable works available.

For example, FloPoCo [86] 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 trade-off for a slightly higher dynamic range present in FloPoCo FP formats. The FMA operation is not available.

Hardfloat [87] 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 non-standard 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.

3.5.3 FPUs for RISC-V

Some vagueness exists in IEEE 754 concerning *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. 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 to avoid at least the hardware-related issues. 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 [88, 89], are based on Hardfloat components in specific configurations for RISC-V.

Kaiser et al. [90] 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 towards the end of this section.

3.5.4 Novel Arithmetics / TP FP Accelerators

Non-standard 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 [23] and NVIDIA's Tensor Cores [24] 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 we presented is directly actionable, as the format has been popularized for DL applications with negligible accuracy loss [91].

Dedicated accelerators geared towards neural network training such

as NTX [92], for example, employ non-standard 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 for example unums [77], Posits [93], and logarithmic nubmer systems (LNSs). Unum-based hardware implementations were proposed in [40, 75]. Fast multiplication and transcendental functions were implemented into a RISC-V core in [94]. Nannarelli [95] describes the design of an FPU based on the Tunable floating-point (TFP) format, which supports a variable number of bits for mantissa (from 4 to 24) and exponent (from 5 to 8). However, this solution does not support vectorization, which is a crucial enabler for energy efficiency. Other works have proposed dedicated FP units for reduced-precision [68, 69] and variable-precision [67] arithmetic. In many cases, these formats' adoption has generated some criticisms due to the high cost of hardware implementations and the significant effort for code refactoring.

While the focus of our TP-FPU is to provide IEEE 754-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 similar to a merged functional unit with many 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.

3.5.5 Multi-Mode Arithmetic Blocks

To our knowledge, no fully-featured TP-FPUs with support for multiple formats have been published so far. However, multi-mode FMA architectures have been proposed recently, usually featuring computations on two or three FP formats [96, 97, 98, 67], or a combination of integer and FP support [100]. Table 3.4 compares the proposed architectures with our silicon implementation (see Chapter 5) under

Table 3.4: FMA comparison of this work (complete FPU) with other standalone [multi-mode] 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.* [90], 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.* [96], 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.* [97], 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.* [98], 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.* [67], 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.* [99], 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

nominal conditions. Note 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.

The RISC-V-compatible FMA unit from [90] 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 [96, 97, 98] 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 better with smaller formats – reaching 16.6× lower energy for operations on FP8 w.r.t. FP64 (width reduction of 8×). 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 trade-off by using the voltage scaling knob (see Chapter 5).

The FMA implementation in [67] 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. The intermediate formats used in this unit do not correspond to any standard IEEE 754 formats.

FPMax [99] 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.

3.5.6 Other uses of our TP-FPU

The open-source nature of FPnew and 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 flows. FPnew or subcomponents of it have found use under the hood of some recent works, as will be discussed in Chapters 4 and 5.

3.6 Summary and Conclusion

An open-source, fully configurable TP-FPU is the key enabler of TP computing in hardware. However, the software must also be made capable of leveraging the tunability of such hardware units. Our RISC-V ISA extensions bridge this gap to facilitate TP computing in practice. Specifically, the findings in this chapter are:

- We have presented FPnew, a configurable open-source transprecision floating-point unit capable of supporting arbitrary FP formats. It offers scalar and SIMD-vectorized variants of FP arithmetic operations. Furthermore, it offers efficient casting and packing operations with high energy efficiency and proportionality. Notably, our architecture trades off excess circuit area for improved energy efficiency. Our design achieves better energy efficiency scaling than other multi-mode FMA designs thanks to the parallel datapaths approach taken in our architecture.
- We have introduced a set of extensions for the RISC-V ISA aimed at supporting a set of *sub-32*-bit FP formats. Adopting these formats has been proven to be highly beneficial in both HPC and embedded systems domains. We present both a complete specification for the proposed SmallFloat extensions and the compiler support design and implementation.
- In our case study, we showcase the programming possibilities of our SmallFloat ISA and compiler extensions. We achieve FP32 precision without incurring any performance overhead compared to an optimal scalar FP16 baseline, reducing system energy by 34% w.r.t. the FP32 implementation.

- Thanks to our efforts' open-source nature, FPnew can be utilized in many different application scenarios. In fact, it has found use both in embedded IoT applications and high-performance vector processing accelerators (see Chapters 4 and 5).

We conclude that we have laid a robust foundation for implementing TP computing into a vast range of systems. As such, we will target enabling TP in both embedded platforms and high-performance computing applications.

Chapter 4

Transprecision FP in the Embedded Domain

4.1 Introduction

In Chapter 2, we have laid the foundation for TP computing in ULP systems by showing that precision modulation using SmallFloat formats can bring increased performance and energy savings. Combined with the SmallFloat ISA extensions from Chapter 3, we now have the tools to assemble a TP-capable RISC-V system. We completed the implementation of the TP-FPU prototype into the PULPino ULP microcontroller that was used in Chapter 2 and evaluated the potential of TP computing on the processor level.

We also implemented FPnew, the full TP-FPU developed in the previous chapter, into the most up-to-date version of the SoC based on the RI5CY core [50], called PULPissimo. We used simulations from this implementation to perform the TP programming case study found in the previous chapter.

Furthermore, we have built a full-blown multi-core SoC based on a cluster of RI5CY cores with FPnew targeting high-end ULP signal processing tasks. The implementation as an FPGA soft-IP allows for extensive architectural exploration considering the sharing of TP-FPUs

amongst several cores.

The main contributions of this chapter are:

- A full architecture for ultra-low-power TP computing, based on the open-source PULPino SoC [27] with a TP FPU integrated into the RI5CY processor core [50]. We support the "SmallFloat" formats defined in Chapter 2 as well as SIMD, further increasing the performance and energy efficiency of the SoC. We characterize the energy consumption of the SmallFloat instructions on the post-place & route (P&R) implementation of the proposed TP SoC and evaluate the execution of a set of signal processing benchmarks to assess the energy saving when tuning the precision of the operations for pre-defined accuracy targets. The proposed TP SoC significantly improves system performance and energy efficiency over a traditional 32-bit FP SoC on the analyzed applications (Section 4.2).

- Integration of FPnew (see Chapter 3) into RI5CY [50], an embedded 32-bit RISC-V processor core and implementation into the PULPissimo SoC [101]. We perform a full P&R layout in order to perform the measurements and simulations presented in the previous chapter (Section 4.3).

- The architectural design of a multi-core TP cluster and its software infrastructure. A dedicated design for the FPU interconnect enables multiple policies for FPU sharing, with the possibility to guarantee different trade-offs. Using FPGA emulation, we explore the design space considering different architectural configurations of the TP cluster: the number of cores, the number of FPUs and the related sharing factor, the number of pipeline stages in the FPUs (Section 4.4).

- A vertical exploration to identify the most efficient solutions optimizing non-functional requirements (i.e., operating frequency, power, and area) using post-P&R models in 22nm FDX technology. We compare the most efficient configurations deriving from the design space exploration with SoA solutions, considering a broader scenario that includes high-performance and embedded computing domains (Section 4.4).

- Overview of an evolution of the TP cluster architecture described above, which is currently being commercialized by second parties (Section 4.5).

The remainder of this chapter is organized as follows: Section 4.2 contains the implementation and evaluation of the prototype TP-FPU into PULPino. Section 4.3 outlines the FPnew unit's implementation into the RI5CY core and the implementation into PULPissimo. Section 4.4 details the architecture, implementation and evaluation of a multi-core transprecision SoC. Section 4.5 introduces a notable system based on the TP cluster architecture. The last section provides a summary and conclusion of this chapter.

4.2 Embedded SoC for Transprecision

An increasing amount of deeply embedded applications such as monitoring and processing of vital signs, building health profiles, and audio processing algorithms require extreme energy efficiency and complex, highly dynamic numerical computations involving double-precision (*binary64*) or single-precision (*binary32*) FP operations, defined by the IEEE 754 standard. In many of these FP-intensive applications, the execution of FP operations and the related memory transfers emerge as the main bottleneck for energy efficiency consuming up to 50% of the overall system power (see Chapter 2). The most traditional approach to optimizing the energy consumption of applications requiring high dynamic range and precision in power-constrained platforms is to shift to fixed-point implementations and adjust the dynamics and precision of operands according to the processing chain's requirements. However, this approach is often highly intrusive, requiring an in-depth understanding of the target algorithms. To trade energy for dynamic range and precision of FP operations, we have introduced the suite of SmallFloat formats in Chapter 2.

In this section, we present a full SoC architecture for ultra--low-power TP computing. The proposed hardware architecture extends the PULPino open source SoC [102] with a TP FPU integrated into the RI5CY processor core [50]. We have demonstrated that significant energy savings can be achieved leveraging sub-32-bit

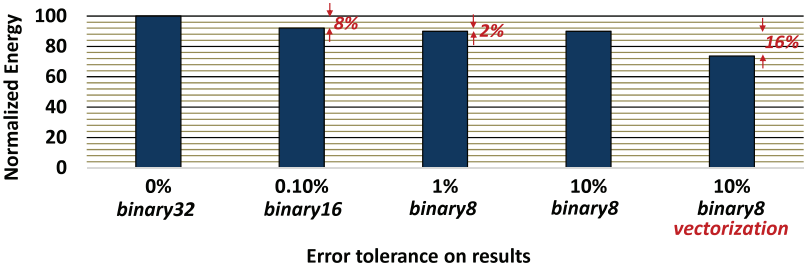


Figure 4.1: Energy consumption of the KNN application for three precision requirements, normalized to *binary32* baseline.

FP formats on top of standard IEEE 754 *binary32* and *binary16*. The proposed FPU thus supports – along with *binary32* and *binary16* – two non-standard formats, namely *binary16alt* and *binary8*, collectively referred to as "SmallFloat" formats. To fully exploit the benefit of reduced precision formats, the proposed FPU implements SIMD operations on sub-32-bit formats, further increasing the performance and energy efficiency of the SoC. This concept is summarized in Fig. 4.1. It shows the energy savings on a sample application (k-nearest neighbors) achieved by shifting 32-bit FP operations to reduced precision while constraining the precision requirements of the whole kernel with the methodology described in Chapter 2. Note that significant energy savings can be achieved by exploiting SIMD operations that, on top of the lower energy cost of reduced precision instructions, also reduce the execution time of applications leveraging data-level parallelism (column 5 of Fig. 4.1).

We characterize the energy consumption of the SmallFloat instructions on the post-P&R implementation of the proposed TP SoC and evaluate the execution of a set of signal processing benchmarks to assess the energy saving when tuning the precision of the operations for pre-defined accuracy targets. Special attention has been paid to the design and analysis of the *binary16* and *binary16alt* datapath (i.e., pipelined vs. non-pipelined), which are widely used in most of the analyzed TP applications.

4.2.1 System Architecture

The TP SoC proposed in this section is based on the PULPino open-source architecture and is shown in Fig. 4.2. The processor core implements the RISC-V instruction set architecture and is optimized for energy-efficient digital signal processing. It includes custom extensions such as hardware loops, load, and store with address pre- and post-increment to speed-up pointer arithmetic and lightweight support for fixed-point computations, including small SIMD instructions and saturation instructions [50]. The SoC features 4 KiB of data memory, 4 KiB of instruction memory and a bootup ROM, tightly coupled to the processor, as well as a standard peripheral set which includes SPI, I2C, UART, timers and interrupt controller.

In this section we extend the PULPino SoC with a TP FPU supporting vectorization of reduced-precision operations. The hardware unit, pictured in Fig. 4.3, consists of three slices featuring a width of 32-bit, 16-bit and 8-bit, respectively, that support additions, subtractions and multiplications as well as conversion operations. *Binary8* is an 8-bit format featuring 3-bit of mantissa, and 5-bit of exponent, while *binary16alt* is a 16-bit format complementary to the IEEE 754 featuring 8-bit of exponent and 8-bit of mantissa. In order to enable SIMD sub-word parallelism inside the unit, the narrower slices are replicated such that two 16-bit or four 8-bit FP operations can execute simultaneously. Individual operation blocks are instantiated as Synopsys DesignWare FP Datapath IPs. Operand isolation logic is employed at every data path's inputs to save inactive subunits' dynamic power.

To meet the timing requirements of the SoC, 32-bit FP arithmetic operations are pipelined with one stage. Arithmetic operations in *binary8* and all conversion operations complete in one clock cycle. One design parameter explored in this section concerns the option of pipelining the 16-bit arithmetic operations: Although the timing requirements are met with both options, this parameter imposes a trade-off between the energy cost of 16-bit FP instructions and the number of cycles required to run applications on the system, analyzed in Section 4.2.3.

In addition to integrating the SmallFloat unit (SFU) into the execution stage of the RI5CY core, the decoder was extended with a custom set of SmallFloat instructions that bear similarity to standard

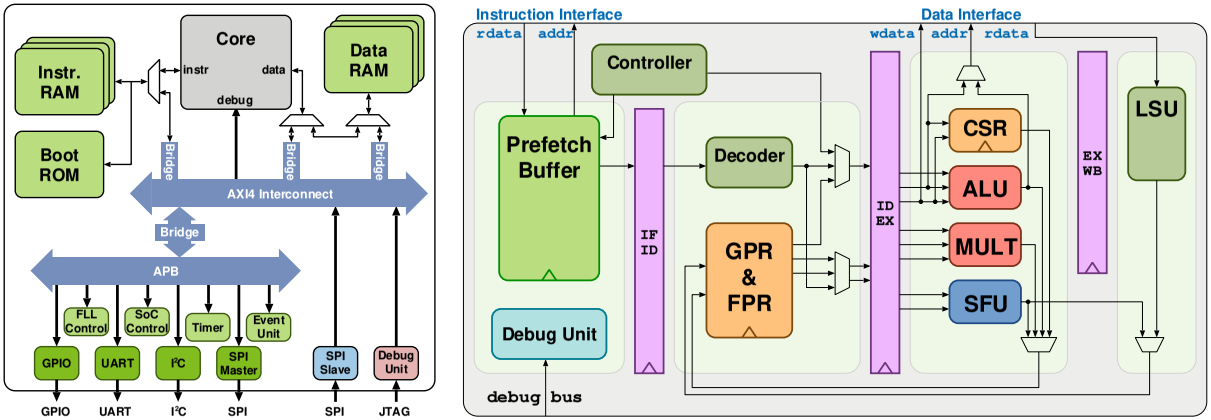


Figure 4.2: Simplified architectural overview of the PULPino SoC (left) and the RI5CY core (right). The new SmallFloat unit (labelled *SFU*) was implemented in the execution stage of the core.

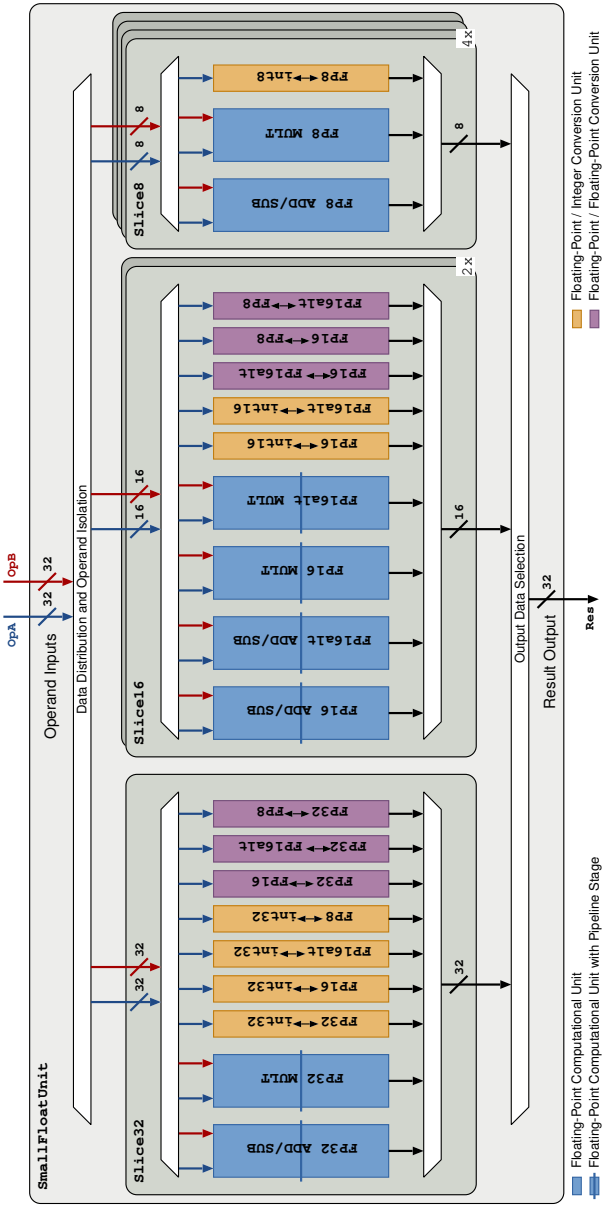


Figure 4.3: Sliced architecture used for the SmallFloat unit.

RISC-V FP instructions. *Binary32* operations utilize the FP register file. However, SmallFloat values are stored in the general-purpose register file to make them visible to the vector shuffling hardware already present in the RI5CY core.

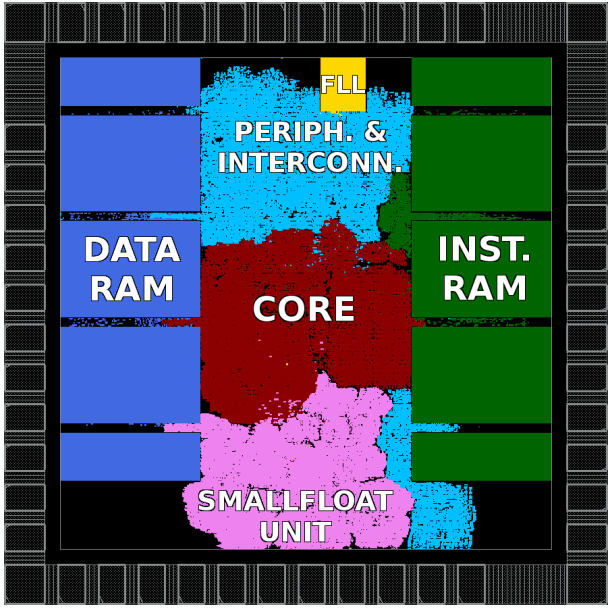
4.2.2 SoC Implementation

The SoC was synthesized and implemented (i.e., full layout) in the UMC 65 nm technology with Design Compiler 2015.6 and Cadence Innovus 15.2 using worst case libraries (slow-slow, 1.08 V, 125 °C) constraining the design to match the same target frequency of the original PULPino SoC (i.e., 350 MHz) [50]. Two versions of the SoC were implemented, one employing pipelining only for 32-bit operators, the other resorting to pipelining for both 32-bit and 16-bit arithmetic operations. The system's layout is shown in Fig. 4.4, together with a breakdown of the different blocks' area utilization.

The total area of the TP SoC is 0.906 mm^2, with the largest contributor to chip area being the data and instruction memory instances. The SmallFloat unit – supporting various operations on four FP formats – makes up a significant part of the new core, filling 47% of the core area in the baseline configuration with single-cycle 16-bit operations. In the pipelined scenario, the SFU shrinks to 40% of the core area since the strong timing pressure on the 16-bit FP arithmetic operations can be alleviated.

To provide an accurate estimation of the power consumption of the TP SoC and characterize the system-level power consumption of both original integer instructions and the new FP instructions, we conduced post-P&R power simulations in the typical corner (typycal-typical, 1.20 V, 25 °C). To this end, the Value Change Dump (VCD) traces of the system executing the various instructions have been generated with Mentor Modelsim 10.5c_3 and passed to Cadence Innovus to extract the power numbers. Figure 4.5 shows the power breakdown of the baseline TP system in the non-pipelined configuration. In contrast, the energy cost of pipelined and non-pipelined instructions is shown in Table 4.1.

Figure 4.5 outlines the power consumption of the major blocks in the system when running multiplication instructions on either intergers or 32-bit FP values. It should be noted that data memory is unused when



Chip area excluding pad instances.

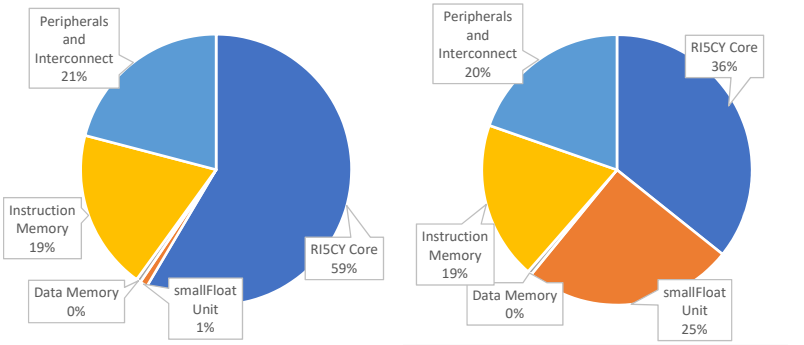
16-bit Pipelined?	No	Yes
Total [mm^2]	0.906	0.872
RI5CY Core	13.5%	13.7%
SmallFloat Unit	12.0%	9.0%
Data Memory	29.9%	31.1%
Instr. Memory	30.7%	31.8%
Periph. & Interc.	12.5%	13.1%
FLL	1.3%	1.4%

Figure 4.4: Full layout of the SoC (left), highlighting significant blocks. The area breakdown (right) shows both pipelined and non-pipelined configurations.

Table 4.1: Average energy per operation extracted from post-layout simulation, for various instruction groups on the two configurations of the TP SoC.

Format	Operation	Instruction	16-bit pipelined? No	Yes
	Idle Cycle	`nop`*	62.2 pJ	62.9 pJ
int32	Data movement	`lw`,`sw`*	94.4 pJ	94.6 pJ
	Arithmetic	`add`, `mul`*	106.4 pJ	102.4 pJ
binary32	Arithmetic	`f{add,mul}.s`*	106.8 pJ	102.4 pJ
	Conversions	e.g. `fcvt.w.s`*	79.7 pJ	78.1 pJ
binary16	Arithmetic	`f{add,mul}.h`†	98.8 pJ	82.0 pJ
	Conversions	e.g. `fcvt.h.s`†	74.7 pJ	74.6 pJ
	Vector Arithmetic	`vf{add,mul}.h`†	132.6 pJ	93.9 pJ
	Vector Conversions	e.g. `vfcvt.x.h`†	86.4 pJ	77.6 pJ
binary16alt	Arithmetic	`f{add,mul}.ah`†	87.2 pJ	83.2 pJ
	Conversions	e.g. `fcvt.ah.s`†	73.5 pJ	73.7 pJ
	Vector Arithmetic	`vf{add,mul}.ah`†	108.9 pJ	92.7 pJ
	Vector Conversions	e.g. `vfcvt.x.ah`†	79.5 pJ	74.3 pJ
binary8	Arithmetic	`f{add,mul}.b`†	74.0 pJ	75.5 pJ
	Conversions	e.g. `fcvt.b.s`†	72.5 pJ	72.8 pJ
	Vector Arithmetic	`vf{add,mul}.b`†	95.2 pJ	94.1 pJ
	Vector Conversions	`vfcvt.x.b`†	77.8 pJ	74.0 pJ

* RISC-V mnemonic † Custom SmallFloat mnemonic, based on RISC-V mnemonic



Integer multiplication `mul`. *Binary32* multiplication `fmul.s`.

Figure 4.5: Power distribution in the baseline SoC for multiply instructions.

executing an arithmetic instruction in isolation and thus has a negligible impact on system power. When an integer multiplication is performed, nearly 60% of the system power is used inside the ALU, while the SFU is isolated and clock gated, thus only contributing insignificant static power. During FP multiplications, the SFU consumes 25% of the system power. The ALU power consumption is reduced by approximately the same amount, indicating that our FP multiplier is similar to its integer counterpart.

Table 4.1 showcases the significant energy use of data movement, comparable to arithmetic operations themselves, and generally even overshadows the operations on SmallFloat types in the pipelined scenario. Vectorizing arithmetic operations costs up to 35% more energy at the system level, albeit at double or quadruple throughput, depending on the format used. Furthermore, vectorization directly reduces the number of load and store operations in the same way, drastically reducing the energy spent on a single value during a load-execute-store cycle. The impact of added timing pressure in the non-pipelined baseline over the pipelined configuration is visible, with an average 7% higher energy consumption, up to 40% in some cases. However, the baseline instructions complete within a single cycle, alleviating the need for stall cycles in difficult-to-schedule applications. Since there

is a trade-off between the total number of cycles and energy used per cycle, which depends on the target application's schedule friendliness, a set of benchmarks was run to explore this design space.

4.2.3 Benchmarking

Benchmarking of the TP SoC has been performed on a set of applications that implement algorithms for two domains of ULP systems near-sensor computing and embedded machine learning. SmallFloat types have been introduced in the source code using the methodology from Chapter 2. A software library (FlexFloat) emulates arbitrary FP types, and an external tool (fpPrecisionTuning) selects the smallest FP type among the supported ones for each variable, meeting strict constraints on the result accuracy. The accuracy of results is expressed as a value of the SQNR that program outputs must satisfy. Also, FlexFloat features a detailed run-time report on FP operations, which provides the number of executions classified by FP type, arithmetic operator, and class (i.e., scalar, vectorial, cast). The ANSI-C programs have been compiled using GCC 5.2 with a RISC-V back-end optimized for PULPino [50], featuring support for single-precision FP types as defined in the RISC-V instruction set architecture (ISA).

As performing a broad exploration of large applications requires a long simulation time on the RTL platform, the binaries have been executed on the cycle-accurate PULPino virtual platform, which provides detailed statistics. Since the current version of GCC does not include the support for the extended instruction set needed to handle *binary16*, *binary16alt* and *binary8* formats, we have used the *binary32* type to measure the exact number of cycles required by each instruction to execute. This value depends on the compiler's ability to schedule other operation classes to fill latency cycles and avoid stalls in the core pipeline, so it is strictly dependent on the application and compiler back-end.

To assess the trade-off between the pipelined and non-pipelined solution for *binary16* units, we have conducted an analytic exploration varying the percentage of latency slots filled for every application. Figure 4.6 depicts the ratio between the energy consumption of the pipelined design (Pipelined Energy) over the energy consumption of the non-pipelined design (Non-Pipelined Energy) for the analyzed

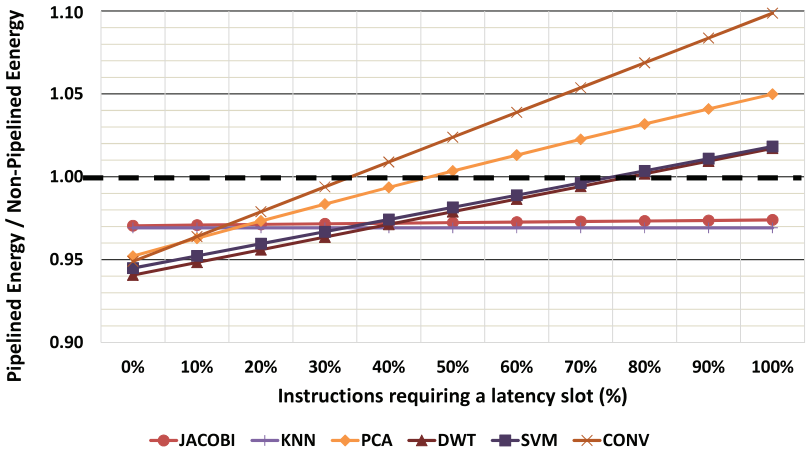


Figure 4.6: Energy of applications running on the *binary16* pipelined design normalized to that of non-pipelined design when varying the percentage of latency slots in in *binary16* and *binary32* operations.

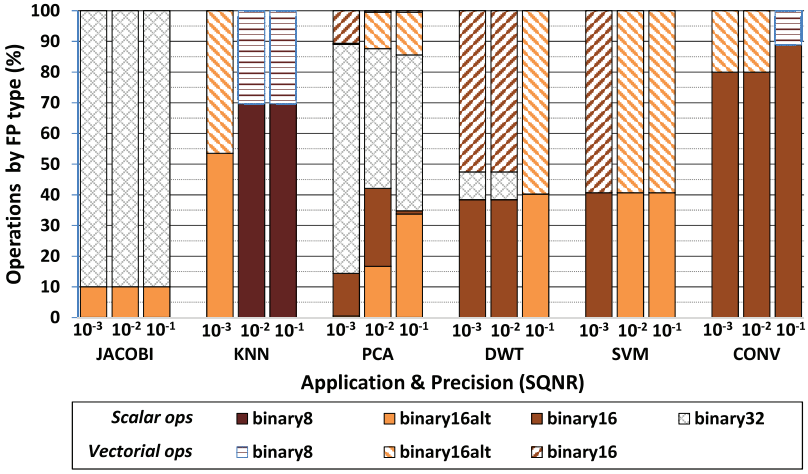


Figure 4.7: Breakdown of FP operations for three precision requirements.

applications. The horizontal axis reports the percentage of instructions that require a latency slot, which varies from 0% (no latency cycles for FP instructions) to 100% (a latency cycle per pipelined FP instruction). This experiment explores a full range of compiler capabilities to understand the trade-off between the energy/instruction (where pipelined is better) and the number of execution cycles (where non-pipelined is preferable). This trend may be better explained by examining a breakdown analysis of operation based on FP classes, depicted in Fig. 4.7.

The energy ratio of applications that contain a high amount of vectorial 16-bit operations w.r.t. scalar ones (DWT, SVM) grows moderately with the number of latency slots. In these cases, the energy savings of SIMD operation in the pipelined version are relatively higher than in the non-pipelined case. The ratio of applications with a predominance of *binary32* or *binary8* operations (JACOBI, KNN) is nearly constant since these operations are pipelined the same way in both designs. Finally, the ratio of applications characterized by a relatively high scalar 16-bit workload (PCA, CONV) is mainly affected by the number of latency slots, and their slopes are steep since scalar operations are heavily penalized by the pipelined design. However, if the compiler could reduce the latency slots under 70% using instruction scheduling techniques, these adverse effects would be highly mitigated.

Compiling the baseline version of applications (which uses *binary32* scalars only), we measured latency slots between 50% and 80%, so we conclude that the pipelined design is, in general, the best solution.

Figure 4.8 depicts groups of bars for each application running on the pipelined architecture, reporting a breakdown of the executions cycles for three precision requirements (SQNR = 10^{-3}, 10^{-2}, 10^{-1}). The bottom contributions consider the best execution scenario with no stalls due to latency cycles (0% latency). The gray segment on top considers the worst case, in which each operation involving 16-bit and 32-bit FP types requires a stall (100% latency). The reported values are normalized to the *binary32* baseline version of the application, and operation classes are highlighted with distinct patterns.

We observe performance improvements and energy savings when using TP operations mainly due to vectorization, which allows executing multiple reduced precision operations in parallel and reduce the number of memory accesses. Furthermore, energy is saved due to the

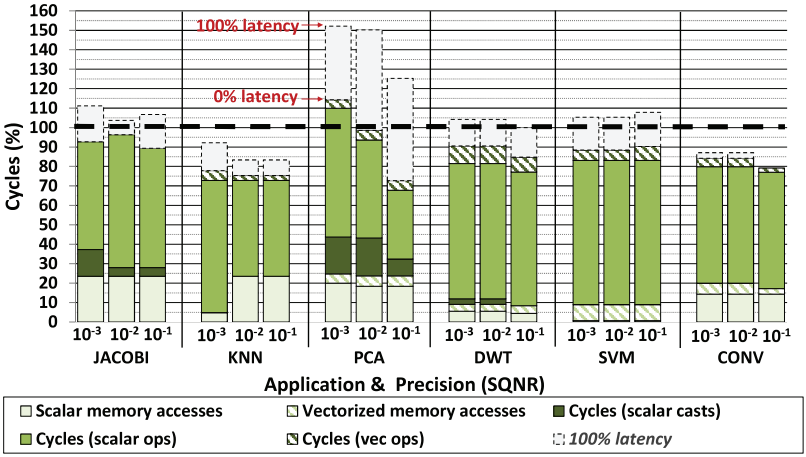


Figure 4.8: Execution cycles of applications for three precision requirements and assuming two latency slots conditions (0%, 100%), normalized to *binary32* baseline

reduced energy cost of reduced precision FP instructions themselves. Any overhead is mainly caused by the cast operations required to dynamically move from one FP format to another.

In Fig. 4.8 we see that on average, the number of cycles is decreased by 15% and 25% for 100% and 0% latency slots, respectively. The number of cycles reported for JACOBI and PCA is higher than the original version as SmallFloat variables are limited to disjoint program regions, introducing many casts. In other benchmarks, we can observe that the overhead of cast operations is not relevant.

Figure 4.9 reports the energy consumption of each application, normalized to the *binary32* baseline. Each bar contains three contributions, the FP operations (FP ops), the memory accesses (Memory ops), and all the remaining instructions (Other ops).

These values are strictly related to the ones shown in Fig. 4.8, and also in this figure, the contribution to energy consumption due to stalls is shown on top (100% latency). PCA's energy consumption is greater than the baseline due to the high number of casts coupled with a predominant number of scalar operations on *binary32* values.

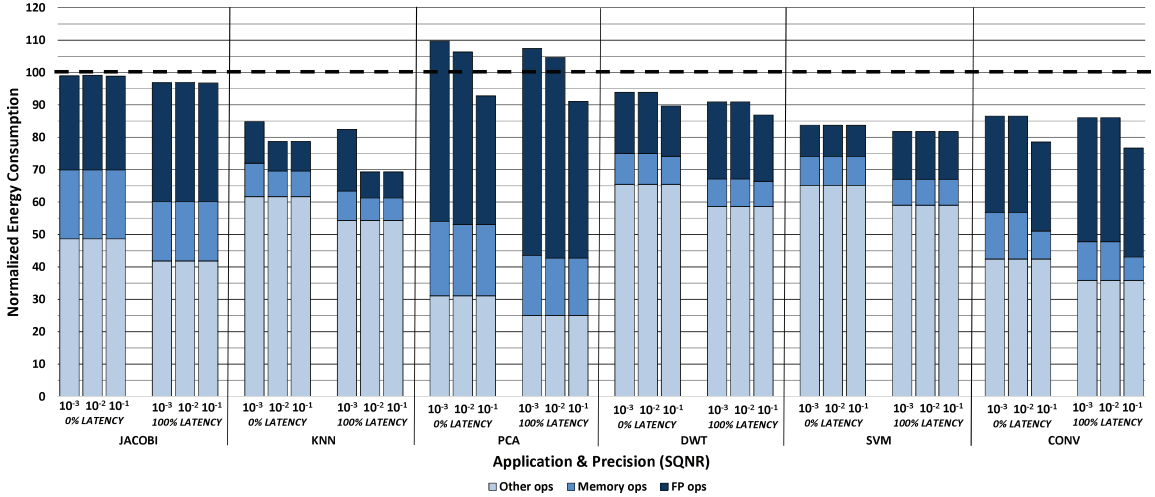


Figure 4.9: Energy consumption of applications for three precision requirements and assuming two latency slots conditions (0%, 100%), normalized to *binary32* baseline

The other applications have average energy savings between 14% and 18% compared to the baseline, with a maximum of 31% measured for KNN.

4.3 Augmenting RI5CY with FPnew

RI5CY is an open-source 32-bit, four stage, in-order RISC-V RV32IMFC processor [103]. This small core is focused on embedded and DSP applications, featuring several custom non-standard RISC-V extensions for higher performance, code density, and energy efficiency [50]. With this core, we want to showcase non-standard TP operations within a low-power MCU-class open-source RISC-V core, which has gained broad industry adoption [104].

4.3.1 Integration

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 registers and host FP data within the general-purpose register file to conserve area and reduce data movement[1].

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.

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

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

Table 4.2: The configuration of the TP-FPU as implemented into the RI5CY core. The FPU width is $\mathtt{w}_{fpu} = 32$ bit.

	Implementation (number of cycles, number of lanes)			
Format	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)

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, we can 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 compared to the infrastructure already in place for the "F" extension.

FPnew Configuration

We enable support for the above formats without adding any extra pipeline stages, as shown in Table 4.2. 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, $\mathtt{w}_{fpu}$ is set to 32 bit.

The ADDMUL block is implemented as a merged multi-format slice to allow for multi-format operations among FP16[alt] and FP32. The DIVSQRT block has been disabled as we do not utilize it for our case study, demonstrating 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.

4.3.2 Implementation Results

To benchmark applications on the TP-enabled RI5CY core, we perform a full P&R implementation of a platform containing the core. This section presents the implementation results, while Chapter 3 contains an application case study on the implemented design.

Implementation

We make use of PULPissimo [101] 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, I^2C, and SPI. We use our extended RI5CY core as described in Section 4.3.1, including the single-cycle TP-FPU configuration shown in Table 4.2.

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. 4.10. Synthesis and P&R 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 design's critical path is between the memories and the core, involving the SoC interconnect.

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. 4.11. The ADDMUL block hosting the merged multi-format 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 w.r.t. 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).

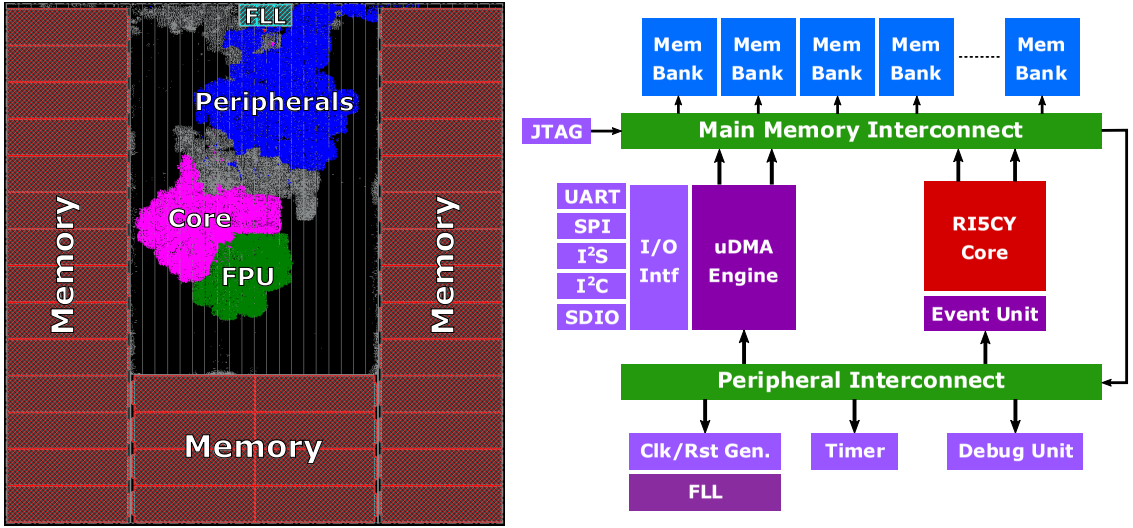


Figure 4.10: Floorplan and block diagram of the PULPissimo SoC (without pad frame) using RI5CY Core with TP-FPU.

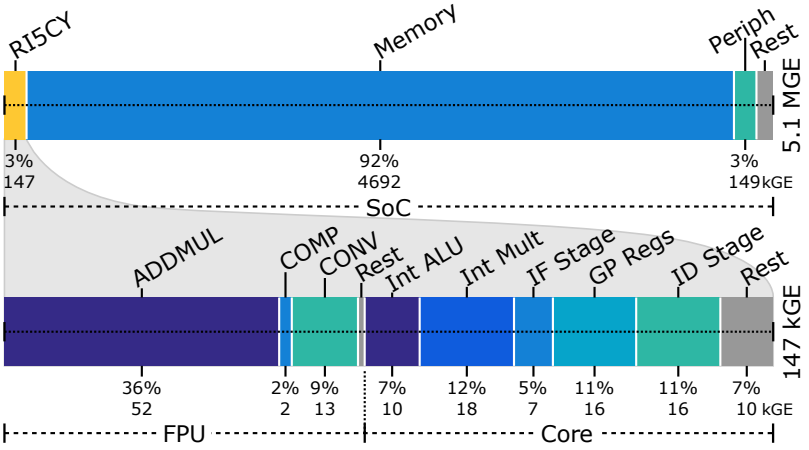


Figure 4.11: Area distribution of the PULPissimo SoC and the RI5CY core (in kGE, 1 GE $\approx 0.199\,\mu m^2$).

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 Chapter 3), achieving equal performance. These minor system-level differences in area and static and dynamic energy imply that FP computations are affordable even in an MCU context.

4.4 Embedded Transprecision Cluster Architectures

The pervasive adoption of edge computing increases the computational demand for algorithms targeted on low-power embedded devices operating in the mW range. Besides the aggressive optimization strategies

adopted on the algorithmic side (see Chapter 2), there is a great effort to find the best trade-off between architectural features and computational capabilities [105]. Indeed, deploying artificial intelligence algorithms or digital signal processing (DSP) on near-sensor devices poses several challenges to resource-constrained low-power embedded systems.

An architectural design that aims to achieve the goals discussed above must exploit additional features of the ULP computing domain. A tightly coupled cluster composed of several processing elements (PEs) enables improving the system's computational capabilities using parallel programming techniques without increasing the operating frequency. Specialized hardware extensions allow programmers to accelerate key parallel patterns and exploit the advantages of packed-SIMD operations. Combining these features with near-threshold computing on a fully programmable multi-core architecture leads to a highly scalable and versatile system suitable for a wide range of applications. The total number of FPUs and their sharing among the cluster cores require a careful evaluation since these aspects directly impact area and energy efficiency. For instance, having a dedicated FPU for each core can be detrimental if the data demand from the PEs can not be satisfied by the memory throughput: this effect is known as the Von Neumann bottleneck. Thus, reducing the number of FPUs and adopting a sharing policy among the cores can improve overall efficiency. Another aspect to consider is the pipelining of the FPU, which allows designers to increase the maximum operating frequency to the cost of potential deterioration of performance if pipeline latency cannot be completely hidden. In this complex scenario, finding the best trade-off requires an accurate exploration of the design space that includes the definition of adequate metrics and an experimental assessment of kernels from end-to-end applications.

In this section, we propose the design of a *TP computing cluster* tailored for applications in the domain of low-power (1 mW to 20 mW) near-sensor computing.

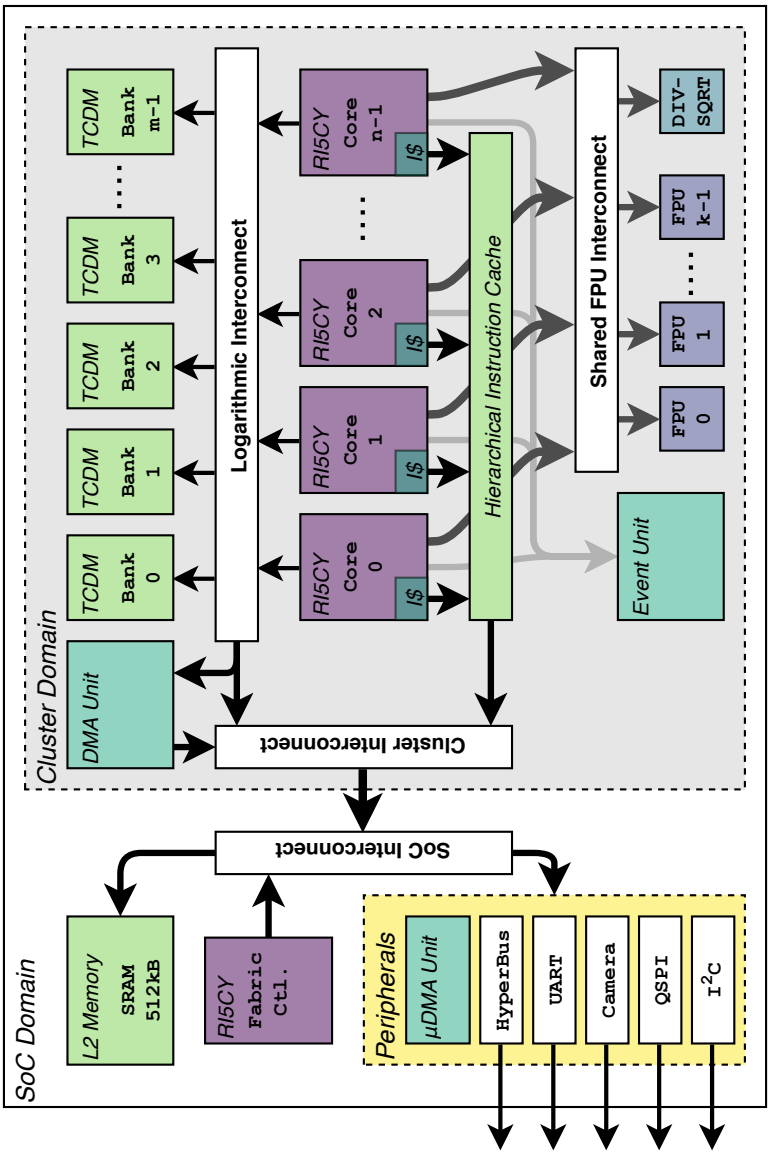


Figure 4.12: Top-level view of the proposed TP cluster.

4.4.1 Architecture and Implementation

Cluster Architecture

The cluster architecture proposed in this section is a soft IP implementing a tightly coupled cluster of processors built around a parametric number of RI5CY [50] cores. Figure 4.12 shows the top-level design of the TP cluster.

The cores fetch instructions from a 2-level shared instruction cache optimized for performance and energy efficiency when running SIMD workloads typical of near-sensor data analytics applications. To enable the single-cycle exchange of data among cores, they share a multi-banked Tightly-Coupled Data Memory (TCDM) behaving as a scratchpad memory (i.e., there is no data caching mechanism to avoid coherency and control overheads). The TCDM enables the cores to share data through a word-level interleaved, single-cycle latency logarithmic interconnect, allowing the execution of data-parallel programming models such as OpenMP. A dedicated hardware block (Event Unit) provides low-overhead support for fine-grained parallelism. It accelerates the execution patterns typical of data-parallel programming models (e.g., thread dispatching, barriers, and critical regions) and enables the adoption of power-saving policies when cores are idle [106].

Outside the cluster, at the SoC level, the architecture features one more memory hierarchy level, composed of a 15-cycle latency multi-banked scratchpad memory used to serve the core data bus, the instruction cache refills, and the cluster DMA. We base the explorations performed in this section on a set of cluster configurations with 8 and 16 cores. The L2 memory comprises 512 kB, the TCDM is 64 kB for the 8-core configurations and 128 kB for the 16-core ones. The cluster cores are connected to multiple FPU instances, whose number and interconnect are a central part of our exploration. Unlike the standard configuration for the RI5CY core, the proposed cluster does not employ core-private FPUs. Instead, a set of FPUs is shared among all cores in the cluster, using an interconnect which enables various mappings of cores to available FPUs. The following section provides insights into the FPU subsystem proposed in this section.

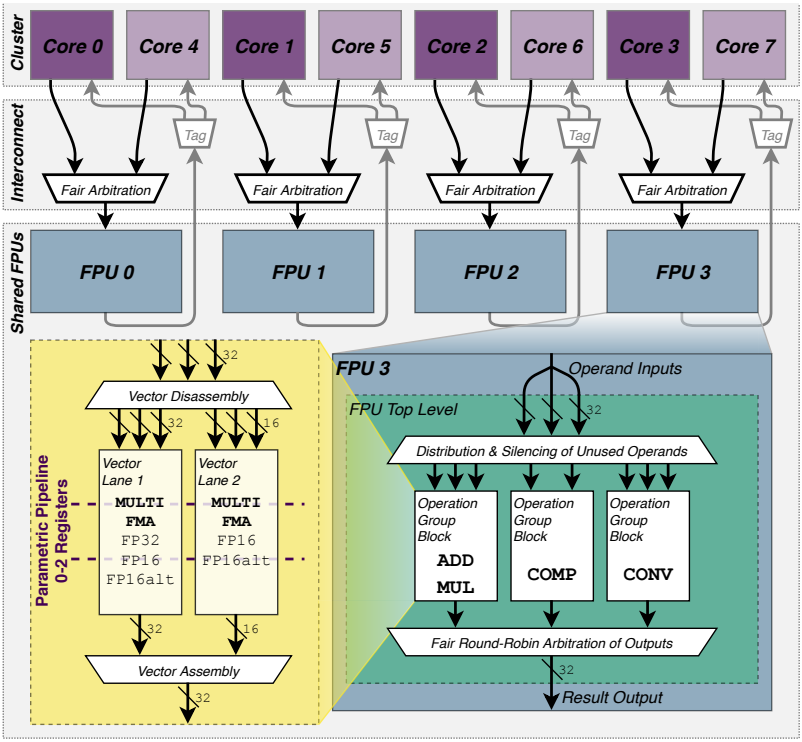


Figure 4.13: FPU sharing for the 8-core, 4-FPU configuration.

FPU and Interconnect

The cluster exploits configurations of FPnew (see Chapter 3) as FPU instances in our evaluation. FPnew is a parametric FPU architecture that supports multiple FP formats, SIMD vectors, and the insertion of any number of pipeline stages. Figure 4.13 (bottom) shows an architectural overview of a single shared FPU instance. The IP supports the standard IEEE 754 formats, *binary32* (float) and *binary16* (float16), as well as *bfloat16*. Some operations such as multiplication and FMA can also be performed as multi-format operations, taking the product of two 16-bit operands but returning a 32-bit single-precision result.

Such multi-format operations help many near-sensor data analytics applications accumulate data in a higher-precision variable to avoid overflows or precision loss. To make full use of the 32-bit data path, we enable packed-SIMD operations for the 16-bit types, boosting the execution performance when using 16-bit data types. Division and square root operations are disabled in the FPU instances as these operations reside in stand-alone blocks (DIV-SQRT), which are shared separately. The DIV-SQRT units feature a fixed latency of 11, 7, and 6 cycles for *float*, *float16*, and *bfloat16*, respectively. Moreover, since DIV-SQRT is designed as an iterative block, back-to-back pipelined operations are impossible when using these units.

The individual FPU instances are linked to one or more cores through a logarithmic tree interconnect, allowing to share one FPU among multiple cores in a fully transparent way from a software perspective. On the core side, the interconnect interface replaces the FPU in the execution stage, mimicking a core-private unit. The FPU instances connect to the cores through an auxiliary processing unit (APU) interface, featuring a *ready/valid* handshake and support tagging of all in-flight operations, requiring no modification to be shared.

In the proposed design, we employ a partial interconnect with a static mapping of FPUs to cores, such that a core (or a group of cores) will always access the same physical FPU instance. It arbitrates cases of simultaneous accesses to the FPU by using a fair round-robin policy and propagating the ready signal to only one core, stalling other competing cores. As such, the fact that FPUs are shared is transparent to both the core and FPU instances.

Moreover, we use a connection scheme with interleaved allocation to reduce access contentions on the FPUs in unbalanced workloads. For example, in a configuration featuring eight cores and four FPUs, units 0, 1, 2, 3 are shared among cores 0 & 4, 1 & 5, 2 & 6, and 3 & 7, respectively, as shown in Fig. 4.13 (top). This approach reduces the area and timing overhead compared to a monolithic, fully connected crossbar, which puts significant pressure on the paths from the cores to the first pipeline stage of the FPU, severely limiting the cluster's operating frequency and jeopardizing energy efficiency.

Our connection scheme provides an almost optimal allocation (only up to 1% overhead in performance has been measured against a fully

connected crossbar), avoiding contentions on the shared units also when the number of workers in parallel sections is smaller than the number of cores.

In the remainder of the section, we present a design space exploration of the proposed TP cluster, modifying the key configuration parameters presented previously in this section, namely the pipeline stages and sharing factor. The former's rationale lies in the fact that in most near sensor-data analytics applications, the density of FPU instructions is smaller than 50%, hence employing a private, per core FPU may form a bottleneck for area and energy. On the other hand, the pipelining of the FPU provides a powerful knob to tune the performance and energy efficiency of the TP cluster. If the number of cores and FPUs is equal (1/1 sharing factor), the system effectively degenerates into a core-private scenario, and the interconnect disappears from the design. In all the considered configurations, a single DIV-SQRT unit is shared among all cores. Finally, the proposed exploration involves designs of 8-core and 16-core clusters with supply voltages ranging from 0.65 V to 0.8 V to explore the whole design space in between energy-efficient and high-performance solutions.

Implementation

This section presents the physical implementation results and related explorations of the proposed cluster. Table 4.3 describes the 18 different configurations, given by the combination of the three architectural parameters (number of cores, number of FPUs, and number of FPU pipeline stages), as described in the previous section. The various configurations of the clusters have been synthesized using Synopsys Design Compiler 2019.12, using LVT libraries from 22nm FDX technology from Global Foundries. Physical implementation has been performed with Cadence Innovus v19.10-p002_1, using both 0.65 V near-threshold (NT) and 0.8 V super-threshold (ST) corners. We considered all permutations of operating conditions for signoff: fast and slow process transistors, 125°C and −40°C temperatures, ±10% of the voltage supply, as well as optimistic and pessimistic parasitics. Power analysis has been performed with Synopsys PrimeTime 2019.12 using the nominal corners at 0.65 V and 0.8 V, extracting *value change dump* (VCD) traces through parasitic-annotated post-layout simulation of

Table 4.3: Description of the architectural configurations of the proposed TP cluster that compose the design space. Cluster (8-16-cores), FP units (2-16), and pipeline stages (0-2).

Mnemonic	Cluster	FP units	Pipeline Stages
8c2f0p	8-cores	2	0
8c2f1p	8-cores	2	1
8c2f2p	8-cores	2	2
8c4f0p	8-cores	4	0
8c4f1p	8-cores	4	1
8c4f2p	8-cores	4	2
8c8f0p	8-cores	8	0
8c8f1p	8-cores	8	1
8c8f2p	8-cores	8	2
16c4f0p	16-cores	4	0
16c4f1p	16-cores	4	1
16c4f2p	16-cores	4	2
16c8f0p	16-cores	8	0
16c8f1p	16-cores	8	1
16c8f2p	16-cores	8	2
16c16f0p	16-cores	16	0
16c16f1p	16-cores	16	1
16c16f2p	16-cores	16	2

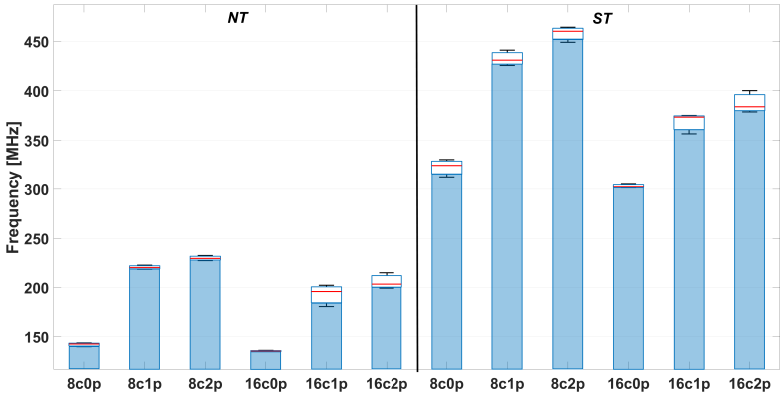


Figure 4.14: Minimum, maximum, and median values of the frequencies for all the configurations of the TP cluster, divided in NT and ST voltage corners.

a 32-bit FP matrix multiplication performed using Mentor Modelsim 2008.06. Each configuration has been synthesized and implemented at its maximum operating frequency. In contrast, power consumption has been analyzed at the same operating frequency for all configurations (100 MHz) to guarantee a fair comparison.

Figures 4.14 to 4.16 show the frequency, the area, and the power consumption of the cluster configurations analyzed in this work at 100 MHz. This frequency supports a power consumption in the range of 2 mW to 4 mW. In Fig. 4.14, we report the minimum, maximum, and median values of the frequencies obtained varying the number of FPUs. When considering single-cycle latency FPUs, we note that the entire system's operating frequency suffers profoundly. The long paths starting from the ID/EX registers of the core towards the FPUs and then back to the EX/WB registers form a considerable bottleneck for operating frequency. On the other hand, the absence of pipeline registers makes this solution relatively small and low-power. When moving to single-stage pipeline solutions, we note a very significant increase in the operating frequency when using NT cells (almost 50%). In contrast, the performance increase using ST cells is more limited

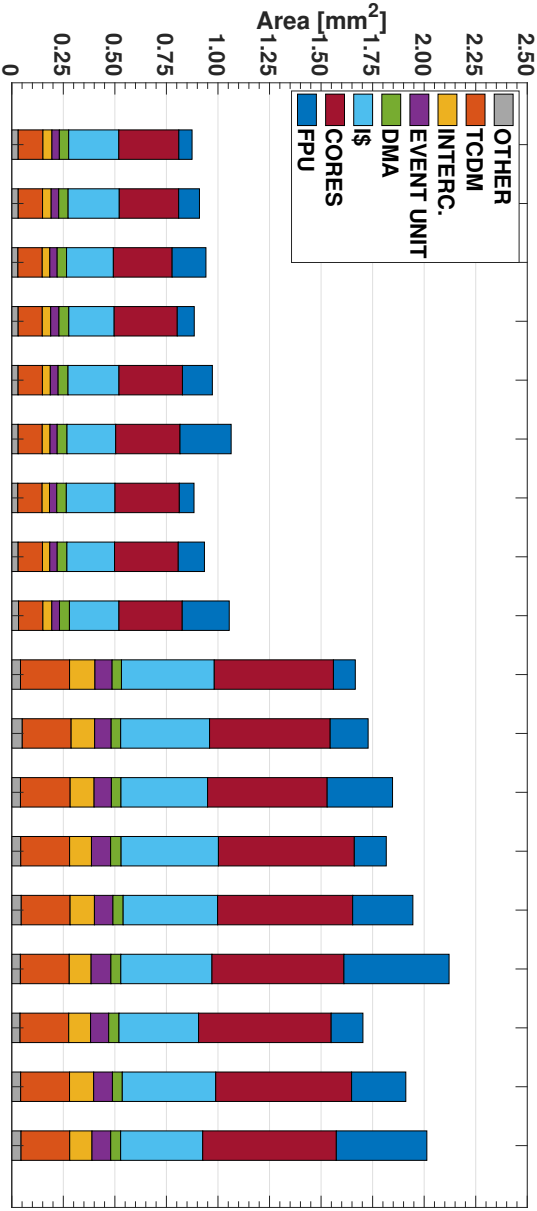


Figure 4.15: Total area of all the configurations in the design space of the TP cluster.

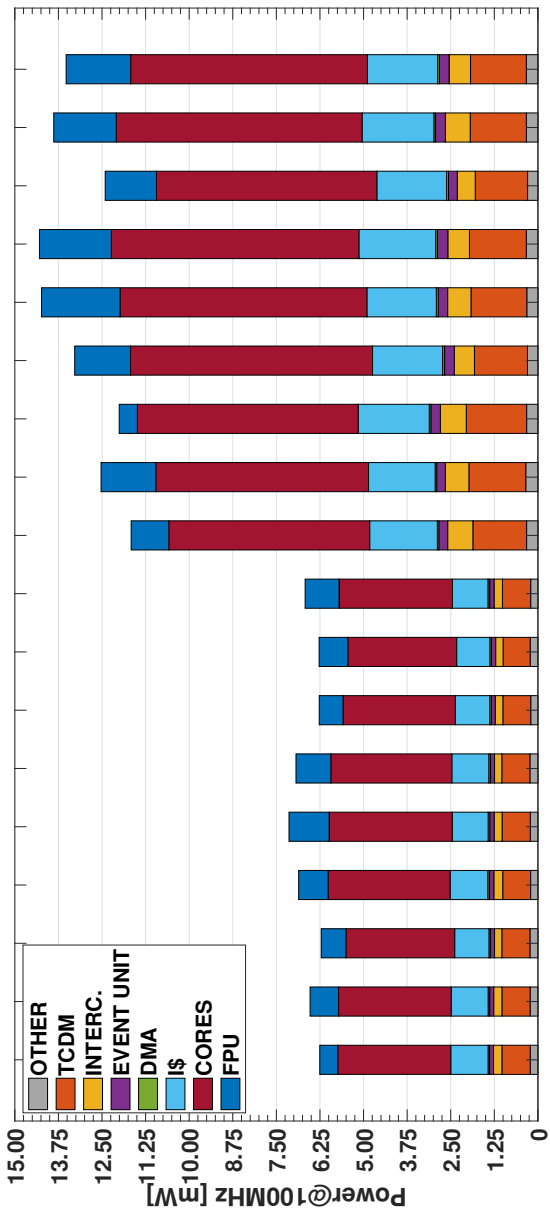


Figure 4.16: Total power consumption (at 100 MHz) of all the configurations in the design space of the TP cluster.

since the design already hits a structurally critical path from the TCDM SRAMs (featuring wide-voltage range but low-performance in ST) to the core through the logarithmic interconnect. We can observe an increase in power and area in all configurations featuring one pipeline stage due to the extra stage's additional overhead. When adding a second pipeline stage to the FPUs, we can see another slight increase of frequency in all configurations. We also encounter structurally critical paths using NT through the interconnect control paths to the instruction cache in these configurations. Although the area increases for all configurations, with two pipeline stages, the power consumption decreases thanks to the smaller timing pressure on the FPU.

Considering the sharing factor, we note that the FPU interconnect only negligibly impacts the frequency and that the area linearly increases when moving from 1/4 to 1/1 sharing for all configurations. On the other hand, when moving from 1/4 to 1/2 sharing factor, the power increases significantly due to high FPU utilization. When moving from 1/2 to 1/1 sharing, we note that the power consumption decreases in almost all cases. This effect occurs because even if we consider a highly intensive benchmark (e.g., matrix multiplication), the FP intensity around 50% leads to underutilization of the available resources, causing reduced power consumption. Additionally, the 1/1 configuration removes the interconnect, which relaxes the paths through the FPU, further reducing power consumption. Finally, if we consider the scaling of the number of cores, we can notice that most of the power components scale linearly with the number of cores (i.e., core power, TCDM power, and FPU power). On the other hand, other components such as the interconnect and the instruction cache scale superlinearly, indicating a smaller efficiency for the 16-core configuration. Moreover, the operating frequency of the 16-core cluster decreases compared to the one using eight cores. This effect is due to the longer path through the interconnects. Finally, we can notice that the area increases less than linearly due to some blocks not being duplicated, such as the DMA, the event unit, and the shared banks of the I$.

4.4.2 Software Infrastructure

The full exploitation of the TP cluster requires a comprehensive open-source software infrastructure, the basis of which has already been

introduced in Chapter 3.

Compiler Back-End

In addition to the middle-end passes described in the previous chapter, we further extended the compiler at the back-end level to support a parametric number of FPU pipeline stages. This parameter substantially impacts the instruction scheduling algorithm: Imprecise modeling of the FPU instruction latency may introduce stalls due to data dependencies with the result. We have modified the FPU pipeline description to include the hardware functional units and introduce a command-line option to specify the number of stages in the target configuration. Based on this option, the model specifies different latency and reservation delays for the functional units involved in FP operations.

Finally, we specified a set of platform-specific parameters for the instruction scheduling algorithm. The GCC algorithm uses a heuristic function to estimate the relative costs of operations; this value enables the choice of the best assembly sequence in case of multiple alternatives in the lowering process.

Parallel Programming Model

The architectural template of the TP cluster promotes a Single-Program Multiple-Data (SPMD) parallel paradigm. This paradigm is supported by a hardware abstraction layer (HAL), which allows minimal access to the platform features. The HAL provides information such as the core's identifier that can be used to organize the parallel workload for both data and task parallelism.

In this programming model, all the cores of the cluster follow the same execution flow unless the programmer explicitly indicates that a specific region should be executed by a subset of the cores, splitting the cores' workload concurrently on different data. Inter-core synchronization barriers are explicitly indicated to ensure the correctness of the results. Our architecture features dedicated hardware support that allows optimizing synchronization construct like barriers or critical sections.

The HAL provides the basic primitives to support high-level parallel programming models such as OpenMP. These models can provide a more intuitive interface at the cost of higher overhead, particularly when considering fine-grained workloads [107]. However, our exploration is focused on finding the maximum performance that we can obtain from real applications without considering a multi-layer software stack.

4.4.3 Experimental Results

Experimental Set-Up

The experiments have been performed on a hardware emulator implemented on a Xilinx UltraScale+ VCU118 FPGA board. The emulation on the FPGA provides cycle-accurate results, with a significant speed-up of the experiments compared to an RTL-equivalent simulation. A set of non-intrusive per-core performance counters included in the hardware design record the number of executed instructions and cycles spent in different states (total, active, L2/TCDM memory stalls, TCDM contention, FPU stall, FPU contention, FPU write-back stall, instruction cache miss). We have generated all the bitstreams for all the configurations reported in Table 4.3. After loading a bitstream on the FPGA, we load and run application binaries using OpenOCD and GDB interfaces. The same interface is used to load a program binary in the L2 memory, start the program execution, and finally read the performance counters from an emulated terminal. The values of power consumption used to calculate the efficiency have been derived from an annotated post-layout simulation, as described in Section 4.4.1.

Benchmarks

To evaluate the different configurations of the proposed TP cluster architecture, we analyzed eight benchmarks commonly used in the near-sensor processing applications for filtering, feature extraction, classification, and essential linear algebra functions. Table 4.4 illustrates the target benchmarks associated with their domains (i.e., audio processing, image processing, ExG biosignal processing). The Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) are digi-

Table 4.4: Main application domains (Domains), FP intensity (FP I.), and memory intensity (M. I.) for scalar and vector variants of the benchmarks.

		Scalar		Vector	
Apps	**Domains**	**FP I.**	**M. I.**	**FP I.**	**M. I.**
CONV	Audio, Image, ExG	0.33	0.67	0.28	0.29
DWT	Audio, Image, ExG	0.29	0.59	0.21	0.57
FFT	Audio, Image, ExG	0.32	0.52	0.26	0.38
FIR	Audio, Image, ExG	0.32	0.65	0.32	0.48
IIR	Audio, Image, ExG	0.19	0.55	0.17	0.33
KMEANS	ExG	0.55	0.36	0.44	0.30
MATMUL	Audio, Image, ExG	0.28	0.58	0.27	0.41
SVM	ExG	0.27	0.53	0.21	0.52

tal filters with various data acquisition and analysis applications. The Discrete Wavelet Transform (DWT) is a standard kernel used for feature extraction, which decomposes a signal into a different level of frequency resolutions through a bank of Low Pass (LPF) and High Pass Filters (HPF), capturing both temporal and frequency information.

The Fast Fourier Transform (FFT) is a mathematical method that transforms a signal from the time domain to the frequency domain. There are several variants of this algorithm; in this section, we consider the decimation-in-frequency radix-2 variant. We consider a state-of-the-art supervised classifier, the Support Vector Machine (SVM), widely used in near-sensor applications [108]. We also include another classifier named K-Means, an unsupervised ML algorithm able to infer an unknown outcome from input vectors. The last two kernels are basic linear algebra subprograms (BLAS) commonly used in DSP: matrix multiplication (MATMUL) and convolution (CONV), which is the most computing-intensive kernel in convolutional neural network (CNN) workloads.

We have implemented different variants of each kernel, using scalar (*float*) and vector (2 × *float16*, 2 × *bfloat16*) data types. There is no significant difference in execution time and energy consumption between *float16* and *bfloat16* vectors when considering the design of the FPU; in the following experiments, we report a single value for

Table 4.5: Performance [Gflop/s], energy efficiency [Gflop/s W], and area efficiency [Gflop/s mm^2] executing the benchmarks on the 8-cores configurations. Performance and area efficiency are computed at 0.8 V, energy efficiency at 0.65 V. The last group of lines reports normalized average values. A box around the metric value highlights the best configuration for each benchmark.

		Scalar									Vector								
		8c2f0p	8c2f1p	8c2f2p	8c4f0p	8c4f1p	8c4f2p	8c8f0p	8c8f1p	8c8f2p	8c2f0p	8c2f1p	8c2f2p	8c4f0p	8c4f1p	8c4f2p	8c8f0p	8c8f1p	8c8f2p
CONV	PERF	1,16	1,62	1,74	1,43	1,94	1,97	1,52	2,04	1,96	1,91	2,57	2,19	2,04	2,89	2,36	2,32	2,98	2,35
	E. EFF	72	66	72	82	76	83	91	83	81	119	105	91	117	113	100	139	121	97
	A. EFF	1,5	1,8	2,0	1,8	2,0	2,1	1,6	1,9	1,9	2,5	2,9	2,5	2,6	3,0	2,5	2,5	2,8	2,2
DWT	PERF	0,54	0,73	0,77	0,70	0,87	0,86	0,75	0,95	0,85	0,83	1,12	1,18	0,89	1,17	1,19	0,92	1,21	1,16
	E. EFF	33	30	32	40	34	36	45	39	35	51	46	49	51	46	50	55	49	48
	A. EFF	0,7	0,8	0,9	0,9	0,9	0,9	0,8	0,9	0,8	1,1	1,3	1,3	1,1	1,2	1,3	1,0	1,1	1,1
FFT	PERF	0,67	0,91	0,97	0,92	1,23	1,21	1,02	1,37	1,27	1,21	1,56	1,54	1,49	1,83	1,66	1,60	1,98	1,63
	E. EFF	42	37	40	52	48	51	61	56	52	75	64	64	85	72	70	96	80	67
	A. EFF	0,9	1,0	1,1	1,2	1,3	1,3	1,1	1,3	1,2	1,6	1,8	1,7	1,9	1,9	1,8	1,7	1,9	1,5
FIR	PERF	1,21	1,54	1,40	1,49	1,76	1,48	1,62	1,88	1,47	2,24	3,03	2,76	2,54	3,38	2,86	2,70	3,57	2,79
	E. EFF	75	63	58	85	69	63	97	76	61	139	124	114	145	132	121	162	145	115
	A. EFF	1,6	1,7	1,6	1,9	1,8	1,6	1,7	1,8	1,4	3,0	3,4	3,1	3,2	3,5	3,1	2,9	3,3	2,6
IIR	PERF	0,61	0,82	0,86	0,70	0,90	0,91	0,74	0,94	0,91	1,06	1,40	1,46	1,15	1,49	1,49	1,19	1,55	1,48
	E. EFF	38	33	35	40	35	39	45	38	37	66	57	60	65	58	63	72	63	61
	A. EFF	0,8	0,9	1,0	0,9	0,9	1,0	0,8	0,9	0,9	1,4	1,6	1,6	1,4	1,5	1,6	1,3	1,5	1,4
K-M.	PERF	0,75	1,02	1,05	1,15	1,49	1,30	1,34	1,68	1,30	1,21	1,64	1,72	1,68	2,13	2,06	1,88	2,33	2,10
	E. EFF	47	42	43	66	58	55	80	68	53	75	67	71	96	83	87	113	94	86
	A. EFF	1,0	1,1	1,2	1,4	1,5	1,4	1,4	1,6	1,2	1,6	1,9	1,9	2,1	2,2	2,2	2,0	2,2	2,0
MAT.	PERF	1,06	1,54	1,51	1,28	1,74	1,61	1,35	1,81	1,61	2,10	2,77	2,68	2,36	3,16	2,77	2,46	3,32	2,71
	E. EFF	66	63	62	73	68	68	81	73	66	130	113	111	135	123	117	148	135	111
	A. EFF	1,4	1,7	1,7	1,6	1,8	1,7	1,4	1,7	1,5	2,8	3,1	3,0	3,0	3,2	3,0	2,6	3,1	2,6
SVM	PERF	0,53	0,69	0,69	0,59	0,74	0,71	0,62	0,77	0,70	0,63	0,85	0,82	0,68	0,89	0,83	0,69	0,91	0,81
	E. EFF	33	28	29	34	29	30	37	31	29	39	35	34	39	35	35	42	37	33
	A. EFF	0,7	0,8	0,8	0,7	0,8	0,8	0,7	0,7	0,7	0,8	1,0	0,9	0,9	0,9	0,9	0,7	0,9	0,8
NAVG	PERF	0,00	0,24	0,26	0,16	0,40	0,35	0,23	0,48	0,35	0,40	0,76	0,73	0,54	0,92	0,80	0,62	1,00	0,78
	E. EFF	0,13	0,01	0,04	0,27	0,12	0,16	0,43	0,24	0,13	0,73	0,54	0,52	0,79	0,62	0,62	1,00	0,76	0,56
	A. EFF	0,03	0,20	0,23	0,24	0,30	0,29	0,12	0,27	0,14	0,66	0,91	0,87	0,80	0,94	0,86	0,61	0,85	0,61

Table 4.6: Performance [Gflop/s], energy efficiency [Gflop/s W], and area efficiency [Gflop/s mm^2] executing the benchmarks on the 16-cores configurations. Performance and area efficiency are computed at 0.8 V, energy efficiency at 0.65 V. The last group of lines reports normalized average values. A box around the metric value highlights the best configuration for each benchmark.

		Scalar									Vector								
		16c4f0p	16c4f1p	16c4f2p	16c8f0p	16c8f1p	16c8f2p	16c16f0p	16c16f1p	16c16f2p	16c4f0p	16c4f1p	16c4f2p	16c8f0p	16c8f1p	16c8f2p	16c16f0p	16c16f1p	16c16f2p
CONV	PERF	2,19	2,80	2,93	2,61	3,10	3,14	2,71	3,37	3,26	3,51	4,28	3,63	3,69	4,54	3,74	4,00	4,78	3,87
	E. EFF	77	69	72	84	75	73	94	79	78	123	106	89	118	110	87	140	113	92
	A. EFF	1,5	1,8	2,0	1,7	1,8	1,9	1,7	1,8	1,8	2,5	2,7	2,5	2,5	2,7	2,2	2,5	2,5	2,2
DWT	PERF	0,74	0,91	0,95	0,87	0,98	0,97	0,89	1,06	1,00	0,88	1,07	1,10	0,94	1,05	1,06	0,94	1,11	1,08
	E. EFF	26	22	23	28	24	23	31	25	24	31	26	27	30	26	25	33	26	26
	A. EFF	0,5	0,6	0,6	0,6	0,6	0,6	0,5	0,6	0,6	0,6	0,7	0,8	0,6	0,6	0,6	0,6	0,6	0,6
FFT	PERF	1,21	1,51	1,56	1,52	1,78	1,81	1,60	1,99	1,90	2,13	2,54	2,50	2,25	2,62	2,53	2,22	2,74	2,58
	E. EFF	42	37	38	49	43	42	56	47	45	74	62	61	72	63	59	78	64	62
	A. EFF	0,8	1,0	1,1	1,0	1,0	1,1	1,0	1,1	1,1	1,5	1,6	1,7	1,5	1,5	1,5	1,4	1,5	1,5
FIR	PERF	2,29	2,66	2,38	2,71	2,86	2,39	2,85	3,08	2,47	4,17	5,19	4,62	4,62	5,38	4,54	4,79	5,92	4,62
	E. EFF	80	66	59	87	69	56	99	73	59	146	128	114	148	130	106	167	139	110
	A. EFF	1,6	1,7	1,6	1,8	1,7	1,4	1,8	1,6	1,4	2,9	3,3	3,2	3,1	3,1	2,7	3,0	3,1	2,6
IIR	PERF	0,78	0,95	0,99	0,81	0,93	0,95	0,81	0,98	0,97	1,37	1,69	1,73	1,42	1,62	1,65	1,41	1,71	1,68
	E. EFF	27	23	24	26	23	22	28	23	23	48	42	42	46	39	39	49	40	40
	A. EFF	0,5	0,6	0,7	0,5	0,5	0,6	0,5	0,5	0,5	1,0	1,1	1,2	1,0	0,9	1,0	0,9	0,9	0,9
K-M.	PERF	1,14	1,39	1,39	1,28	1,45	1,35	1,25	1,50	1,40	1,72	2,11	2,13	1,95	2,28	2,22	1,93	2,43	2,29
	E. EFF	40	34	34	41	35	32	43	35	33	60	52	52	63	55	52	67	57	54
	A. EFF	0,8	0,9	1,0	0,9	0,8	0,8	0,8	0,8	0,8	1,2	1,3	1,5	1,3	1,3	1,3	1,2	1,3	1,3
MAT.	PERF	1,96	2,57	2,41	2,23	2,65	2,41	2,30	2,86	2,50	3,98	4,83	4,57	4,34	5,14	4,48	4,42	5,47	4,57
	E. EFF	68	63	59	72	64	56	80	67	60	139	119	113	139	125	104	154	129	109
	A. EFF	1,4	1,6	1,6	1,5	1,5	1,4	1,4	1,5	1,4	2,8	3,0	3,1	2,9	3,0	2,7	2,7	2,9	2,6
SVM	PERF	0,90	1,07	1,06	0,99	1,11	1,07	1,00	1,19	1,09	1,14	1,40	1,39	1,21	1,38	1,33	1,21	1,47	1,36
	E. EFF	31	26	26	32	27	25	35	28	26	40	34	34	39	34	31	42	35	32
	A. EFF	0,6	0,7	0,7	0,7	0,6	0,6	0,6	0,6	0,6	0,8	0,9	1,0	0,8	0,8	0,8	0,7	0,8	0,8
NAVG	PERF	0,00	0,23	0,24	0,15	0,31	0,27	0,17	0,41	0,31	0,51	0,84	0,80	0,62	0,88	0,77	0,64	1,00	0,81
	E. EFF	0,22	0,05	0,06	0,28	0,10	0,02	0,43	0,15	0,08	0,85	0,58	0,54	0,82	0,59	0,43	1,00	0,64	0,50
	A. EFF	0,03	0,17	0,29	0,16	0,15	0,14	0,08	0,12	0,10	0,67	0,87	0,97	0,73	0,76	0,67	0,61	0,69	0,60

both configurations.

Each variant accepts a parameter representing the number of cores available in the current configuration to exploit the parallelism provided by the TP cluster. The source code includes a form of parametric parallelism based on the number of available cores and the core id, using the low-overhead HAL interface described in Section 4.4.2. We exploited data parallelism at the loop level with static scheduling of the iterations on the available cores. This policy guarantees maximum balancing with a limited overhead related to the computation of per-core iteration boundaries. Whenever feasible, we apply data parallelism to the benchmarks' outer loops (CONV, FIR, MATMUL). In other cases, data parallelism is applied to single stages of the algorithm, separated by a synchronization barrier (DWT, FFT, KMEANS, SVM); except for FFT, these benchmarks are characterized by sequential regions interleaved with parallel loops and executed by a single core.

A common problem of IIR filters working on a single stream is that data dependencies limit parallelism. To alleviate this limitation, we have adopted a technique based on a block formulation of recursive filters tailored for vector units [109]. The algebraic transformations required by this technique are applied off-line and do not imply any overhead. However, the algorithm's time complexity is higher than the original one, and the size of the vector state (equal to the number of taps) severely limits the exploitability of parallelism. For this reason, the vector variant of this benchmark is the only reported case with alternative configurations achieving the best result for energy efficiency.

Table 4.4 also reports the FP and memory intensity of the benchmarks for scalar and vector variants. The FP intensity is computed as the ratio between the number of FP instructions and the total number of instructions. Analogously, the memory intensity is the number of load/store instructions over the total number of instructions. These numbers provide a quantitative evaluation of the pressure on the FPU and memory subsystems. They are essential to understand the actual FP workload in a real execution scenario.

Performance, Energy Efficiency, and Area Efficiency

We have performed extensive benchmarking considering all the benchmark variants and all the configurations of the TP cluster. We have

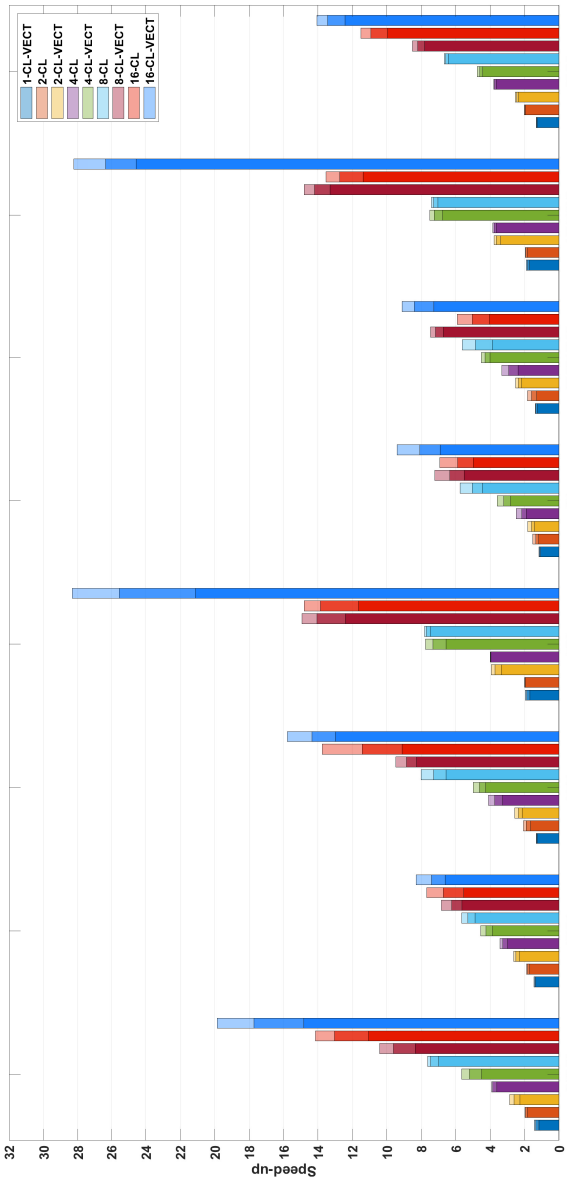


Figure 4.17: Speed-ups obtained executing scalar and vector variants on all the platform configurations. Each configuration reports the number of available cores and the support to vectorization. Each bar shows the minimum (dark color), maximum and average (light color) value.

measured the performance [Gflop/s], the energy efficiency [Gflop/s W], and the area efficiency [Gflop/s mm^2] for each benchmark variant and platform configuration: Table 4.5 and Table 4.6 report the result of the experiments. Each table's last three rows report the measures' average values, with normalized values ranging between 0 and 1. Computing these metrics allows establishing the configurations that guarantee the best performance and energy/area efficiency for the considered benchmarks. Moreover, the tables use a color scale to visually emphasize the worst (light color) and the best (dark) configurations.

The configuration with 16 cores, private FPUs, and one pipeline stage provides the best performance, with a maximum of 3.37 Gflop/s for scalars and 5.92 Gflop/s for vectors. Intuitively, using the maximum number of cores and FPUs is beneficial for performance. An additional pipeline stage could increase the frequency, but this is not the case due to critical structural paths (as discussed in Section 4.4.1.

The configuration with 16 cores, private FPUs, and zero pipeline stages is the most energy-efficient, with a maximum of 99 Gflop/s W and 167 Gflop/s W for vectors. Using the maximum number of cores is never detrimental to performance, mainly thanks to adopting aggressive power-saving policies that turn off cores waiting for synchronization events. Moreover, this configuration prevents the occurrence of FPU stalls, which are detrimental to energy efficiency.

The configuration with 8 cores and 4 shared FPUs configured with one pipeline stage is the most area-efficient, with a maximum of 2.0 Gflop/s mm^2 for scalars and 3.5 Gflop/s mm^2 for vectors. This configuration saves area by reducing the number of cores and the sharing factor, but maintaining a single pipeline stage represents the best trade-off with performance.

Parallelization and Vectorization Figure 4.17 depicts the speed-ups from the execution of the benchmarks on the 16-core architectures, combining the benefits deriving from parallelism and vectorization. Each configuration of the TP cluster is denoted by the abbreviation n-CL, where n indicates the number of cores. The suffix VECT designates the execution of the vector variant. The baseline to compute the speed-up is the execution on a single core with no vectorization support. The bars show the average, maximum, and

minimum values of the speed-ups executed on all the architectural configurations.

Focusing on the parallel speed-up, we can notice that the values reported for DWT, IIR, and K-MEANS are modest, reaching a saturation point around 8. These benchmarks have a complex parallel execution flow, requiring several synchronization barriers and regions with sequential execution to ensure correctness, limiting the parallelism. However, this effect is not detrimental to energy efficiency, as discussed in Section 4.4.3 The rest of the kernels (CONV, FFT, FIR, and MATMUL) demonstrate a nearly ideal speed-up.

Vectorization leads to an additional improvement of the speed-up – between 1.3× and 2× – thanks to the beneficial effects described in Section 1. Moreover, the improvement derived from vectorization is higher than the parallel speed-up for some applications. When passing from 8CL-VECT (8 cores working on vectors) to 16CL (8 cores working on scalars), this trend is more pronounced for FIR, IIR, MATMUL KMEANS. This effect is due to the different overheads related to parallelization and vectorization. As discussed above, IIR and K-MEANS require several synchronization barriers and regions with sequential execution sematic. Conversely, FIR and MATMUL are amenable to advanced manual vectorization techniques. For instance, the vector variant of MATMUL reaches a near-ideal improvement vectorizing both input matrices. The efficiency is achieved by unrolling the two inner loops, adding shuffle operations to compute the transpose, and using a dot-product intrinsic to accumulate two products. A similar technique is applied to FIR. On the other side, the complex multiplication kernel required by FFT requires seven cycles for scalar data and ten cycles for vector data; consequently, the maximum gain from vectorization is 1.43×.

Sharing Factor Figure 4.18 reports average values of performance, energy efficiency, and area efficiency varying the sharing factor. The left part of the figure references 8-core configurations, the right part 16-core ones. The number of pipeline stages has been set to one for all experiments, while the number of FPUs corresponds to sharing factors 1/4, 1/2, and 1/1, respectively.

As a general trend, performance grows when increasing the sharing

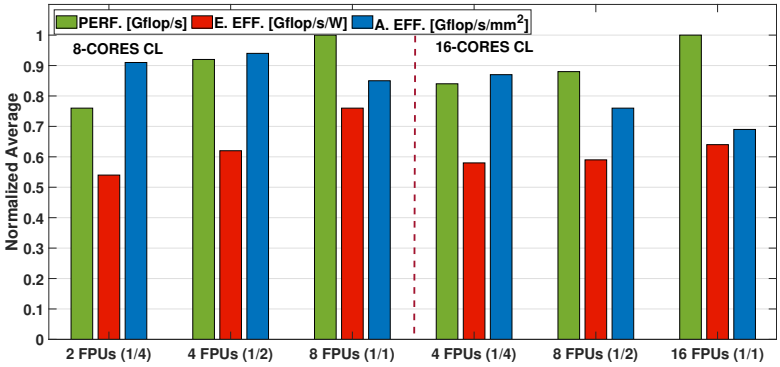


Figure 4.18: Performance (PERF.), energy efficiency (E. EFF.), and area efficiency (A. EFF.) fixing one pipeline stage and varying the number of FPUs. The values are the average of the normalized results.

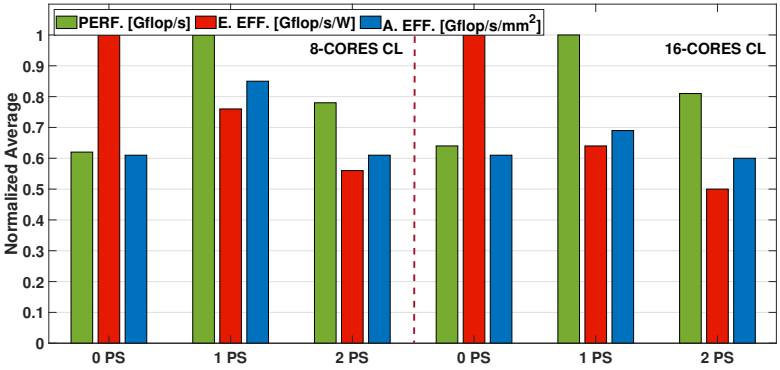


Figure 4.19: Performance (PERF.), Energy efficiency (E. EFF.), and area efficiency (A. EFF.) fixing a 1/1 sharing factor and varying the pipeline stages (PS). The values are the average of the normalized resultss.

factor. This increment is more evident, passing from 1/4 to 1/2 in 8-core configurations, and passing from 1/2 to 1/1 in the 16-core configurations.

The energy efficiency increases with the sharing factor. This effect has a minor relevance for 16-core configurations because the contribution of FPUs to the total energy consumption is proportionally lower. Conversely, the area efficiency increases by reducing the sharing factor from 1/1 to 1/4. This trend is inverted in the transition from 1/4 to 1/2 with eight cores. This effect is related to the FP intensity of benchmarks, which is always less than one (as expected in real applications). A 1/2 sharing factor can sustain an FP intensity up to 0.5 with no additional stalls. This value is enough for most applications' requirements, considering that 0.31 is the average FP intensity of the benchmarks in Table 4.4. On the 16-core configuration, the number of FPUs to reach the same sharing factor is higher, implying a significant increase of the area; in this case, the best area efficiency corresponds to the minimum sharing factor (1/4).

Pipelining Figure 4.19 shows average values of performance, energy efficiency, and area efficiency varying the number of pipeline stages. The support for pipelining improves performance since this technique allows for increasing the operating frequency of the TP cluster. Conversely, performance degrades with two pipeline stages. Even if we can increase the operating frequency, we observe an increment in the number of cycles due to the register file's write port contentions. A write-back stall may happen when a load/store post-increment operation or an integer operation arrives right after an FP operation when configuring the FPU for two pipeline stages. For instance, when we encounter the valid signal for the FP operation in clock cycle n, and a load/store post-increment request in cycle $n + 1$, the FPU must wait until the other instructions end before storing its results, causing a stall for the use of the port. There are no contentions with zero pipeline stages because there is a dedicated write port for the FPU.

In all cases, energy efficiency decreases by incrementing pipeline stages since the design makes the logic more complex. Finally, area efficiency follows a trend very similar to performance. The area required to enable one-stage pipelining leads to a considerable benefit, while

Table 4.7: Comparison with state-of-the-art architectures in high-performance and low-power embedded domain.

	Ara [110]	**Hwacha** [111]	**Snitch** [112]	**Ariane** [43]	**NTX** [43]	**Xavier***	**STM32H7**†	**Mr.Wolf** [105]	**This work** Best perf. (16c16f1p) Best en. eff. (16c16f0p) Best area eff. (8c4f1p)
Domain	High-perf.	High-perf.	High-perf.	High-perf.	High-perf.	Embedded	Embedded	Embedded	Embedded
Technology	GF 22FDX	45nm SOI	GF 22FDX	GF 22FDX	GF 22FDX	TSMC 12FFN	40nm CMOS	40nm CMOS	GF 22FDX
Voltage (V)	0.80[2]	0.80[1]	0.80[2]	0.80[1]	0.80[1]	0.75[1]	1.80 / 1.80[1]	1.10[1]	0.80 / 0.65 / 0.80[3]
Frequency (GHz)	1.04	0.55	1.06	0.92	1.55	1.38	0.20 / 0.48	0.45	0.37 / 0.30 / 0.43
Area (mm^2)	2.14	3.00	0.89	0.39	0.56	11.03	–	10.00	2.10 / 1.80 / 0.97
Performance (Gflop/s)	64.80	3.44	14.38	2.04	18.27	153.00	0.03 / 0.07	1.00	**2.86** / 2.30 / 1.74
Energy eff. (Gflop/s/W)	81.60	25.00	103.84	33.02	110.05	52.39	0.44 / 0.33	4.50	26.00 / **81.00** / 23.40
Area eff. (Gflop/s/mm^2)	30.34	1.14	25.83	5.23	32.63	13.84	–	1.70	1.50 / 0.60 / **1.78**
FP formats	float float16 bfloat16 minifloat	double float	double float	float float16 bfloat16 minifloat	float‡	float float16	float	float	float float16 bfloat16
Programming interface	ISA extension	ISA extension	ISA extension	ISA extension	Memory-mapped configuration	Base ISA	Base ISA	Base ISA	ISA extension
Execution model	SIMD vector unit (accelerator)	SIMT vector-thread unit (accelerator)	Loop-buffers for tensor streaming (accelerator)	SIMD processor	Loop-buffers for tensor streaming (accelerator)	SIMT vector-thread unit (accelerator)	Processor	Multi-core processor	Multi-core processor
Compiler support	Yes	Yes (OpenCL)	Partial (inline ASM)	Yes	No	Yes (CUDA)	Yes	Yes	Yes

* Numbers extracted from [112]. † Measurements taken on a NUCLEOH743ZI development board executing a 128×128. matrix multiplication. [1] Silicon measurements. [2] Post-layout simulation using *typical* frequency. [3] Post-layout simulation using *worst-case* frequency. ‡ Higher internal accumulation precision with float results.

a further area increase for a second stage is not equally convenient. This trend has a minor impact on 16-core configurations since the contribution of pipeline logic becomes negligible.

4.4.4 Comparison with the SoA

Table 4.7 depicts a comparison with SoA platforms with FP support in high-performance and embedded domains. The number of FP operations has been measured by executing a single-precision matrix multiplication on all the platforms. We have considered three configurations of the TP cluster (reported in the TP column), corresponding to the best performance, the best energy efficiency, and the best area efficiency. Our solution outperforms a single-core Cortex core and the Mr.Wolf multi-core cluster in all metrics in the targeted domain of low-power embedded systems. Comparing with high performance embedded platform, the Tegra Xavier SoC contains eight streaming multiprocessors (SMs) composed of four execution units. Each execution unit includes 16 single-precision FPUs sharing a register file and an instruction cache. The TP cluster is 53% better than an SM in terms of energy efficiency. As regards performance, a single execution unit is 13× faster.

As expected, absolute performance and area efficiency in the high--performance domain is higher than the TP cluster due to the higher operating frequencies. In our design, we consider the worst-case corner for the computation of the operating frequency. Simultaneously, the other solutions report silicon results or typical corners, which is penalizing our results. Nevertheless, our solution outperforms an Ariane and is comparable with a Hwacha vector processor. The energy efficiency of the TP cluster is comparable with Snitch, NTX, and Ara, despite these architectures are heavily specialized for FP intensive computations. This outcome is due to three main factors. Firstly, operating at low voltage in near-threshold operation makes the TP cluster very power efficient. Secondly, the best solution is not to adopt pipelining, so it does not pay the pipeline logic's energetic overhead. Thirdly, the support to FMA operations increases the number of operations performed per cycle by 2× and is highly beneficial.

Finally, compared to most energy-efficient solutions in Table 4.7, the proposed cluster provides full compiler support and flexibility typical of

high-level parallel programming models such as OpenMP, not requiring programmers to use low-level accelerator-centric interfaces such as OpenCL or memory-mapped APIs or even lower-level abstractions (e.g., inline assembly). This support is a crucial requirement for the wide adoption of these solutions for near-sensor computing.

4.4.5 Related Work

This section provides an overview of the current SoA. First, we consider the main alternatives available for platform and component design. Then, we explore the role of packed-SIMD and vector units in embedded platforms, as they are central to our discussion. We also discuss software-based approaches since they represent a platform-agnostic alternative to our solution. Finally, we provide an overview of both low-power and high-end embedded systems that provide FP support because these architectures provide us a baseline to compare our results.

Non-IEEE 754 Floating-Point Formats

In recent years, researchers have started to explore custom formats that are alternative to the IEEE 754 standard ones and their closer derivatives (e.g., bfloat16). Gautschi et al. [70] propose an FPU based on the logarithmic number system (LNU), which is up to 4.1× more energy-efficient than standard FPUs in non-linear processing kernels. Universal numbers (*unums*) [77] adopt a variable-width representation based on interval arithmetic to guarantee that the result contains the exact solution (see Chapter 2). The variable width provided by unum enables to scale up precision in scientific computing applications [113], but hardware implementations not suitable for the area and energy constraints of the embedded computing domain. A recent version of the unum specification, known as *unum type III* or *posit* [93], proposes a solution to the hardware overhead issue. In [40] we introduce a posit arithmetic unit supporting additions and subtractions, coupled with dedicated compression units to reduce the memory footprint.

We mention these solutions as they represent a viable alternative to reduce energy consumption in our target domain. However, we have not included these hardware components in the TP cluster design for

two main reasons. First, current hardware implementations are characterized by high overhead compared to actual benefits (in Chapter 2, we estimate a reduction of memory footprint around 7%). Second, their adoption requires a significant effort in code refactoring.

Transprecision Computing Building Blocks

The choice of an energy-efficient and TP-enabled FPU design is a crucial enabler for this work. In literature, there are several designs of FPUs that enable TP operations. For instance, Kaul et al. [67] describe a variable-precision fused multiply-and-add (FMA) unit with vector support (1, 2, or 4 ways). Their design considers 8 bit for the exponents and 24 bit for the mantissa. Moreover, a 5-bit certainty field tracks the number of accurate mantissa bits; computations that do not fulfill the application's accuracy constraints are recomputed with increased precision. The energy consumption for a 32 nm CMOS implementation is 19.4 pJ/flop, even though the overhead due to precision tracking and fixed-size exponents increases the total energy consumption at the application level. Moreover, if maximum precision is required, applications become very inefficient due to the need for repeated operations performed at a lower precision.

Nannarelli [95] describes the design of an FPU based on the Tunable floating-point (TFP) format, which supports a variable number of bits for mantissa (from 4 to 24) and exponent (from 5 to 8). However, this solution does not support vectorization, which is a crucial enabler for energy efficiency. Jaiswal et al. [114] present a pipelined design of two FP adders that support multiple precision configurations. The results are promising in terms of area and energy efficiency, but this solution does not support additional FP operations. Hardfloat [88] is an open-source library (written in Chisel) that contains parameterized blocks for FMA operations, conversions between integer and FP numbers, and conversions among different FP formats. This library offers individual function blocks instead of a fully-featured FPU, missing unit-level optimizations at the current development stage. Zhang et al. [98] present a multiple-precision FP FMA with vector support Their work aims at minimizing the area overhead, but the hardware sharing inside the datapath constrains all formats to use the same latency. Moreover, the FPU does not provide any support for scalars in smaller

formats.

FPnew (see Chapter 3) is an open-source TP-FPU capable of supporting a wide range of standard (double, float, and float16) and custom (bfloat16 and 8-bit minifloat) FP formats. FPnew supports both scalar and packed-SIMD operations, and the experimental results shown in Chapter 3 assess that this design outperforms all its competitors in terms of area and energy efficiency. We have integrated this FPU in our architecture, Section 4.4.1 describes its design and integration aspects in further detail. FPNew includes a DIVSQRT module to compute divisions and square roots using an iterative non-restoring divider, similar to the design presented in [115].

Packed-SIMD Support And Vector Units

Intel initially introduced Packed-SIMD extensions for FP computations with the MMX and SSE ISAs. ARM later introduced the NEON extension, which supports up to 128-bit single-precision FP operations, which has been evaluated as a better solution than the Intel counterpart in low-power embedded platforms [116]. In this context, heterogeneous architectures have also provided support for packed-SIMD instructions featuring a DSP accelerator, such as Texas Instruments Keystone II [117]. Considering the impact of packed-SIMD instructions on the processing capabilities of DSP platforms, the RISC-V consortium is working on a dedicated ISA extension [118].

Variable-length vector units have been initially introduced on the CRAY-1 architecture [119], and today they are a well-established solution in high-end computer systems. The ARM Scalable Vector Extension (SVE) [120] is a vector extension introduces as a SIMD instruction set for the AArch64 architecture. The SVE specification allows system designers to choose a vector register length between 128 and 2,48 bit to satisfy different constraints. The programming model is vector-length agnostic; there is no need to recompile the source code or use compiler intrinsics to change the vector length. The A64FX chip by Fujitsu in TSMC 7nm technology implements the SVE extension, including 48 cores with support for 512-bit vectors and reaching peak performance of 2.7 Tflop/s [121]. This chip has been used in Fugaku, which entered the TOP500 list in June 2020 as the fastest supercomputer globally.

The current working draft for the RISC-V "V" vector extension [122] defines a variable-length register file with vector operation semantics. This extension supports FP16, FP32, FP64, and FP128 types and includes widening FMA operations to support mixed-precision computations (e.g., multiplying two FP16 registers and adding the result to an FP32 register). ARA [110] and Hwacha [111] are two embodiments of this provisional standard. ARA includes an RV64 core and a 64-bit vector unit based on the version 0.5 draft of the RISC-V vector extension. Hwacha is based on the vector-fetch paradigm and is composed of an array of single-issue, in-order RISC-V Rocket cores [111]. In general, these solutions' area and power consumption are too high for low-power, MCU-class processing systems. This observation is the main reason why vector semantics in ULP embedded systems are typically supported by providing packed-SIMD instructions.

Software-Based TP Approaches

Besides approaches involving custom HW design to enable mixed-precision operations, several researchers have proposed multiple-precision arithmetic libraries that extend the IEEE 754 formats to perform FP computations with arbitrary precision. This solution allows application designers to overcome the limitations of fixed-format FP types without dedicated hardware support. ARPREC [60] and GNU MPFR [61] provide APIs to handle multiple formats characterized by a fixed-size exponent (a machine word) and an arbitrary size mantissa (multiples of a machine word). Arb [123] is a C library for arbitrary-precision interval arithmetic using the midpoint-radius representation that outperforms non-interval solutions such as MPFR in some applications. These libraries are widely used in contexts requiring a high dynamic range and are characterized by relaxed constraints on computation time and energy consumption (e.g., scientific computing on data center nodes). To speed up the library execution time, Lefèvre [124] presents a new algorithm to speed up the sum of arbitrary-precision FP number using the MPRF internal representation, However, the approach based on software emulation is not a viable solution for energy-efficient embedded systems. Both time and energy efficiency are negatively affected by at least an order of magnitude compared with solutions

based on dedicated hardware.

Anderson et al. [125] propose a software approach for the reduced-precision representation of FP data. They define a set of non-standard FP multibyte formats (flytes) that can be converted to the next larger hardware type to perform arithmetic computations. The exponent is set to the maximum bitwidth of the containing type to minimize the conversion overhead. The adoption of the vector units available on general-purpose processors (e.g., Intel Haswell) or high-end accelerators (e.g., Intel Xeon Phi) allows the software library to coalesce memory accesses and then amortize the conversion overhead.

Low-Power Parallel Architectures for FP Computing

Coarse Grain Reconfigurable Architectures (CGRAs) recently emerged as a promising solution for the near-sensor processing domain. CGRAs are systolic arrays containing a large number of processing elements with a low-latency routing interconnect. MuTARe [126] is a CGRA working in the near-threshold voltage regime to achieve low energy. This solution improves by 29% the energy efficiency of a heterogeneous platform based on the ARM big.LITTLE platform. However, MuTARe targets high-end embedded systems and does not provide FP support. Transpire [127] is a CGRA architecture with FP support. The authors state an improvement of around 10× in performance and energy efficiency compared with a RISC-V core extended with packed-SIMD vectorization. These benefits are limited to specific algorithms since the design of CGRAs enables programmers to exploit different combinations of data-level and pipeline-based parallelism. However, near-sensor processing includes a wide variety of algorithms presenting complex access patterns that cannot be efficiently implemented on CGRAs.

Mr.Wolf [105] is a multi-core programmable processor implemented in CMOS 40nm technology. The platform includes a tiny (12 Kgates) RISC-V core accelerated by a powerful 8-core cluster of RI5CY cores [50] sharing two single-precision FPUs. The limited number of FPUs provided by this architecture represents a severe limitation to the maximum FP intensity that applications may expose. A primary contribution of our work consists of finding the best trade-off between the number of cores and the number of available FPUs, yet considering strict area and power constraints, and exploiting TP units to improve

performance and execution efficiency. In Section 4.4.4, we include Mr.Wolf in our comparison with SoA platforms.

Helium [128] is an extension of the Armv8.1-M architecture targeting low-power MCU-class computing systems. The ISA extension includes a set of scalar and vector instructions supporting fixed-point (8-bit, 16-bit, and 32-bit) and FP (float and float16, optionally double) formats. These instructions are beneficial for a wide range of near-sensor applications, from machine learning to DSP. The Cortex-M55 [129] core includes the Helium extension, but chips based on this IP are not yet available on the market to perform a comparison with our solution.

High-End Embedded Systems for FP Computing

The most widely used commercial architectures for compute-intensive FP workloads are GP-GPUs. With the growth of emerging applications such as training of neural networks, they have also started to support reduced precision FP formats such as *brain-float* and *binary16*. Indeed, training algorithms for deep neural networks such as backpropagation are naturally robust to errors. These features of modern GPUs have also been exploited in other application domains, such as machine learning [130] and linear algebra [131], demonstrating significant benefits for performance and efficiency. NVidia Pascal has been the first GPU supporting 16-bit FP formats. NVidia Pascal features SIMD float16 operations that can be executed using a single paired-operation instruction. Furthermore, the new Volta micro-architecture further extends support to reduced precision types featuring mixed-precision multiply-and-add instructions.

Other research works targeting neural network training and FP intensive workloads leverage more specialized architectures. Neurostream [132] is a streaming memory-mapped co-processor targeting inference and training of deep neural networks in near-memory computing systems. This design removes the register-file bottleneck of SIMD architectures accessing the memory exploiting programmable hardware loops and address generators, enabling execution efficiency close to one MAC per cycle. Neurostream achieves an average performance of 240 Gflop/s within a power-budget of 2.5 W. The architecture has been further improved in [133], with a 2.7× energy efficiency improvement over GPGPUs at 4.4× less silicon area, delivering 1.2 Tflop/s. The

latter architecture has been implemented in 22nm FDX in Kosmodrom [43]. The chip includes two Ariane cores and one NTX accelerator. Kosmodrom achieves an energy efficiency of 260 Gflop/s W and a 28 Gflop/s performance within a 6.2 mW to 400 mW power envelope.

In [112], the memory-mapped control has been replaced by a tiny general-purpose processor meant to drive double-precision FPUs, improving the efficiency and flexibility of previous approaches. This architecture introduces two ISA extensions to reduce the pressure on the core: the stream semantic registers (SSR) and the FP repetition instruction (FREP). SSRs allow the core to implicitly encode memory accesses as register reads/writes, removing a significant number of explicit memory instructions. The FREP extension decouples the FP and integer pipeline by sequencing instructions from a micro-loop buffer. The evaluation on an octa-core cluster in 22 nm technology reports a 5× multi-core speed-up and a 3.5× gain in energy efficiency.

The architectures discussed in this section target the domain of servers and high-end embedded systems, and presenting further details is beyond our work's scope. However, the comparison with these solutions provides valuable insight and is discussed in Section 4.4.4.

4.5 Notable Embedded Systems Using FPnew

The Internet-of-Things requires end-nodes with ultra-low-power always-on capability for a long battery lifetime, as well as high performance, energy efficiency, and extreme flexibility to deal with complex and fast-evolving near-sensor analytics algorithms (NSAAs). We present Vega, an always-on IoT end-node SoC capable of scaling from a 1.7 µW fully retentive cognitive sleep mode up to 32.2 Gop/s (at 49.4 mW) peak performance on NSAAs, including mobile DNN inference, exploiting 1.6 MB of state-retentive SRAM, and 4 MB of non-volatile MRAM. To meet the performance and flexibility requirements of NSAAs, the SoC features 10 RISC-V cores: one core for SoC and IO management and a 9-core cluster supporting multi-precision SIMD integer and FP computation. Two programmable ML accelerators boost energy efficiency in sleep and active state, respectively.

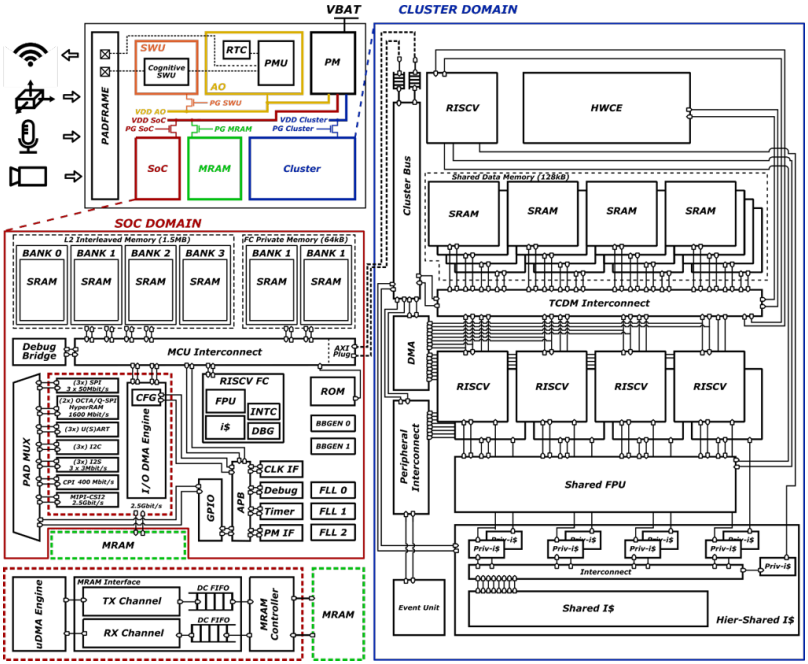


Figure 4.20: Vega SoC architecture and power domains.

The Vega system is an evolution of the TP cluster architecture presented in the previous section. As shown in Fig. 4.20, the SoC consists of two main switchable power domains (SoC and cluster), plus an always-on domain operating from 0.6 V to 0.8 V supplied by two commercial off-the-shelf on-chip regulators (buck converter plus LDO) from 3.6 V (VBAT). Two body-bias generators are included for process compensation of SoC and cluster domains. A Power Management Unit (PMU), clocked by a 1 MHz internal ring oscillator, manages transitions between the power states of the SoC.

The SoC Domain is an advanced MCU featuring a RISC-V processor named Fabric Controller (FC) and several peripherals, including a 1.6 Gbit/s DDR interface supporting external IoT DRAMs such as Cypress Semiconductor's HyperRAM (Fig. 4.20). The cluster, built around nine 70 kGE 4-pipeline stage RISC-V cores, is turned on and adjusted to the required frequency when applications running on the FC offload computation-intensive kernels. The cores share data on a 128 kB shared multi-banked L1 memory through a 1-cycle latency logarithmic interconnect. The cluster L1 memory can serve 16 parallel memory requests with <10% contention rate even on data-intensive kernels, delivering up to 28.8 GB/s at 450 MHz. The program cache is hierarchical: 512 B private per-core plus 4 kB of 2-cycle latency shared cache to maximize efficiency with data-parallel code.

The RISC-V cores feature extensions (RVC32IMF-Xpulp) for NSAAs, such as hardware loops, post-incremented LD/ST, SIMD such as dot products operating on narrow 16-bit and 8-bit data types. The cores share four instances of FPnew as shown in Fig. 4.21, supporting FP32, FP16, and bfloat16 operations.

Fine-grained parallel thread dispatching is accelerated by a dedicated hardware event unit, which manages clock gating of idle cores waiting for synchronization and enables resuming execution in 2 cycles. ISA extensions coupled with parallelism and optimized memory hierarchy deliver a MAC/cycle performance 62× better than a baseline RISC-V ISA running on a single-core, similar to [134]. The cluster delivers up to 15.6 8-bit Gop/s and up to 614 Gop/s W, and up to 3.3 Gflop/s and 129 Gflop/s W, on GP processors, demonstrating leading-edge performance on a wide range of NSAA (Fig. 4.22).

Figure 4.23 shows a die micrograph, highlighting system components included in the measurements. More details about this imple-

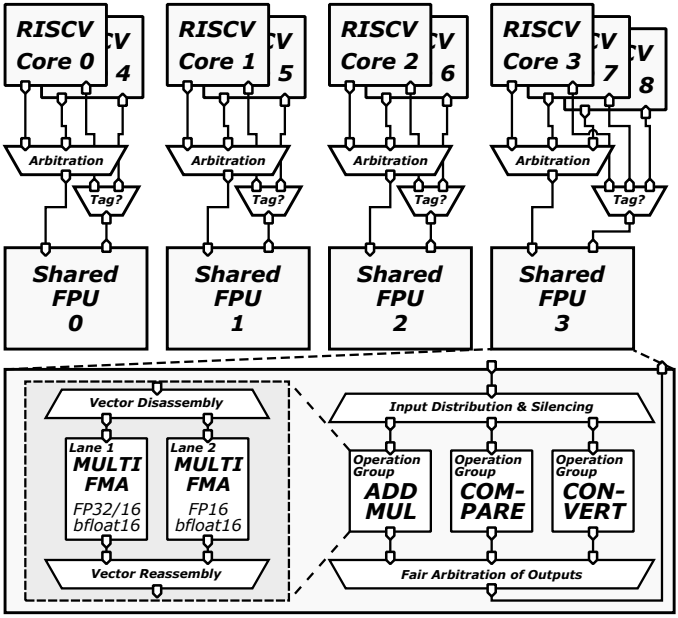


Figure 4.21: Architecture of the shared multi-precision FPU and its integration in the 9-core cluster.

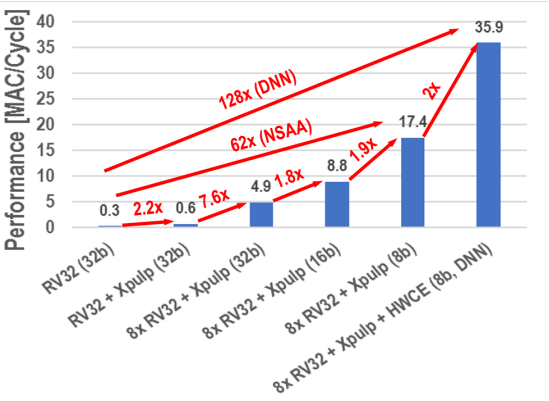
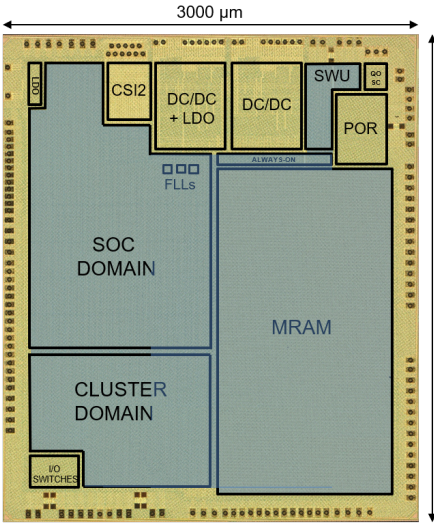


Figure 4.22: Performance of integer matrix multiplication exploiting Xpulp Extensions, software parallelism on 8 cores, SIMD parallelism on 16-bit and 8-bit datatypes, HWCE. Similar performance gains can be achieved on FP kernels when applicable.



Technology	22nm FDSOI
Chip Area	12mm^2
SRAM	1728 kB
MRAM	4 MB
WU Sources	GPIO, RTC, Cognitive
VDD range	0.5V - 0.8V
VBB range	0V - +1.1V
Freq. Range	32 kHz - 450 MHz
Pow. Range	1.7 µW - 49.4 mW
Int. Perf.	15.6 GOPS
Int. Eff.	614 GOPS/W
F.P. Perf.	3.3 GFLOPS
F.P. Eff.	129 GFLOPS/W
Int. ML Perf.	32.2 GOPS
Int. ML Eff.	1.3 TOPS/W

Figure 4.23: Chip micrograph and specifications.

mentation can be found in [41].

GreenWaves Technologies have announced their GAP9 processor [135], a commercial IoT application processor based on this design and containing FPnew to enable sub-32-bit TP FP computation.

4.6 Summary and Conclusion

Our work on TP hardware and making such hardware usable in systems has led to the implementation of several embedded-class SoC platforms. The main findings of this chapter are:

- We have presented an SoC for ULP TP computing, extending the PULPino microcontroller architecture with the TP-FPU prototype from Chapter 2 within the RI5CY processor core. Energy efficiency and performance of the SoC are increased by implementing SIMD operations on sub-32-bit formats. The results of our exploration show that the introduction of a SmallFloat unit improves system performance by 15% to 25% and the energy efficiency by 14% to 18% for 100% and 0% latency slots, respectively, on the analyzed applications when allowing the target precision to be relaxed by 10% compared to a traditional *binary32* baseline.

- We have added FPnew into the RI5CY core, using our TP RISC-V ISA extension, both developed in the last chapter and implemented the core into the PULPissimo microcontroller. Compared to a standard PULPissimo system with support for only FP32, area increases by 0.7% and static energy by 0.9%, due to the significant impact of memories (92% and 96% of system area and leakage, respectively). Considering the system-level energy consumption, operating on FP32 data proves to be very affordable and averages 22.2 pJ per cycle. In comparison, the integer variant would require 21.2 pJ for running a filtering kernel (see Chapter 3), achieving equal performance.

- We have described the design of a TP cluster for near-sensor computing, performing a design space exploration on an FPGA emulator varying the number of cores, the number of FPUs,

and the pipeline stages. A set of experiments on near-sensor algorithms and an analysis on post-P&R models have allowed us to identify the most efficient configurations. Our experimental results show that the configuration with 16 cores and private FPUs configured with one pipeline stage provides the best performance (5.92 Gflop/s). The one with 16 cores and private FPUs configured with zero pipeline stages is the most energy-efficient (167 Gflop/s W), while the configuration with eight cores and four shared FPUs configured with one pipeline stage is the most area-efficient (3.5 Gflop/s mm^2). Finally, the energy efficiency of the TP cluster outperforms all the other solutions that provide FP support in the area of embedded computing.

- We have outlined Vega, an evolution of the TP cluster for always-on embedded IoT SoC. With respect to fully programmable IoT end-nodes [134, 105], the proposed SoC delivers more than 1.3× - 2× better performance and 3.2× - 4.3× better efficiency on NPAA workloads. On non-DNN, NSAA workloads, our SoC achieves 10× and 2.5× higher performance and energy efficiency, respectively, and the design is being commercialized.

We conclude that our efforts have created ample opportunities to push for ever-higher energy efficiency in embedded systems while simultaneously improving their performance. Thanks to the openness and formidable flexibility of our TP-FPU, exploration and construction of new energy-efficient embedded systems with solid FP capabilities is more feasible than ever.

Chapter 5

Transprecision FP in the High-Performance Domain

5.1 Introduction

After focussing on implementing our TP technology into embedded and ULP systems, we now focus on energy-efficient, high-performance application-class processors and beyond.

In this chapter, we demonstrate the first fully functional silicon implementation of a complete open-source TP-FPU inside a RISC-V application-class core in a 22 nm process. 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). Furthermore, there is full support for SIMD vectorization and vectorial conversions and data packing.

We further discuss other applications that have made use of our work on TP FP, made possible by our work's open-source nature.

The main contributions of this chapter are:

- Integration of FPnew (see Chapter 3) into Ariane [136], a 64-bit

application-class RISC-V processor core and subsequent silicon implementation in GLOBALFOUNDRIES 22FDX. The manufactured silicon's energy and performance measurements confirm our architecture's substantial energy proportionality and leading-edge energy efficiency. We perform a detailed breakdown of per-instruction energy cost, vectorization gains, and evaluate the voltage/frequency scaling and body biasing impact on the manufactured silicon. Our design surpasses the SOA of published FPU designs in both flexibility and efficiency (Section 5.2).

- Application case study that performs parts of a CNN inference pass on the manufactured silicon, confirming our approach's performance and energy efficiency gains on the processor level. A comparison with commercial application-class processors shows the superior effciency of our design (Section 5.2).

- Extension of the embedded TP cluster so that it can be used as a simulator on FPGAs and deployed in data centers for cloud-based development of TP software. We find that our implementation can also be used as a full co-processor to the high-performance server by leveraging heterogenous TP computing.

The remainder of this chapter is structured as follows: Section 5.2 contains the integration of FPnew into the Ariane core, the implementation in silicon, and subsequent analysis. Section 5.3 implements the embedded TP cluster as an emulator into cloud computing environments. Section 5.4 introduces some notable systems that make use of our TP-FPU. The last section provides a summary and conclusion of this chapter.

5.2 Application-Class Transprecision Computing

In order to evaluate TP computing in the high-performance general-purpose processing domain, we integrate FPnew (see Chapter 3) into the Ariane processor. Ariane is an open-source 64-bit, six stage, partially in-order RISC-V RV64GC processor [137]. It has full hardware

support for running an operating system as well as private instruction and data caches. To speed up sequential code, it features a return address stack, a branch history table, and a branch target buffer [136]. We aim at bringing a full TP computing system to silicon with this core, with support for energy-proportional computation supporting many formats.

5.2.1 Integration

ISA Extension Support

Ariane supports the RISC-V "F" and "D" standard ISA extensions, which makes the FP register file of the core 64-bit wide. We add support for FP16, FP16alt, and FP8 and 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.

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.

FPU Configuration

We configure the TP-FPU to include the aforementioned formats and add format-specific pipeline depths as shown in Table 5.1. The number of pipeline registers is set so that the processor core can achieve a clock frequency of roughly 1 GHz. $\mathtt{w}_{fpu}$ 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.

Table 5.1: The configuration of the TP-FPU as implemented into the Ariane core. The FPU width is $\mathtt{w}_{fpu} = 64$ bit.

	Implementation (number of cycles, number of lanes)			
Format	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.

Three mantissa bits are produced every clock cycle in addition to a constant three cycles for pre-/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.

5.2.2 Silicon Implementation

We implement a complete test system called Kosmodrom with the TP-enabled Ariane core in GLOBALFOUNDRIES 22FDX, and perform a detailed analysis of the per-operation energy efficiency of FP instructions.

Architecture

Kosmodrom contains three different processing engines, each tuned to tackle a particular set of FP problems. Two application-class RISC-V Ariane cores [136] take care of the general-purpose payload, and a dedicated accelerator, network training accelerator (NTX) [133], is explicitly designed for data-oblivious kernels such as Deep Neural Network training, scientific computing stencils, and general linear algebra workloads. The units share 1.25 MiB of L2 memory via a 64 bit Advanced eXtensible Interface (AXI) bus and a set of peripherals

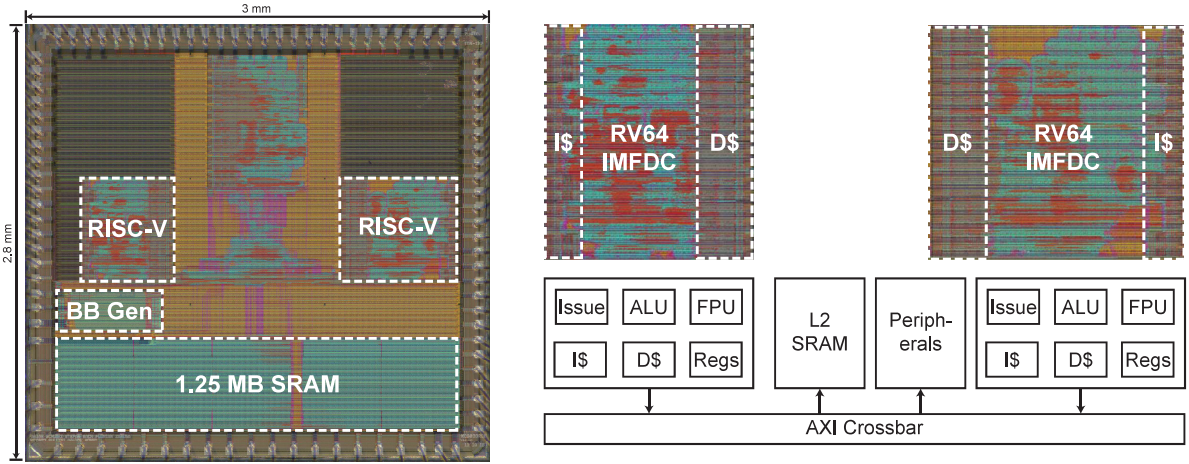


Figure 5.1: Die micrograph and block diagram of entire Kosmodrom chip showing the placement of the two Ariane core macros with TP-FPU.

such as debug infrastructure, on-chip body bias (BB) generator and a Universal Asynchronous Receiver Transmitter (UART). The high-level floorplan of the chip and a block diagram are depicted in Fig. 5.1. Each core and the NTX can be individually clocked and powered.

The Ariane cores are general-purpose (RV64GC) 64 bit, 6-stage, in-order issue, out-of-order execute RISC-V cores. Each core contains 16 KiB of instruction cache and 32 KiB of data cache and an instance of FPnew (see Chapter 3). We support five FP formats with dedicated datapaths for each one, leveraging the format-specific latencies shown in Table 5.1. The TP-FPU offers all standard RISC-V FP operations on the five formats, as well as SIMD operations for formats narrower than 64 bit.

On Kosmodrom, we provide two different flavors of the same core implemented in different cell libraries and tuned for different operating conditions:

Ariane High Performance (AHP) Tuned for *high-performance* application. The L1 caches are implemented from single-ported, high--performance static random-access memories (SRAMs) and the standard-cells used are 8-track, low, and super-low threshold voltage transistors with gate lengths of 20 nm, 24 nm and 28 nm. The nominal supply voltage is 0.8 V.

Ariane Low Power (ALP) Tuned for *light, single-threaded* applications. L1 caches are implemented from single-ported, low power SRAMs and the standard-cells used are 7.5-track, low power, low and super-low threshold voltage transistors with gate lengths of 28 nm, 32 nm and 36 nm. The nominal supply voltage is 0.5 V.

Silicon Implementation

The synthesis was performed on the design using Synopsys Design Compiler 2017.09. 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

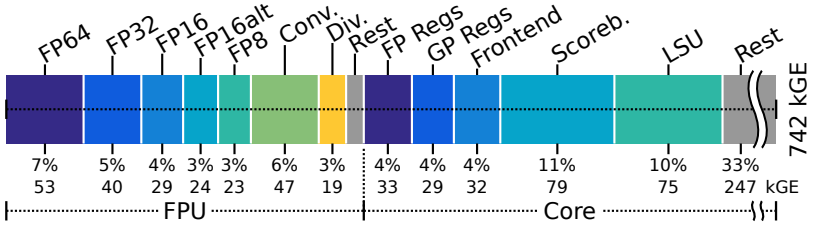


Figure 5.2: Area distribution of the entire Ariane RISC-V core, excluding cache memories (in kGE, 1 GE $\approx 0.199\,\mu m^2$).

only the handshaking tokens and the finite-state machine controlling division and square root. The pipeline registers' locations in the entire FPU were optimized using the synthesis tool's register retiming functionality.

Placement and routing are done in Cadence Innovus 17.11 with a 1 GHz constraint in a Multi-Mode Multi-Corner 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 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.

5.2.3 Implementation Results

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. 5.2. 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.

Silicon Measurements

Evaluation Methodology We extract a detailed breakdown of energy consumption within the FPU by stressing individual operations using Ariane synthetic applications. Each instruction is fed with randomly distributed normal FP values constrained so that 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 complete hierarchical visibility allows us to determine individual hardware blocks' relative energy cost contribution. 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.

FPU Instruction Energy Efficiency and Performance The top of Fig. 5.3 shows the average *per-instruction*[1] 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 multiplier's high impact (first three groups of bars). For

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

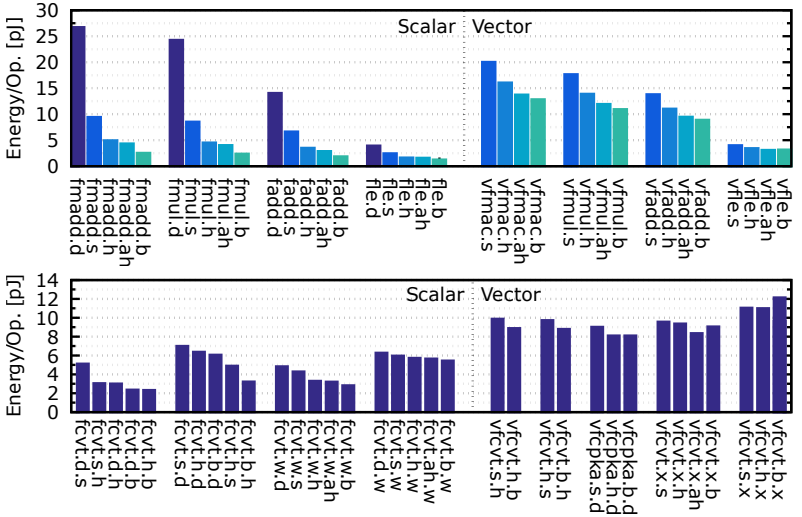


Figure 5.3: FPU energy cost per instruction for the fully pipelined scalar operations (top left), vectorial operations (top right), grouped by FMA, multiply, add, and comparison; and scalar conversion operations (bottom left) and vectorial conversion operations (bottom right).

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 compared to the next-larger format for FP16, FP16alt, and FP8, respectively. FP16alt instead of FP16 is energetically 12% cheaper due to the smaller mantissa multiplier needed for FP16. Similarly, reducing the FP format width leads to relative energy gains 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.

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 examples of less pronounced energy scaling. The bottom of Fig. 5.3 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 data toggles, padding most of the

output mantissa 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 using 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 destination vector elements 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 is not directly available in the ISA, would be required without this functionality.

Measuring scalar FP-integer conversions where the integer width is fixed also shows the relatively small relative gains, up to only 25% for FP16alt/int32 vs. 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. This impact diminishes *per-datum* energy scaling compared to the parallel slices, such as in the FP8/int8 cast.

Under nominal conditions, our TP-FPU thus achieves scalar FMA in 2.5 pJ to 26.7 pJ, SIMD FMA in 1.6 pJ to 10.0 pJ per data item, over our supported formats. FP-FP casts cost 7.0 pJ to 26.7 pJ for scalar, and 2.2 pJ to 4.9 pJ for vectorial data, respectively. Our approach of dividing the unit into parallel slices has proven to effectively achieve 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/s to 14.83 Gflop/s

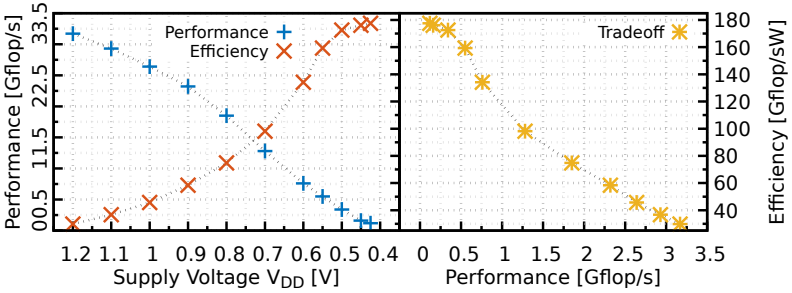


Figure 5.4: Compute performance and energy efficiency of FP64 FMA versus supply voltage (left), trade-off between compute performance and energy efficiency, achieved by adjusting supply voltage and operating frequency. Measured on manufactured silicon.

and 75 Gflop/s W to 1245 Gflop/s W for the FMA across formats.

Impact of Voltage Scaling Figure 5.4 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 V and 1.2 V. We observe peak compute and efficiency numbers of 3.17 Gflop/s and 178 Gflop/s W for FP64, 6.33 Gflop/s and 473 Gflop/s W for FP32, 12.67 Gflop/s and 1.18 Tflop/s W for FP16, 12.67 Gflop/s and 1.38 Tflop/s W for FP16alt, and 25.33 Gflop/s and 2.95 Tflop/s W for FP8.

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. Figure 5.5 shows the processor blocks' per-data energy consumption 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

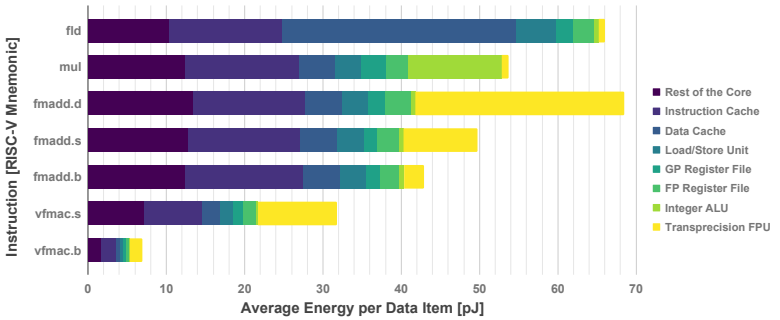


Figure 5.5: Energy cost per data item of operations in the entire Ariane core.

core energy consumption is decreased by only 38%. For small and embedded applications, scalar FPU-level energy savings might be sufficient. However, they are not enough to bring maximum savings in energy efficiency through TP in application-class cores such as Ariane due to the relatively large core-side overheads.

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

5.2.4 Application Performance Study

Reduced-precision formats are increasingly utilized in ML/DL domains, with FP16alt (bfloat16) being able to deliver almost identical accuracy to standard FP32 approaches in inference [19] and even training [91, 138]. Thus, we have performed measurements of TP-enabled convolutions, which make up most operations in these workloads on the silicon implementation of AHP in Kosmodrom.

On Ariane, we execute convolutions as performed in the layers of GoogLeNet [139], namely with a 3×3 patch size using images sizes of 56×56 from 64 input/output channels. Convolutions such as these are the most prominent and intensive workload in DL applications; hence convolutional performance will directly translate into classification

Table 5.2: 3×3-convolutions with $56 \times 56 \times 64$ outputs for deep learning in FP32, bfloat16, and FP8 performed on the AHP core in Kosmodrom.

Format	**Execution Time**		**Power**		**Energy Efficiency**	
	[Mcycle]	speedup	[mW]	diff.	[conv/sW]	rel.
FP32	304.3		51.26		59.17	1.00×
bfloat16	153.4	1.98	49.96	−2.54%	120.45	2.04×
FP8	77.9	3.91	49.76	−2.92%	238.02	4.02×

performance. The baseline implementation of the convolution on FP32 runs for over 304 Mcycle, and the resulting measurements are given in Table 5.2. Since the FPU datapath in Ariane is 64 bit wide, we use SIMD vectorization to parallelize the workload across input channels, always considering two channels concurrently. With bfloat16 and FP8, the parallelization is further increased to 4 and 8, respectively, leading to overall convolution speed-ups of 1.98 and 3.91.

The speed-up is not ideal as branching overheads, mispredictions, and output updates become relatively more noticeable with fewer innermost iterations. System power is slightly reduced by going to ever-narrower FP formats despite the entire datapath being in use, thanks to the energy-proportional operation of the TP-FPU. Finally, reduction of FP precision greatly benefits overall convolutional energy efficiency, with convolutions on bfloat16 being super-linearly energy-efficient compared to the speed-up achieved – again enabled by the energy-proportional architecture of FPnew.

As bfloat16 has become a viable choice for use in both DL classification and training [19, 91] and convolutions make up most layers in DCNNs, these results will directly translate into energy efficiency and performance gains on DL workloads on the system. Furthermore, we can offer multi-precision inference at no additional hardware cost by running the convolutions of different layers using any of the available five FP formats implemented in the system. FPnew thus also serves as an ideal evaluation and implementation platform for such applications.

Table 5.3: Key metric comparison between Ariane cores on Kosmodrom and other processors. Performance based on 32-bit flops.

	AHP [us]	**ALP** [us]	**Cortex A53**[140]	**Rocket 64b**[111]	**Tesla V100**§	**Xeon 8180**§
Node/V_{DD}	22/0.45	22/0.45	16/0.8	40/0.65	12/1.0	14/0.9
32-bit floats						
Energy Eff.†	93 ‖	98 ‖	38.7 *	16.7 ¶	122	21.9
Area Eff.‡	7.5	5.2	8.7 *	7.3 ¶	20.5	3.57
64-bit floats						
Energy Eff.†	41	44	19.4 *	16.7 ¶	61	11.0
Area Eff.‡	3.75	2.6	4.4 *	7.3 ¶	10.3	1.79

† Gflop/s W; ‡ Gflop/s mm^2 (node-scaled); § our estimates;
* assuming NEON; ¶ no SIMD ‖ extrapolated from 64 bit

5.2.5 Comparison to the State of the Art

Unit-Level Comparison

FPnew achieves industry-leading energy efficiency and energy-proportionality amongst multi-format FPUs, even if the entire TP-FPU is compared against stand-alone FMA units. This comparison has been performed in Chapter 3, using data obtained from silicon measurements of Kosmodrom (see Section 5.2.3).

Core-Level Comparison

In Table 5.3 we compare our TP-enabled Ariane cores to leading industry-strength architectures such as the ARM Cortex A53 [140] and another RISC-V open-source core called Rocket [111].

For both the AHP and the ALP we achieve higher FP energy-efficiencies. The area efficiency is slightly worse compared to Rocket as we include 32 KiB data cache compared to Rocket's 16 KiB data cache. The area difference between ALP and AHP is an artifact of the less mature cell library used for implementation of the ALP (2261 cells available vs. 7224 in the more mature AHP). Similarly, the energy efficiency of the ALP is penalized as the increased area also implies more considerable total leakage power and reduces the efficiency at

low voltages. We expect a much more evident energy efficiency offset in favor of the ALP at lower voltage in future versions of the 7.5-track library, providing similar amounts of library cells for implementation.

Furthermore, we include a Tesla V100 GPU and an Intel Xeon 8180 processor in the comparison. While we trail behind the GPU in terms of energy efficiency, it is manufactured in 12 nm while we utilize 22 nm FDX. This gap may close further when accounting for technology scaling. Notably, our system is the only one of the ones listed which exhibits super-linear energy-efficiency gains when comparing 32-bit with 64-bit FP operations.

5.3 Data Center Scale Embedded Transprecision Computing

While the title of this section might seem contradictory, a side-effect of our work on the *XwattPilot* system enables the use of embedded TP clusters (see Chapter 4) in a high-performance data center setting. The ultimate target of the XwattPilot system consists of improving the speed and energy consumption of *the development* of TP applications for embedded systems themselves. This section gives an overview of the XwattPilot system, focusing on the interaction between high-performance servers with our embedded TP clusters, made possible by leveraging open-source ecosystems. Refer to [39] for more in-depth information about intended use of this system.

5.3.1 Agile Transprecision Software Development

Energy efficiency is one of the most challenging design objectives of modern SoCs as the era of performance benefits attributed to technology node scaling is coming to an end. Typical power-aware design techniques, such as dynamic frequency and voltage scaling, are constrained by stochastic process variation effects as technology strides into the deep-submicron regime [141]. Transprecision computing is a promising form of heterogeneity in processors because it enables instructions to be executed with smaller bit widths, thus directly impacting the area and power dissipation of the corresponding circuit logic. However, developing TP software for such processors assumes a

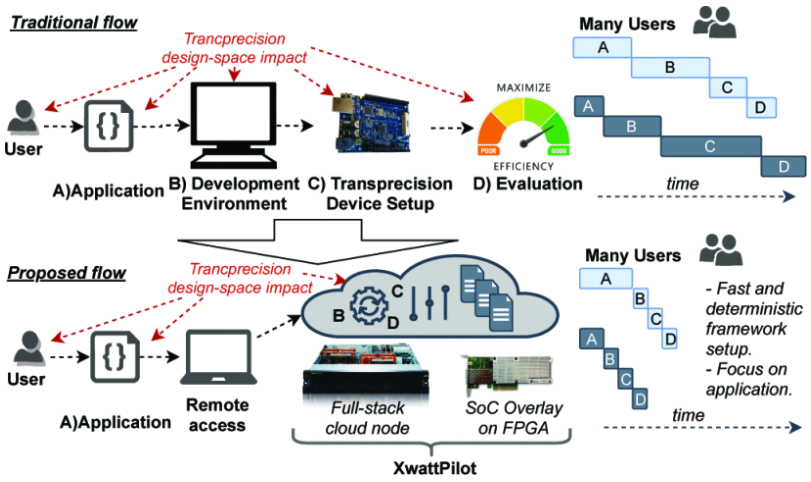


Figure 5.6: XwattPilot conceptual system leverages cloud services and FPGAs to increase low-power software productivity.

customized toolchain that spans the support of TP-enabled devices to include application support, i.e., compilers and libraries. This induced development time creates a productivity gap that has stimulated the development of new methodologies to enable the completion of TP software on schedule and within the power budget.

We propose that cloud technology be adopted to support the development of TP software in an agile way, inspired by the positive disruption of the Hardware Agile Manifesto [142]. As depicted in Fig. 5.6, the processes of customizing the development environment, bringing up the TP device, and evaluating the energy efficiency of an application are all maintained by a cloud infrastructure that allows multiple users to access shared services remotely.

5.3.2 XwattPilot Cloud System

The proposed system allows users to conduct research on TP computing by employing PULP [143], i.e. the first silicon-proven processor featuring TP extensions (see Chapter 4 and Section 5.2). XwattPilot offers the development and evaluation of TP software as a service on

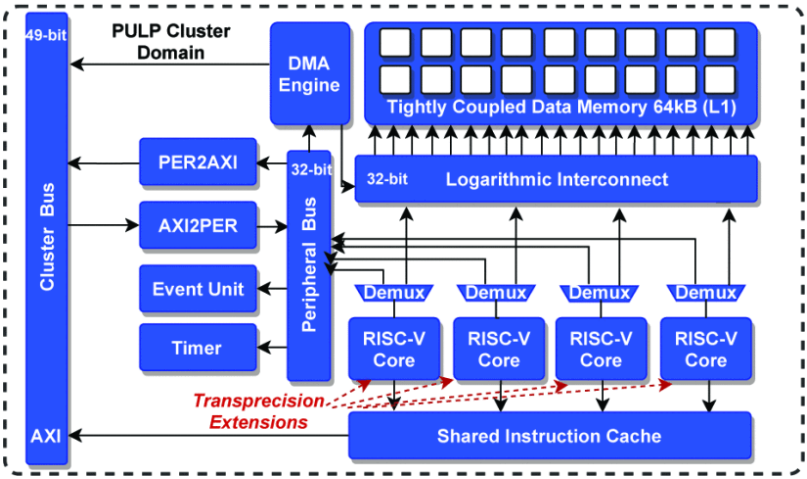


Figure 5.7: PULP cluster consists of a DMA unit, a TCDM, and one or more RISC-V processor cores extended with TP instructions.

the cloud, using *PULP FPGA*. PULP FPGA acts as a physical device emulator, where TP software can be executed on a complete PULP SoC overlaid on an FPGA device, thus delivering accurate execution cycles and power figures.

The basic computational engine present on the FPGA shown in Fig. 5.7 is a cluster with a configurable number of cores that implement the RISC-V ISA and support TP extensions, c.f. Chapter 4. A configurable number of such clusters can be assembled to instantiate a complete SoC as shown in Fig. 5.8. Off-cluster level-2 (L2) memory and peripheral accesses are managed by a tightly coupled DMA through an AXI4-compliant interconnect. We designed an adapter to PULP's AXI bus connecting to the host's main bus to enable external access through cloud services. XwattPilot is currently based on OpenPOWER systems, which include POWER™-based processors and allow coherent access to their bus, using CAPI™ technology. Hence our system has the unique advantage of making the POWER memory space visible in the PULP memory map as shown in Fig. 5.9, giving any component attached to the interconnect of PULP access to the host memory as if

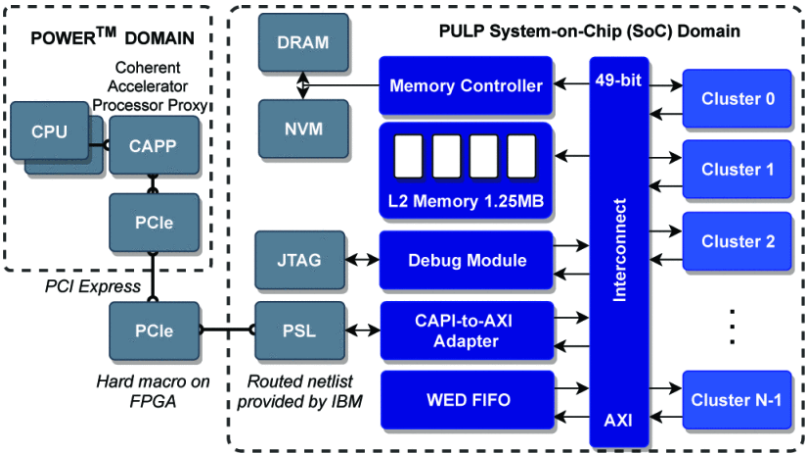


Figure 5.8: XwattPilot hardware is realized on an heterogenous node composed of a POWER™-powered system and FPGA cards used to host tranprecision PULP SoCs.

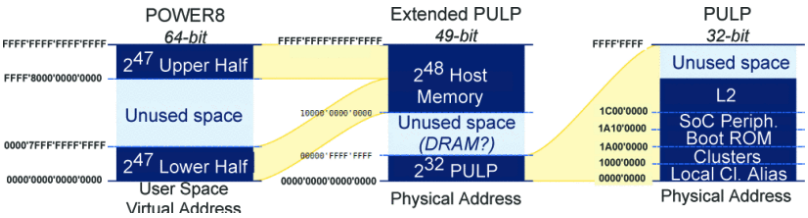


Figure 5.9: POWER™-PULP memory map scheme for PULP-FPGA.

it were a local memory. As such, bring-up time is significantly reduced, and PULP is accessible through any services running on the host, such as cloud services, for our particular requirement.

The programming model exploiting the POWER–PULP interface relies on code sections within POWER8 host code mapped to PULP cores. The handler for the overall offloading is maintained through a descriptor called "job". Jobs from POWER8 are added to a "Work Element Descriptor" (WED) FIFO-supported logic, where PULP can fetch them. PULP is awakened by an interrupt when a new job is available. The PULP RISC-V cores and the cluster peripherals are 32-bits wide. The DMA engine inside a PULP cluster is extended to support 64 bit (Fig. 5.9), and it transfers data between TCDM and L2. The intended purpose of the POWER–PULP interface presented here is the ability to test TP software on PULP cores quickly, managed by control software running on the host through cloud services. PULP FPGA is offered as a service by XwattPilot, which ports the WED interface to both C++ and VHDL for FPGA, respectively.

XwattPilot is built on an OpenStack cloud computing platform, deployed as infrastructure-as-a-service (IaaS) in public and private clouds where virtual servers and other resources are available to users. The user can instantiate multiple containerized services to test TP software on PULP FPGA.

5.3.3 Implications for High-Performance & Embedded Co-Execution

While meant as an emulation platform for embedded TP SoCs which leverages cloud infrastructure and efficient software development methodologies, our implementation can also be used as a co-processor. Both the POWER8 host and the PULP-based device have full access to each other's memory space, both systems can co-operate on heterogeneously computing tasks in the data center. While currently implemented as an FPGA emulator, the device could just as well consist of a hardened embedded multi-cluster TP SoC on an add-in card connected over PCIe. From this perspective, the current system is an emulator for a potential embedded system used as an accelerator/co-processor to high-performance computers in data centers.

5.4 Notable High-Performance Class Systems Using FPnew

Thanks to our efforts' open-source nature, several other projects have adopted FPnew, parts thereof, our ISA extensions, or entire systems for their purposes, furthering the reach of TP computing in the process. In the following, we provide an overview of a selection of notable projects using our work.

5.4.1 Snitch

Snitch [112] is a tiny open-source [144] pseudo dual-issue 32-bit RISC-V processor paired with a powerful double-precision FPU. The goal of the system is to achieve high area and energy efficiency through reducing and simplifying as much as possible the non-computational integer core area while including a very powerful FPU and keeping it fed with data. Furthermore, integer and FP instruction streams are quasi decoupled, allowing latency hiding through overlapping execution.

Snitch employs an instance of FPnew to provide FP64 and FP32 compute capabilities. It implements the SSR [145] and FREP [112] ISA extensions that offer operand streaming and hardware-loops to the FPU, achieving speed-ups of 6.45× and energy-efficiency gains of 2× over state of the art.

5.4.2 Ara

Ara [110] is a scalable RISC-V in-order vector processor implementing the 0.5 draft version of the RISC-V "V" extension. Its main goals are providing a parametric and scalable architectural template for vector processors while remaining energy-efficient throughout.

Ara uses FPnew instances to perform efficient matrix operations on up to 16 64-bit vector lanes. It achieves 2.5× the energy efficiency than the Ariane [136] RISC-V application-class core (see Section 5.2) and performs only 3% below the theoretically achievable ideally scaled peak performance.

5.4.3 European Processor Initiative

The European Processor Initiative (EPI) project is an ongoing endeavor by the European Commission whose aim is to design and implement a new family of low-power European processors for extreme-scale computing, high-performance Big-Data, and a range of emerging applications [146, 147].

Much of the FP hardware found in the EPI project is based on the open-source FPnew design. Examples include the "VaRiable Precision Processor" (VRP), "stencil/tensor accelerator" (STX) accelerators, and vector processor lanes general-purpose cores. Thanks to our work's permissive license, users of FPnew are not bound to declare or publish their derivative designs, facilitating adoption in a wide range of domains.

5.5 Summary and Conclusion

Our work on energy-proportional and flexible TP hardware and the open-source nature of our contributions has led to the adoption of FPnew in a slew of different systems. The main findings of this chapter are:

- We have implemented FPnew (see Chapter 3) as part of a RISC-V application-class processor 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/s W 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 greatly outmatches commercial application-class processors on the energy efficiency and proportionality front, thanks to the parallel datapaths approach taken in our architecture.

- We have extended the embedded TP cluster from the last chapter with the modifications necessary to be deployed as a co-processor

within large-scale data centers. Leveraging open-source frameworks, we have implemented the infrastructure to develop and run embedded TP applications in a cloud environment. Our implementation also works as a stand-alone accelerator to the POWER8 host machines. Thus, the FPGA emulation platform, or a hardened version featuring multi-cluster embedded ASICs with the appropriate connection infrastructure, could be leveraged to pursue heterogeneous TP computing in the cloud.

- We have outlined several notable works that have made of the work developed within this thesis to push the envelope of energy-proportional TP computing into new directions, including vector processors, data-streaming processors, and dedicated accelerators.

We have provided tangible evidence of our FP architecture's superb energy proportionality in a silicon-proven, fully functioning application-class processor. Furthermore, our work's broad adoption in other projects is a testament to the benefits that our work's flexibility and openness provide.

Chapter 6

Conclusions

This thesis has investigated floating-point architectures for energy-efficient transprecision computing for both embedded and high-performance domains. An initial exploration beyond the confines of IEEE 754 FP formats has shown the tremendous potential that comes from a more fine-grained FP type system. Notably, many applications can benefit from the availability of more reduced-precision types than available in the standard, as long as these types can be handled by hardware in a more performant and energy-efficient way. Thus, we developed an arbitrary-precision emulation library for exploration and a configurable open-source hardware FPU to capitalize on the tremendous potential that TP offers. With the following specification and implementation of instruction set architecture extensions and compiler support, the ingredients necessary for implanting TP computing into a vast array of systems were complete. The result is a landscape of computing architectures ranging from ultra-low power IoT nodes and embedded SoCs to application class processors and high-performance vector accelerators, all leveraging the benefits of the TP-FPU design in one way or another.

The remainder of this chapter summarizes this thesis' primary results and offers possible opportunities for future work.

6.1 Main Results

Extension of the Rigid FP Type System

A key limitation to energy savings in FP computations is the conservative use of the rigid IEEE 754 FP type system. More specifically, rarely more than one or two machine precision levels, namely *binary64* and *binary32* are available in standard general-purpose computing systems. Furthermore, programmers usually choose to perform all computations in the highest available precision out of convenience and to minimize potential inaccuracies in application results. However, this over-provisioning is not always warranted as many applications are tolerant to smaller precision and range than allotted by the standard formats, especially in intermediate computation steps, which leads to inefficiencies in terms of performance and energy consumption. To leverage transprecision (TP) as a means of recuperating these losses, the levels of precision available in a computing system must be more fine-grained and behave favorably in terms of performance and energy efficiency.

We have developed a software emulation library for arbitrary-precision FP types and used it to analyze various workloads typical in embedded near-sensor applications. Consequently, we have introduced an extended "SmallFloat" type system consisting of the IEEE 754 *binary32* (FP32) and *binary16* (FP16) types, as well as custom *binary16alt* (FP16alt) and *bianry8* (FP8) types that follow the standard principles. Having transformed the applications to make use of the new types and modeling a prototype FPU with support for these types, we observe an energy reduction potential of 18% on average and up to 30% for specific applications, over the FP32 baseline. Simultaneously, execution time is decreased by 12%, and memory accesses are reduced by 27%.

Open-Source Transprecision Floating-Point Unit

Convinced of the benefits of the extended FP type system approach we have found, we set out to implement a proper TP-FPU in hardware. As we have realized, being able to explore various custom FP types and architectural configurations is immensely valuable. As such, full

configurability in terms of formats was required, ruling out the use of already available IEEE 754 FP blocks. Furthermore, as the hardware unit's constraints are entirely dependent on its intended field of application and target technology, we needed to avoid vendor-specific IP blocks or reliance on one technology. Hence, a configurable design with control over architectural parameters was necessary. Extra features such as SIMD vectorization would be indispensable for achieving high performance and energy-efficiency of computations. Since the many requirements we had for this unit already made it so that we were not burdened by proprietary IP blocks, choosing an open-source model for the hardware we develop was a natural choice.

We have developed FPnew, a highly configurable open-source transprecision floating-point unit capable of supporting arbitrary FP formats following IEEE 754 principles. It offers FP arithmetic and efficient casting and packing operations, in both scalar and SIMD variants, with high energy efficiency and proportionality. Notably, the architecture trades off excess circuit area for improved energy efficiency. Our design achieves better energy efficiency scaling than other multi-mode FMA designs thanks to the parallel datapaths approach taken in our architecture.

Transprecision Floating-Point Extensions

TP operations must be made available to the programmer conveniently and effectively to use hardware TP-FPUs in a processing system. As we are primarily targeting programmable general-purpose RISC-V processor cores, extensions to the instruction set architecture were necessary.

We have introduced a set of extensions for the RISC-V ISA supporting the set of SmallFloat FP types formats discussed above. Adopting these formats has been proven to be highly beneficial in both HPC and embedded systems domains. We propose a complete specification for the proposed SmallFloat extensions and design and implement the compiler support in the standard RISC-V GCC compiler. On a test system using the new extensions, we achieve FP32 precision without incurring any performance overhead compared to an optimal scalar FP16 baseline, reducing system energy by 34% w.r.t. the FP32 implementation.

Transprecision in Low-Power Embedded Platforms

Armed with a fully configurable TP-FPU and matching ISA extensions, we start leveraging energy-proportional TP computing in embedded systems. We have added FPnew into the RI5CY RISC-V core and implemented the core into the PULPissimo microcontroller. Compared to a standard PULPissimo system with support for only FP32, area increases by 0.7% and static energy by 0.9%. On a system level, operating on FP32 data proves to be very affordable and averages 1.05× the energy cost of the 32-bit integer variant, achieving equal performance. Furthermore, we have described the design of a multi-core TP cluster for near-sensor computing, performing extensive design space explorations. For example, a 16-core, 16-FPU configuration provides the best performance and energy-efficiency of 5.92 Gflop/s and 167 Gflop/s W, respectively, while an 8-core, 4-FPU configuration is most area-efficient with 3.5 Gflop/s mm^2. The energy efficiency of the TP cluster outperforms all the other solutions that provide FP support in the area of embedded computing.

Our architecture found further use in an ultra-low power always-on IoT embedded platform, targeting commercialization of the TP platform we have built using FPnew.

Transprecision in Application-Class Processor Cores

We also aim at high-performance processors, implementing FPnew as part of the Ariane RISC-V core, creating the first full TP-FPU silicon implementation with support for five FP formats (FP64, FP32, FP16, FP16alt, FP8), in GLOBALFOUNDRIES 22FDX. Adaptive voltage and frequency scaling allows for energy efficiencies up to 2.95 Tflop/s W 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 dramatically outmatches commercial application-class processors on the energy efficiency and proportionality front, thanks to the parallel datapaths approach taken in our architecture.

Other projects have started utilizing FPnew for their FP purposes, as its openness and configurability offer an ideal starting point for

implementing both TP and traditional FP computing systems.

6.2 Outlook

This thesis has focused on providing an excellent base for transprecision computing by adhering to many principles of IEEE 754, reducing the burden for entry as much as possible by not abandoning tried-and-true computing principles altogether. This consistency paired with openness was undoubtedly a key to the adoption of FPnew. However, we also see the appeal of more radical proposals for FP-like arithmetic, many of them try to specifically address the inefficiencies and shortcomings of IEEE 754 FP.

The following are assorted ideas for possible extensions and evolutions of the work presented in this thesis.

Extend Architecture for More Extremely Constrained Targets While we consciously have not tried conserving circuit area to pursue higher energy efficiency, this decision makes FPnew unsuitable for extremely resource-constrained targets. Keeping such platforms in mind, extending FPnew to offer more unit types besides the (both rather large) parallel and merged configurations would further broaden the project's appeal. For example, we could offer minimum-area iterative computation units for specific formats and operations.

Offer Non-FP Types Alongside Standard Behavior The framework of FPnew's architecture with strong specialization and silencing of execution blocks is not limited to standard FP. Exploring and including alternative number formats such as LNS, unum, posit, tunable FP (TFP), amongst many others, could further benefit the TP computing ecosystem.

Hardware Generators The choice of SystemVerilog for the implementation of FPnew has guaranteed compatibility with tried-and-true hardware development flows. However, we have certainly been straining the configurability and customization level that should be done through a hardware description language alone, leading to somewhat

bloated and unsightly code. While we do not necessarily advocate for high-level synthesis, the middle ground of employing a suitable scripting tool to generate a more precise and systematic HDL representation of a specific configuration would streamline and improve the usability of FPnew.

Heterogeneous Platform with Tranprecision We have shown an initial way of combining an HPC cloud computer with embedded devices that follow a TP paradigm. Experimentation with various systems of vastly different compute capabilities, ISAs, or programming models pose an exciting field of study.

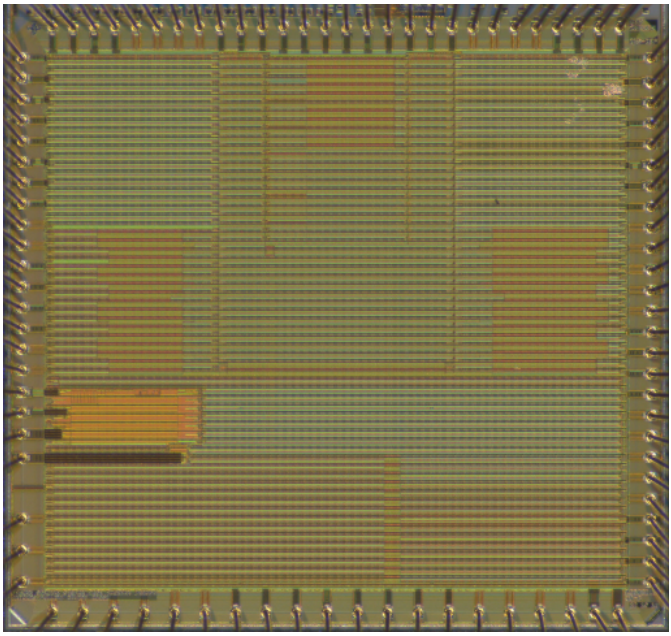
Hands-Off Transprecision Computing The instrumentation and transformation of applications for TP functionality was still a manual process in this work. More intelligent and automated ways of managing TP within the software, compilers, and hardware blocks themselves deserve additional exploration, which would lower the barrier to entry for adoption of TP computing significantly.

Appendix A

Chip Gallery

This appendix lists all chips that have been fabricated and are related to this thesis. A complete, up-to-date list of chips with the author's involvement can be found online at: `http://asic.ethz.ch/authors/Stefan_Mach.html`.

A.1 Treated in This Thesis

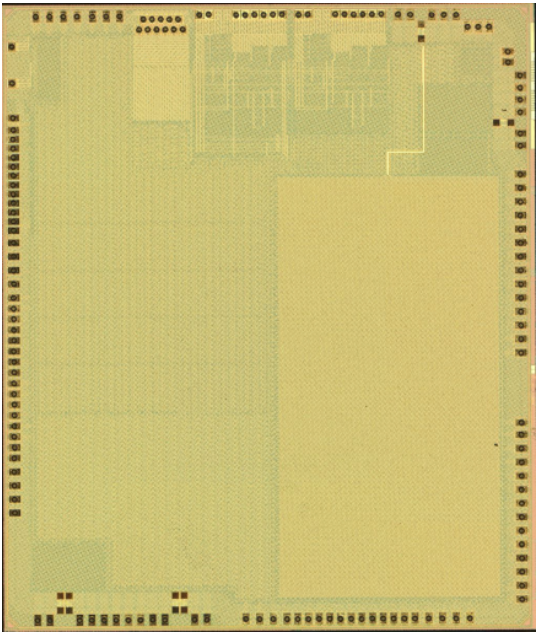


Name	Kosmodrom
Designers	Stefan Mach, Fabian Schuiki, Florian Zaruba
Application / Publication	Application CPU / Research Project [34, 43]
Technology / Package	GF22FDX / QFN56
Dimensions	3000 µm × 3000 µm
Voltage	0.8 V
Clock	1300 MHz

Kosmodrom is a direct collaboration with Globalfoundries for evaluating different library options of the GF22FDX process using a realistic benchmark. For this purpose, there are two 64-bit RISC-V-based Ariane cores with TP-FPUs. Both cores support 5 FP formats, including SIMD support for all sub-64-bit formats.

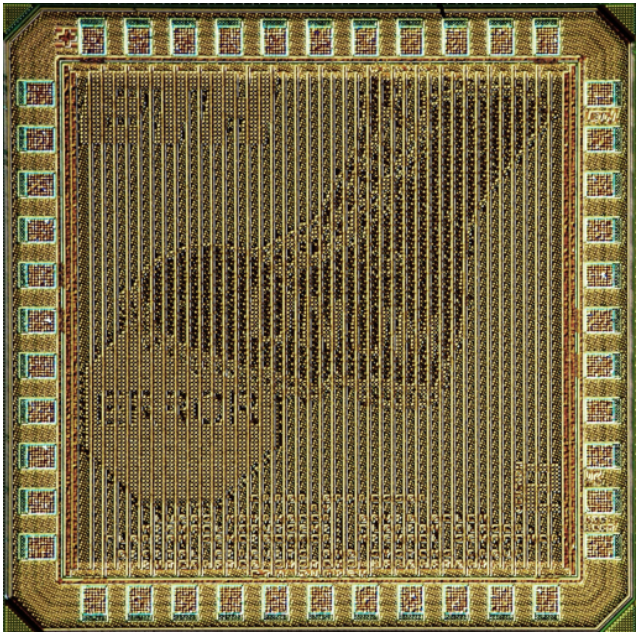
Name	Baikonur
Designers	Stefan Mach, Fabian Schuiki, Florian Zaruba
Application / Publication	Application CPU / Research Project [148]
Technology / Package	GF22FDX / QFN56
Dimensions	3000 µm × 3000 µm
Voltage	0.8 V
Clock	1000 MHz

Baikonur continues our direct collaboration with Globalfoundries. The two Ariane cores return with fixes and updates to the TP-FPU blocks. Furthermore, there is a many-core architecture consisting of 3 clusters, each containing 8 Snitch cores. Snitch cores are small 32bit RISC-V (RV32IMAFD) cores with a tiny integer unit but a very powerful vectorizable 64-bit FPU and extensions for stream processing. All FP functionality in the design is provided by implementations of FPnew.



Name	Vega
Designers	Stefan Mach, Davide Rossi, Francesco Conti, Manuel Eggimann, Alfio Di Mauro, Marco Guermandi, Giuseppe Tagliavini, Antonio Pullini, Igor Loi, Jie Chen, Eric Flamand
Application / Publication	IoT End-Node SoC / Industrial [41]
Technology / Package	GF22FDX / BGA169
Dimensions	4000 µm × 3000 µm
Voltage	0.5 V to 0.8 V
Clock	450 MHz

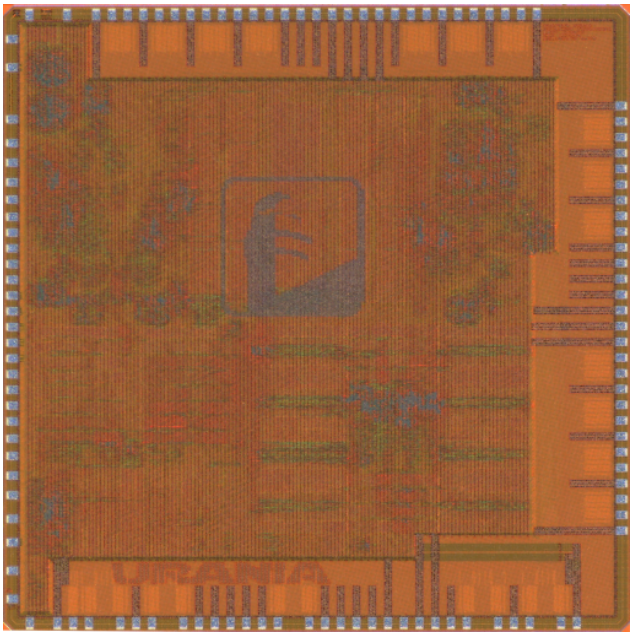
Vega is an always-on IoT end-node SoC capable of scaling from a 1.7 µW fully retentive cognitive sleep mode up to 32.2 Gop/s (at 49.4 mW) peak performance on NSAAs, including mobile DNN inference. The SoC features 10 RISC-V cores: one core for SoC and IO management and a 9-core cluster supporting multi-precision SIMD integer and FP computation, paired with two ML accelerators.



Name	Unum
Designers	Lucas Mayrhofer, David Oelen, Michael Gautschi, Florian Glaser, Florian Scheidegger Michael Schaffner
Application / Publication	Number Systems / Student Project [40]
Technology / Package	UMC65 / QFN40
Dimensions	1252 µm × 1252 µm
Voltage	1.2 V
Clock	400 MHz

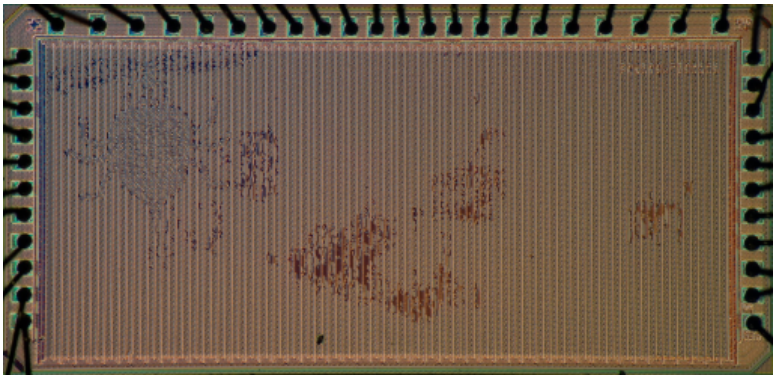
This chip implements the unum number format developed by Dr. John Gustafson. The chip contains an ALU that accepts two unum numbers with a tag that has 4bit exponent and 5bit mantissa (unum45), comprising a ubound. It can add/subtract these numbers and calculate an optimized bound for the result.

A.2 Further Implementations of FPnew



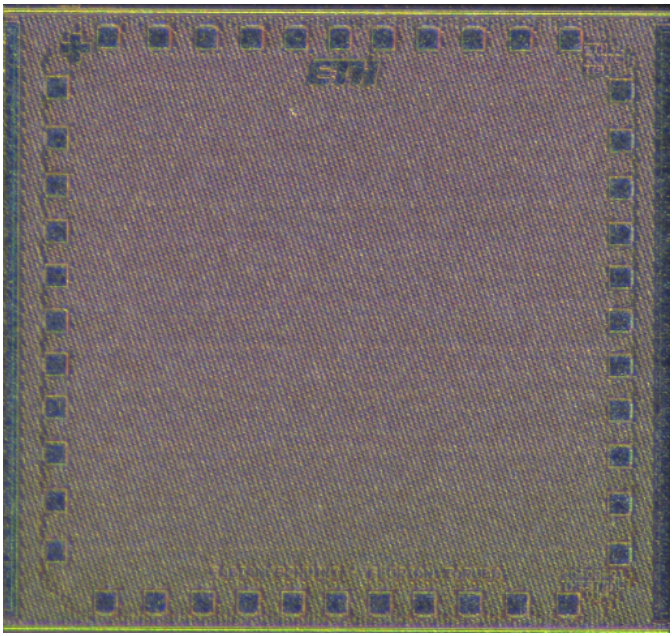
Name	Urania
Designers	Andreas Kurth, Wolfgang Roenninger, Oscar F. Castaneda, Christian Weis, Jan Lappas, Chirag Sudarshan, Beat Muheim
Application / Publication	Heterogeneous Computing / Research Project
Technology / Package	UMC65 / QFN64
Dimensions	4000 µm × 4000 µm
Voltage	1.2 V
Clock	100 MHz

Urania is the first ASIC implementation of a multi-cluster RISC-V architecture for the HERO project. It contains a 64-bit Ariane core and two clusters of four RI5CY cores, each with individual TP-FPUs.



Name	Billywig
Designers	Florian Zaruba, Fabian Schuiki, Beat Muheim
Application / Publication	HPC / Research Project [112]
Technology / Package	UMC65 / QFN40
Dimensions	2626 µm × 1252 µm
Voltage	1.2 V
Clock	350 MHz

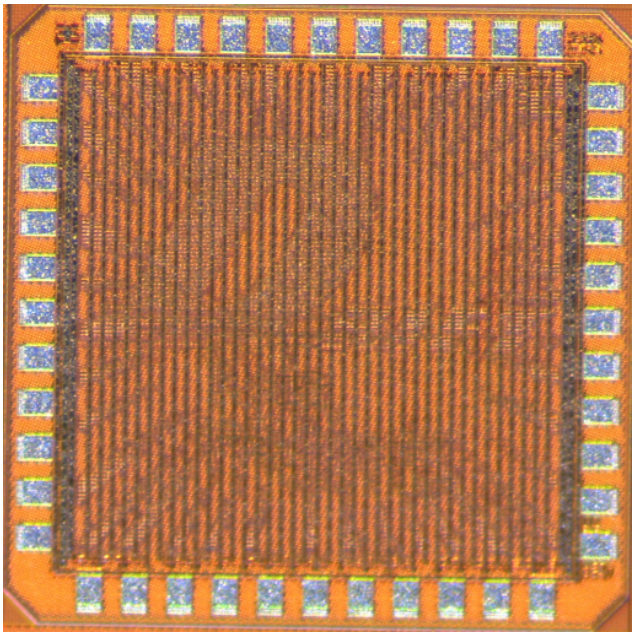
Billywig is a RISC-V-based multicore system for processing regular data structures. It contains four ultra-small RV32IMAFD Snitch cores with custom extensions to improve stream processing. This small core is then paired with a fairly large 64-bit FPU (with 64 64-bit registers), which can be used as a single 64-bit FPU or 2× 32-bit FPUs in parallel. The instruction extensions can be used to program stream processing allowing the FPU to process data directly from memory, giving the core a pseudo-dual-issue capability. While the FPU is active, the integer core can continue to execute code, in particular, to update the configuration of the streaming subsystem.



Name	Thestral
Designers	Fabian Schuiki, Florian Zaruba, Thomas Benz, Paul Scheffler, Wolfgang Roenninger
Application / Publication	FP Power Management / Research Project
Technology / Package	GF22FDX / QFN40
Dimensions	1250 µm × 1250 µm
Voltage	0.8 V
Clock	650 MHz

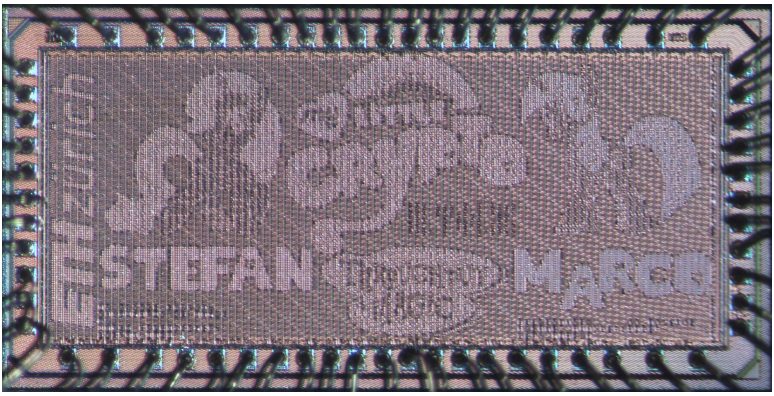
Thestral contains architectural improvements to Snitch and extensive power gating infrastructure. Notable changes compared to the system in Baikonur are 20 different power domains (each FPU/IPU, as well as the cluster, can be individually powered down), and double-pumped FPUs and IPUs. The compute units (and the TCDM) can be operated at twice the integer core speed, allowing for higher peak throughput.

A.3 Miscellaneous ASICs



Name	Drift
Designers	Stefan Mach, Wolfgang Roenninger, Kaja Jentner, Florian Zaruba
Application / Publication	Integer Dividers / Student Project
Technology / Package	UMC65 / QFN40
Dimensions	1252 µm × 1252 µm
Voltage	1.2 V
Clock	600 MHz

Drift implements several different integer divider architectures that can be used as part of the 64-bit RISC-V core Ariane. The name stands for **D**ivision with different **R**adix representations of **I**ntegers using **F**ractional **T**ransformation.



Name	Pony
Designers	Stefan Mach, Marco Eppenberger, Cyril Arnould, Michael Muehlberghuber, Beat Muheim
Application / Publication	Cryptography / Student Project
Technology / Package	UMC65 / QFN56
Dimensions	2626 µm × 1252 µm
Voltage	1.2 V
Clock	600 MHz

This chip contains five candidates for the CAESAR competition to determine an Authenticated Encryption with Associated Data (AEAD) standard. All implementations have been geared towards a 100 Gb/s throughput. For testing purposes, the top-level design of the chip selectively allows feeding candidates from an externally accessible on-chip RAM or pseudo-randomly generated data. Furthermore, clock-gating is used for exact power measurements. The full name of the chip is *My Little Crypto: Throughput is Magic* and it is the most beautiful chip ever taped out.

Appendix B

Acronyms

AHP Ariane High Performance.

ALP Ariane Low Power.

ASIC application-specific integrated circuit.

AXI Advanced eXtensible Interface.

BB body bias.

EPI European Processor Initiative.

FMA fused multiply-add.

FP floating-point.

FPGA field-programmable gate array.

FPU floating-point unit.

HAL hardware abstraction layer.

HPC High Performance Computing.

IoT Internet of Things.

ISA instruction set architecture.

LNS logarithmic nubmer system.

MAC multiply-accumulate.

MCU microcontroller unit.

ML machine learning.

NaN not a number.

NaR not a real.

NTX network training accelerator.

P&R place & route.

SIMD single instruction multiple data.

SoC system on a chip.

SQNR signal-to-quantization-noise ratio.

SRAM static random-access memory.

TDP Thermal Design Power.

TP transprecision.

TP-FPU transprecision floating-point unit.

UART Universal Asynchronous Receiver Transmitter.

ULP ultra-low power.

unum universal number.

Bibliography

[1] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[2] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong, "Device scaling limits of Si MOSFETs and their application dependencies," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 259–288, 2001.

[3] M. Bohr, "A 30 year retrospective on Dennard's MOSFET scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[4] J. J. Dongarra, H. W. Meuer, E. Strohmaier *et al.*, "TOP500 supercomputer sites," *Supercomputer*, vol. 13, pp. 89–111, 1997.

[5] W.-c. Feng and K. Cameron, "The Green500 list: Encouraging sustainable supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, 2007.

[6] C. Hall, "Keeping cool and cutting costs in the data center," https://www.datacenterknowledge.com/power-and-cooling/keeping-cool-and-cutting-costs-data-center, Mar 2018.

[7] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[8] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[9] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 522–531.

[10] W. Kahan, J. D. Darcy, E. Eng, and H.-P. N. Computing, "How Java's floating-point hurts everyone everywhere," in *ACM 1998 workshop on java for high-performance network computing*. Stanford University, 1998, p. 81.

[11] B. Barrois and O. Sentieys, "Customizing fixed-point and floating-point arithmetic – a case study in k-means clustering," in *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2017, pp. 1–6.

[12] A. Volkova, T. Hilaire, and C. Lauter, "Arithmetic approaches for rigorous design of reliable fixed-point LTI filters," *IEEE Transactions on Computers*, vol. 69, no. 4, pp. 489–504, 2020.

[13] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.

[14] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[15] M. Zanghieri, S. Benatti, A. Burrello, V. Kartsch, F. Conti, and L. Benini, "Robust real-time embedded emg recognition framework using temporal convolutional networks on a multicore IoT processor," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 14, no. 2, pp. 244–256, 2020.

[16] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[17] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[18] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International conference on machine learning*, 2016, pp. 2849–2858.

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

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

[21] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in neural information processing systems*, 2017, pp. 1742–1752.

[22] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving DNNs in real time at datacenter scale with project Brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017.

[24] N. Amit, M. Wei, and C.-C. Tu, "Extreme datacenter specialization for planet-scale computing: ASIC clouds," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 96–108, 2018.

[25] D. Mukunoki and T. Imamura, "Reduced-precision floating-point formats on GPUs for high performance and energy-efficient computation," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 144–145.

[26] A. Waterman and K. Asanovic, "The RISC-V instruction set manual, volume I: User-level ISA, document version 20191214-draft," RISC-V Foundation, 2019.

[27] PULP Platform, "The parallel ultra low power (PULP) platform," https://pulp-platform.org, 2021.

[28] Berkeley Architecture Research, "The Berkeley out-of-order RISC-V processor," https://boom-core.org, 2020.

[29] Reconfigurable Intelligent Systems Engineering (RISE), IIT-Madras, "Shakti open source processor development ecosystem," http://shakti.org.in, 2021.

[30] Western Digital Corporation, "EH1 RISC-V SweRV CoreTM 1.9 from Western Digital," https://github.com/chipsalliance/Cores-SweRV, 2021.

[31] RISC-V International, "RISC-V Exchange," https://riscv.org/exchange, 2021.

[32] PULP Platform, "FPnew – new floating-point unit with transprecision capabilities," https://github.com/pulp-platform/fpnew, 2021.

[33] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1051–1056.

[34] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2020.

[35] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "Design and evaluation of smallFloat SIMD extensions to the RISC-V ISA," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 654–657.

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

[37] F. Montagna, S. Mach, S. Benatti, A. Garofalo, G. Ottavi, L. Benini, D. Rossi, and G. Tagliavini, "A transprecision floating-point cluster for efficient near-sensor data analytics," *arXiv preprint arXiv:2008.12243*, 2020.

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

[39] D. Diamantopoulos, F. Scheidegger, S. Mach, F. Schuiki, G. Haugou, M. Schaffner, F. K. Gürkaynak, C. Hagleitner, A. C. I. Malossi, and L. Benini, "Xwattpilot: A full-stack cloud system enabling agile development of transprecision software for low-power SoCs," in *2020 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*. IEEE, 2020, pp. 1–3.

[40] F. Glaser, S. Mach, A. Rahimi, F. K. Gürkaynak, Q. Huang, and L. Benini, "An 826 Mops, 210 uW/MHz Unum ALU in 65 nm," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5.

[41] D. Rossi, F. Conti, M. Eggiman, S. Mach, A. Di Mauro, M. Guermandi, G. Tagliavini, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "A 1.3 Tops/W @ 32 Gops fully integrated 10-core SoC for IoT end-nodes with 1.7 uW cognitive wake-up from MRAM-based state-retentive sleep mode," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. IEEE, Feb. 2021.

[42] F. Schuiki, F. Zaruba, S. Mach, and L. Benini, "Kosmodrom: Energy-efficient ariane cores with transprecision FPU in 22nm," in *RISC-V Workshop Zurich*, 2019.

[43] F. Zaruba, F. Schuiki, S. Mach, and L. Benini, "The floating point trinity: A multi-modal approach to extreme energyefficiency and performance," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2019, pp. 767–770.

[44] A. Pullini, M. Gautschi, F. K. Gürkaynak, F. Glaser, S. Mach, G. Rovere, D. Schiavone, G. Haugou, D. Palossi, A. Marongiu *et al.*, "KISS PULPino – updates on PULPino," in *5th RISC-V Workshop*. ETH Zürich, 2016.

[45] A. Pullini, S. Mach, M. Magno, and L. Benini, "A dual processor energy-efficient platform with multi-core accelerator for smart sensing," in *International Conference on Sensor Systems and Software*. Springer, 2016, pp. 29–40.

[46] M. Eggimann, S. Mach, M. Magno, and L. Benini, "A RISC-V based open hardware platform for always-on wearable smart sensing," in *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*. IEEE, 2019, pp. 169–174.

[47] A. Di Mauro, F. Zaruba, F. Schuiki, S. Mach, and L. Benini, "Live demonstration: Exploiting body-biasing for static corner trimming and maximum energy efficiency operation in 22nm FDX technology," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–1.

[48] H. Müller, D. Palossi, S. Mach, F. Conti, and L. Benini, "Fünfliber-drone: A modular open-platform 18-grams autonomous nano-drone," in *Design, Automation Test in Europe Conference Exhibition (DATE 2021)*, 2021, pp. 12–11.

[49] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[50] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[51] C. Bekas, A. Curioni, and I. Fedulova, "Low-cost data uncertainty quantification," *Concurrency and Computation: Pract. & Exper.*, vol. 24, no. 8, pp. 908–920, 2012.

[52] P. Klavík, A. C. I. Malossi, C. Bekas, and A. Curioni, "Changing computing paradigms towards power efficiency," *Phil. Trans. R. Soc. A*, vol. 372, no. 2018, 2014.

[53] N.-M. Ho, E. Manogaran, W.-F. Wong, and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," in *22nd Asia and South Pacific Design Automation Conf. (ASP-DAC)*. IEEE, 2017, pp. 63–68.

[54] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 2017, pp. 300–315.

[55] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 12:1–12:41, May 2008.

[56] J. L. Gustafson, "A radical approach to computation with real numbers," *Supercomputing Frontiers and Innovations*, vol. 3, no. 2, 2016.

[57] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhäuser Boston, 2018, ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9.

[58] M. Bista, "Image annotation using ZYNQ SoC," Ph.D. dissertation, TRIBHUVAN UNIVERSITY, 2017.

[59] M. M. Trompouki and L. Kosmidis, "Towards general purpose computations on low-end mobile GPUs," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*. IEEE, 2016, pp. 539–542.

[60] D. H. Bailey, H. Yozo, X. S. Li, and B. Thompson, "ARPREC: An arbitrary precision computation package," *Lawrence Berkeley National Laboratory*, 2002.

[61] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, p. 13, 2007.

[62] J. R. Hauser, "Handling floating-point exceptions in numeric programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 2, pp. 139–174, 1996.

[63] H. Brönnimann, G. Melquiond, and S. Pion, "The design of the Boost interval arithmetic library," *Theoretical Computer Science*, vol. 351, no. 1, pp. 111–118, 2006.

[64] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with discrete stochastic arithmetic," https://hal.archives-ouvertes.fr/hal-01331917, 2016.

[65] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 27.

[66] M. Moscato, L. Titolo, A. Dutle, and C. A. Munoz, "Automatic estimation of verified floating-point round-off errors via static analysis," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2017, pp. 213–229.

[67] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar, "A 1.45 GHz 52-to-162 Gflops/W variable-precision floating-point fused multiply-add unit

with certainty tracking in 32nm CMOS," in *2012 IEEE International Solid-State Circuits Conference*. IEEE, 2012, pp. 182–184.

[68] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 273–286, 2000.

[69] T. Rzayev, S. Moradi, D. H. Albonesi, and R. Manchar, "DeepRecon: Dynamically reconfigurable architecture for accelerating deep neural networks," in *International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 116–124.

[70] M. Gautschi, M. Schaffner, F. K. Gürkaynak, and L. Benini, "An extended shared logarithmic unit for nmiscar function kernel acceleration in a 65-nm CMOS multicore cluster," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 98–112, 2017.

[71] W. Tichy, "The end of (numeric) error: An interview with John L. Gustafson," *Ubiquity*, vol. 2016, no. April, pp. 1:1–1:14, Apr. 2016.

[72] "Unum arithmetic in Julia," https://github.com/JuliaComputing/Unums.jl, 2017.

[73] M. Kvasnica, "munum: Matlab(R) library for universal numbers," https://bitbucket.org/kvasnica/munum, 2017.

[74] J. Muizelaar, "Python port of the Mathematica Unum prototype from 'The End of Error'," https://github.com/jrmuizel/pyunum, 2017.

[75] A. Bocco, Y. Durand, and F. de Dinechin, "Hardware support for Unum floating point arithmetic," in *2017 13th Conference on Ph. D. Research in Microelectronics and Electronics (PRIME)*. IEEE, 2017, pp. 93–96.

[76] J. Hou, Y. Zhu, Y. Shen, M. Li, Q. Wu, and H. Wu, "Enhancing precision and bandwidth in cloud computing: Implementation of a novel floating-point format on FPGA," in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, June 2017, pp. 310–315.

[77] J. L. Gustafson, *The End of Error: Unum Computing.* CRC Press, 2017.

[78] J. L. Gustafson, "Posit arithmetic," *Mathematica Notebook describing the posit number system*, vol. 30, 2017.

[79] R. Kirchner and U. Kulisch, "Accurate arithmetic for vector processors," *Journal of parallel and distributed computing*, vol. 5, no. 3, pp. 250–270, 1988.

[80] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM Journal of research and development*, vol. 34, no. 1, pp. 59–70, 1990.

[81] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012.* IEEE, 2012, pp. 1212–1221.

[82] S. Mach, "The smallFloat extensions for RISC-V," https://iis-git.ee.ethz.ch/smach/smallFloat-spec/blob/v0.5/smallFloat_isa.pdf, 2018.

[83] "Auto-vectorization in GCC," https://www.gnu.org/software/gcc/projects/tree-ssa/vectorization.html, 2018.

[84] R. Usselmann, "Floating point unit," https://opencores.org/projects/fpu, 2018.

[85] J. Al-Eryani, "FPU," https://opencores.org/projects/fpu100, 2017.

[86] F. de Dinechin and B. Pasca, "Designing custom arithmetic data pathwith FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[87] B. A. Research, "Berkeley hardware floating-point units," https://github.com/ucb-bar/berkeley-hardfloat, 2020.

[88] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The Rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[89] K. Asanovic, D. A. Patterson, and C. Celio, "The Berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[90] F. Kaiser, S. Kosnac, and U. Brüning, "Development of a RISC-V-conform fused multiply-add floating-point unit," *Supercomputing Frontiers and Innovations*, vol. 6, no. 2, pp. 64–74, 2019.

[91] P. Zamirai, J. Zhang, C. R. Aberger, and C. De Sa, "Revisiting bfloat16 training," *arXiv preprint arXiv:2010.06192*, 2020.

[92] 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 *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 662–667.

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

[94] M. Gautschi, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A 65nm CMOS 6.4-to-29.2 pJ/flop @ 0.8 V shared logarithmic floating point unit for acceleration of nmiscar function kernels in a tightly coupled processor cluster," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 82–83.

[95] A. Nannarelli, "Tunable floating-point adder," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1553–1560, 2019.

[96] K. Manolopoulos, D. Reisis, and V. A. Chouliaras, "An efficient dual-mode floating-point multiply-add fused unit," in *2010 17th IEEE International Conference on Electronics, Circuits and Systems*. IEEE, 2010, pp. 5–8.

[97] V. Arunachalam, A. N. J. Raj, N. Hampannavar, and C. Bidul, "Efficient dual-precision floating-point fused-multiply-add architecture," *Microprocessors and Microsystems*, vol. 57, pp. 23–31, 2018.

[98] H. Zhang, D. Chen, and S. Ko, "Efficient multiple-precision floating-point fused multiply-add with mixed-precision support," *IEEE Transactions on Computers*, 2019.

[99] J. Pu, S. Galal, X. Yang, O. Shacham, and M. Horowitz, "FPMax: a 106 Gflops/W at 217 Gflops/mm2 single-precision FPU, and a 43.7 Gflops/W at 74.6 Gflops/mm2 double-precision FPU, in 28nm UTBB FDSOI," *arXiv preprint arXiv:1606.07852*, 2016.

[100] T. M. Bruintjes, K. H. Walters, S. H. Gerez, B. Molenkamp, and G. J. Smit, "Sabrewing: A lightweight architecture for combined floating-point and integer arithmetic," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–22, 2012.

[101] PULP Platform, "PULPissimo," https://github.com/pulp-platform/pulpissimo, 2021.

[102] PULP Platform, "PULPino," https://github.com/pulp-platform/pulpino, 2021.

[103] OpenHW Group, "OpenHW Group CORE-V CV32E40P RISC-V IP," https://github.com/openhwgroup/cv32e40p, 2021.

[104] OpenHW Group, "OpenHW Group Website," https://www.openhwgroup.org, 2021.

[105] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr. Wolf: An energy-precision scalable parallel ultra low power SoC for IoT edge processing," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, 2019.

[106] F. Glaser, G. Tagliavini, D. Rossi, G. Haugou, Q. Huang, and L. Benini, "Energy-efficient hardware-accelerated synchronization for shared-L1-memory multiprocessor clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 633–648, 2021.

[107] G. Tagliavini, D. Cesarini, and A. Marongiu, "Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2150–2163, 2018.

[108] F. Montagna, M. Buiatti, S. Benatti, D. Rossi, E. Farella, and L. Benini, "A machine learning approach for automated wide-range frequency tagging analysis in embedded neuromonitoring systems," *Methods*, vol. 129, pp. 96–107, 2017.

[109] J. Robelly, G. Cichon, H. Seidel, and G. Fettweis, "Implementation of recursive digital filters into vector SIMD DSP architectures," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5. IEEE, 2004, pp. V–165.

[110] 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 Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[111] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3 GHz 16.7 double-precision Gflops/W RISC-V processor with vector accelerators," in *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*. IEEE, 2014, pp. 199–202.

[112] 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 Transactions on Computers*, 2020.

[113] A. Bocco, Y. Durand, and F. De Dinechin, "SMURF: Scalar multiple-precision Unum RISC-V floating-point accelerator for scientific computing," in *Proceedings of the Conference for Next Generation Arithmetic 2019*, 2019, pp. 1–8.

[114] M. K. Jaiswal, B. S. C. Varma, H. K. . So, M. Balakrishnan, K. Paul, and R. C. C. Cheung, "Configurable architectures for

multi-mode floating point adders," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 8, pp. 2079–2090, 2015.

[115] J. D. Bruguera, "Low latency floating-point division and square root unit," *IEEE Transactions on Computers*, vol. 69, no. 2, pp. 274–287, 2019.

[116] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of SIMD vector operations to accelerate application code performance on low-powered Arm and Intel platforms," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1107–1116.

[117] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture," in *International Workshop on OpenMP*. Springer, 2014, pp. 202–214.

[118] RISC-V Foundation, "RISC-V "P" extension specification," https://github.com/riscv/riscv-p-spec, 2021.

[119] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[120] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The Arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[121] T. Yoshida, "Fujitsu high performance CPU for the Post-K computer," in *Hot Chips*, vol. 30, 2018.

[122] RISC-V Foundation, "RISC-V "V" extension specification," https://github.com/riscv/riscv-v-spec, 2021.

[123] F. Johansson, "Arb: efficient arbitrary-precision midpoint-radius interval arithmetic," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1281–1292, 2017.

[124] V. Lefèvre, "Correctly rounded arbitrary-precision floating-point summation," *IEEE Transactions on Computers*, vol. 66, no. 12, pp. 2111–2124, 2017.

[125] A. Anderson, S. Muralidharan, and D. Gregg, "Efficient multi-byte floating point data formats using vectorization," *IEEE Transactions on Computers*, vol. 66, no. 12, pp. 2081–2096, 2017.

[126] M. Brandalero, L. Carro, A. C. S. Beck Filho, and M. Shafique, "Multi-target adaptive reconfigurable acceleration for low-power IoT processing," *IEEE Transactions on Computers*, 2020.

[127] R. Prasad, S. Das, K. Martin, G. Tagliavini, P. Coussy, L. Benini, and D. Rossi, "TRANSPIRE: An energy-efficient TRANSprecision floating-point Programmable archItectuRE," in *Design, Automation and Test in Europe Conference (DATE)*, 2020.

[128] "Arm helium technology," https://www.arm.com/why-arm/technologies/helium, 2020.

[129] "Arm cortex m55 processor," https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m55, 2020.

[130] N.-M. Ho and W.-F. Wong, "Exploiting half precision arithmetic in Nvidia GPUs," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[131] S. Eliuk, C. Upright, and A. Skjellum, "dMath: A scalable linear algebra and math library for heterogeneous GP-GPU architectures," *arXiv preprint arXiv:1604.01416*, 2016.

[132] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy-efficient deep learning with smart memory cubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 420–434, 2017.

[133] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 484–497, 2018.

[134] D. Bol, M. Schramme, L. Moreau, T. Haine, P. Xu, C. Frenkel, R. Dekimpe, F. Stas, and D. Flandre, "A 40-to-80 MHz sub-4 μW/MHz ULV Cortex-M0 MCU SoC in 28nm FDSOI with dual-loop adaptive back-bias generator for 20 μs wake-up from deep fully retentive sleep mode," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 322–324.

[135] GreenWaves Technologies, "GreenWaves unveils groundbreaking ultra-low power GAP9 IoT application processor for the next wave of intelligence at the very edge," https://greenwaves-technologies.com/gap9_iot_application_processor, 2019.

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

[137] OpenHW Group, "CVA6 RISC-V CPU," https://github.com/openhwgroup/cva6, 2021.

[138] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[139] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[140] S. Shibahara *et al.*, "A 16 nm FinFET heterogeneous nona-core SoC supporting ISO26262 ASIL B standard," *IEEE JSSC*, vol. 52, no. 1, pp. 77–88, 2017.

[141] A. Zjajo, "Random process variation in deep-submicron CMOS," in *Stochastic Process Variation in Deep-Submicron CMOS*. Springer, 2014, pp. 17–54.

[142] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic *et al.*, "An agile approach to building RISC-V microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[143] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beignè *et al.*, "Energy-efficient near-threshold parallel computing: The PULPv2 cluster," *Ieee Micro*, vol. 37, no. 5, pp. 20–31, 2017.

[144] PULP Platform, "Snitch System," https://github.com/pulp-platform/snitch, 2021.

[145] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight RISC-V ISA extension achieving full compute utilization in single-issue cores," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 212–227, 2020.

[146] 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*. Springer, 2019, pp. 185–195.

[147] "European Processor Initiative," https://www.european-processor-initiative.eu, 2021.

[148] F. Zaruba, F. Schuiki, and L. Benini, "Manticore: A 4096-core RISC-V chiplet architecture for ultra-efficient floating-point computing," *IEEE Micro*, 2020.

Curriculum Vitae

Stefan Mach was born on 26th September 1990 in Lucerne, Switzerland. He received his BSc degree from ETH Zurich in 2014 and his MSc degree from ETH Zurich in 2016. He joined the Integrated Systems Laboratory of ETH Zurich as a Ph.D. candidate in 2016 under Prof. Dr. Luca Benini's supervision. His research interests include transprecision computing, computer arithmetics, and energy-efficient processor architectures.