ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

INGEGNERIA ELETTRONICA, TELECOMUNICAZIONI E TECNOLOGIE DELL'INFORMAZIONE

Ciclo 38

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

SECURE RISC-V ARCHITECTURES FOR BRAIN-INSPIRED COMPUTING

Presentata da: Simone Manoni

Coordinatore Dottorato

Davide Dardari

Supervisore

Andrea Bartolini

Co-supervisore

Luca Benini

Esame finale anno 2026

ABSTRACT

Modern Cyber-Physical Systems (CPS) demand high computational capability, strong security guarantees, and energy-aware operation, particularly in embedded, mobile, and autonomous platforms such as robotics, intelligent vehicles, and industrial controllers. Addressing these requirements calls for heterogeneous System-on-Chip (SoC) architectures that balance performance, efficiency, and security while supporting flexible on-device artificial intelligence (AI) execution. This thesis proposes a heterogeneous, high-performance RISC-V–based architectural template integrating a general-purpose host processor, a programmable acceleration cluster, and a Root-of-Trust (RoT), providing the system context for the design and evaluation of security and AI acceleration mechanisms.

To strengthen runtime security, the thesis investigates two hardware-assisted Control-Flow Integrity (CFI) approaches for RISC-V systems. The first introduces CVA6-CFI, the first hardware-extended application-class core implementing the recently ratified `Zicfiss` and `Zicfilp` specifications, achieving robust forward- and backward-edge protection with only 1.0% area overhead and up to 15.6% performance overhead on the MiBench automotive subset. The second approach leverages the OpenTitan RoT as a firmware-based CFI co-processor, enforcing control-flow policies externally to the host pipeline with minimal area cost and less than 10% execution

overhead. Together, these approaches represent complementary design points for secure RISC-V SoCs.

Among emerging on-device AI paradigms, brain-inspired spiking neural networks (SNNs) offer compact and event-driven computation. To avoid reliance on fixed-function accelerators, this thesis introduces SpikeStream, a software-defined neuromorphic acceleration framework that exploits RISC-V sparse streaming ISA extensions to accelerate SNN inference on programmable cores. *SpikeStream* achieves up to 7.3× speedup and 5.7× energy savings over SIMD baselines, while matching or outperforming specialized neuromorphic processors.

Finally, the thesis demonstrates the integration of an industry-grade LPDDR4 memory subsystem within the proposed heterogeneous RISC-V SoC, highlighting the area and integration trade-offs required to sustain high off-chip bandwidth.

Collectively, these contributions advance secure, flexible, and efficient RISC-V architectures for next-generation CPS and edge AI applications.

Keywords: RISC-V, Cyber-Physical Systems, Control-Flow Integrity, Spiking Neural Networks, Heterogeneous Platforms, Systems-on-Chip

Acknowledgments

This thesis is the result of a long journey, one that extends well beyond the technical contributions presented in these pages, I want to steal a page of this thesis to thank all the people who have been close to me during these demanding years of my Ph.D. journey, and even before.

I would like to begin by thanking all my friends and current and former colleagues from our lab in Bologna. I am deeply grateful to Emanuele Parisi, Luca Zanatta, Alberto Musa, Antonio del Vecchio, Giacomo Madella, Sebastiano Mengozzi, Giovanni Esposito and Samuel Zanella. Some of you were there with me from the very first days of my Ph.D. journey, sitting side by side in the lab; others joined along the way. To all of you, I am thankful for the countless hours spent together during long lab days, for the discussions that ranged far beyond research, and for the sense of companionship that made this experience truly shared. I also thank Federico Ficarelli for the many discussions on sparsity, and for patiently trying to pass on a bit of his knowledge of software engineering and compilers. I would also like to give a special mention to Emanuele Venieri, the first student I had the pleasure to supervise, who later started his own Ph.D. journey with us and with whom I am still very happily working.

I am equally grateful to the Zürich branch of the group for hosting me during my period abroad and for making me feel at home in a new academic environment. I thank Luca Bertaccini, Cristian Cioflan, Lorenzo Lamberti, Thomas Benz, Philippe Sauter, Arpan Suravi Prasad, Chi Zhang, Alessandro Ottaviano, Nils Wistoff, Paul Scheffler, Michael Rogenmoser, and Luca Colagrande. A special

mention goes to the J76.2 office, Sergio Mazzola and Victor Jung, and definitely to our "Sardinia group" and dear friends Enrico Zelioli, Christopher Reinwardt, and Cyril Koenig. I am beyond thankful for the countless discussions, long days, shared laughter, and mutual support during the Flamingo project tape-out, without a doubt one of the most challenging and intense projects I have ever taken part in.

I would like to sincerely thank all the professors who guided and supported me throughout this journey: Luca Benini, Andrea Acquaviva, Davide Rossi, and in particular Andrea Bartolini, for his constant guidance and for the many insightful discussions that helped shape both my research and my growth as a researcher.

I am also grateful to Dr. José Moreira from IBM (U.S.) and Prof. Gianluca Palermo from Politecnico di Milano for taking the time to read this thesis and for providing valuable and insightful feedback.

Beyond academia, I want to thank my group of lifelong friends, people who have been part of my life since childhood and who have always been there, regardless of where my path led me. In no particular order, Federico Scazzieri, Alessandro Tomesani, Francesco Marata, Francesco Pieri, Edoardo Delfanti, Andrea Pezzati, Giacomo Sambri, Giovanni Mercatelli, Gregorio Minelli Vacchi, Andrea Vanni, Federico Tarozzi, Andrea Tirini, Federico Stefani, Ugo Leone Cavalcanti, and Francesco D'Errico. Your presence, friendship, and shared memories have been a constant source of balance and strength.

I want to thank my family, Loris, Gloria, and Margherita, for always being close to me and for supporting me unconditionally in all my life choices. Your support has been a quiet but unwavering foundation throughout this journey.

Lastly, I want to thank one of the most important people in my

life, Liudmila. There are so many things I am grateful to you for that I could never list them all here. Your support, patience, and presence have meant more to me than words can express.

Thank you.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

1.1 Motivation

In today's world, Cyber-Physical Systems (CPS) are increasingly ubiquitous, shaping how we interact with the environment around us. They combine computational intelligence with physical processes, powering technologies from autonomous vehicles and smart healthcare devices to industrial automation and intelligent urban infrastructure [1]. CPS platforms span a broad spectrum, from deeply embedded microcontrollers to high-performance multi-core systems-on-chips (SoCs), each with distinct trade-offs in computational capability, energy consumption, and security features required. This thesis addresses the intermediate class of CPS devices, including embedded and autonomous platforms for domains such as robotics, smart mobility, and industrial control, which demand higher performance and memory scalability than ultra-low-power microcontroller-class systems while maintaining tight power and security constraints typical of embedded operation.

In recent years, the rapid integration of artificial intelligence (AI) and machine learning (ML) techniques into CPS has substan-

tially increased their computational and reliability requirements, as these platforms increasingly rely on data-driven perception, control, and decision-making [1, 2]. To meet these demands, modern CPS SoCs integrate dedicated AI processing units, including SIMD/vector extensions, tightly coupled multi-core clusters, and specialized accelerators [1, 3, 4]. These units support domain-specific computation optimized for the parallel and data-intensive nature of AI/ML workloads. In parallel with architectural advances, a wide range of software-level optimization techniques have been developed to improve the energy efficiency of on-device inference. Prominent approaches include network pruning, which removes redundant parameters at various granularities—such as weight-, channel-, or layer-level—to reduce computational cost and memory footprint [5, 6]; quantization, which lowers numerical precision to minimize storage and energy consumption [7, 8]; and sparsity-aware models, which exploit the zeros introduced by pruning or activation sparsity to skip unnecessary computations [9–11]. Together, these methods significantly reduce both computational load and memory traffic, the two dominant contributors to energy consumption in modern CPS devices. Among these methods, Spiking Neural Networks (SNNs) have emerged as a promising approach. By mimicking the event-driven nature of biological neurons, SNNs only perform computations when spikes are generated [12, 13], enabling ultra-low-power event-driven inference [14]. However, the irregular and sparse nature of spike-based workloads poses significant challenges for conventional hardware [15]. To address this, specialized SNN accelerators have been proposed, achieving excellent efficiency for spiking computations but often sacrificing flexibility in supporting other neural network models or future algorithmic innovations [14–16]. This highlights the continued relevance of programmable

architectures that can flexibly execute a wide range of CPS workloads, yet benefit from a controlled level of specialization to enhance efficiency for both traditional neural networks and emerging models such as SNNs.

While performance, efficiency, and flexibility are key enablers for modern CPS architectures, they represent only part of the challenge. As these systems increasingly interact with the physical world and process sensitive or private data—such as video streams from autonomous vehicles, biomedical sensor readings, or industrial control parameters—ensuring their security and trustworthiness becomes equally critical [17]. CPS platforms often operate in partially untrusted or adversarial environments, running complex, multi-layered software stacks that expand the attack surface and expose them to runtime exploitation [18, 19].

Among the most severe threats are control-flow hijacking attacks, where adversaries manipulate program execution to bypass security checks or inject malicious behavior. In the context of mobile and autonomous CPS, such attacks can compromise data integrity and even endanger physical processes. To counter these threats, hardware-enforced Control-Flow Integrity (CFI) has emerged as a fundamental protection mechanism [20, 21]. Rather than enforcing a full control-flow graph, CFI checks that each branch and return instruction targets only valid, pre-authorized destinations derived from compile-time analysis, thereby blocking malicious redirection of execution at runtime [22]. Industrial implementations such as Intel Control-Flow Enforcement Technology (CET), ARM Pointer Authentication, and IBM POWER10 return-oriented programming (ROP) protection demonstrate the practicality of these principles in modern processors [23–25].

Complementary to runtime CFI protections, a hardware Root of

Trust (RoT) provides a dedicated, isolated security domain within the SoC [26]. The RoT implements secure boot, authenticating firmware before execution; device identity and attestation services, enabling trusted interactions with external systems; and on-chip cryptographic engines and key storage for confidentiality and authentication [27, 28]. By anchoring trust at the hardware level, the RoT establishes a chain of trust spanning from boot through runtime, ensuring system integrity and protection of sensitive data. These complementary mechanisms illustrate the importance of multi-layered security architectures in CPS SoCs, motivating approaches that integrate runtime protections and system-wide trust domains.

Finally, memory interfaces play a pivotal role in determining the overall performance of CPS SoCs and their ability to support operating systems like Linux. Many academic and open-source platforms employ low-overhead interfaces such as HyperBus [29] or other serial links [30], which offer simplicity and minimal area cost but are extremely limited in sustained bandwidth and speed. For modern CPS workloads, these slower interfaces quickly become a bottleneck. In contrast, Low-Power Double-Data Rate (LPDDR) memory controllers provide significantly higher throughput and better support for parallel, burst-oriented access patterns, allowing host and accelerator domains to exchange data efficiently without stalling on off-chip transfers. This improvement comes at the expense of increased area and power overhead, primarily due to the analog and mixed-signal transceivers required for high-speed off-chip communication.

In summary, next-generation mobile and autonomous CPS platforms demand a careful balance between high computational performance, energy efficiency, and strong system-level security, while accommodating increasingly complex and diverse AI/ML workloads.

This work presents a RISC-V-based heterogeneous SoC template architecture designed to meet the demanding requirements of next-generation mobile and autonomous CPS platforms. The architecture integrates a secure application-class host processor with a hardware Root-of-Trust (RoT), a flexible multi-core acceleration cluster, and optimized software kernels targeting emerging SNN workloads to improve energy efficiency in event-driven AI processing, all supported by high-performance memory subsystems. By combining a controlled degree of processor specialization with optimized software libraries, the platform enables energy-efficient execution of both conventional and emerging AI workloads, while maintaining system adaptability and robust security through diverse hardware-assisted CFI methodologies. Despite advances in academic research and industrial design, no existing platform combines flexible acceleration for emerging AI workloads (e.g., SNNs), scalable memory performance, and hardware-based security within a unified, heterogeneous architecture.

1.2 Thesis Overview and Contributions

The key technical contributions of this thesis are summarized below, organized by chapter:

- **Chapter 2 – Heterogeneous Architectures for CPS**: Introduces the overall SoC template architecture that underpins the thesis. The chapter motivates the design of heterogeneous SoCs for mobile and autonomous CPS, emphasizing the integration of programmable host processors, flexible accelerator domains, memory hierarchies, and security domains. It provides the foundation for the subsequent chapters, which delve

deeper into each architectural domain introduced here.

- **Chapter 3 – Secure Host-Domain Processors**: This chapter explores the design, integration, and evaluation of hardware extensions for CFI in an application-class RISC-V processor. Building on the open-source, application-class CVA6 core, it introduces dedicated hardware units that enforce both forward- and backward-edge control-flow protection, mitigating runtime hijacking attacks in complex software stacks. The implementation, synthesized in 22 nm FDX technology, achieves only about 1% area overhead and up to 15.6% performance overhead on the MiBench automotive benchmarks.

- **Chapter 4 – Root-of-Trust as Control-Flow Integrity Co-Processor**: This chapter explores a novel method to enforce CFI within a hardware RoT for RISC-V CPS platforms. By streaming control-flow instructions from the protected core to the RoT and enforcing the CFI policy in firmware, the approach leverages existing hardware resources, avoids custom IP, and requires no toolchain modifications. Experimental results show low runtime overhead and minimal area impact ($\sim$1%), providing a secure and efficient foundation for RoT-based CPS systems.

- **Chapter 5 – Acceleration of Brain-Inspired AI through Sparse Streaming RISC-V ISA Extensions**: Addresses the challenge of energy-efficient AI inference in CPS, with a focus on SNNs. This contribution develops optimized software kernels for SNN execution on a programmable RISC-V accelerator domain, leveraging Sparse Linear Algebra ISA extensions to exploit data sparsity and maximize utilization of the floating-

point units. The approach demonstrates how general-purpose cores with lightweight ISA support can provide flexible yet efficient acceleration for emerging AI workloads, without relying on rigid domain-specific hardware units.

- **Chapter 6 – Integration and Analysis of the Memory Subsystem**: Presents the RTL integration, and physical implementation of the memory subsystem within the proposed heterogeneous RISC-V SoC. The subsystem combines a dual-port, software-managed L2 scratchpad memory (SPM) with an industry-grade LPDDR4 controller, implemented in GlobalFoundries 22FDX technology. It provides an analysis on latency and bandwidth between on-chip and off-chip domains, highlighting how industrial DRAM interfaces can be effectively integrated into academic RISC-V ecosystems to support higher-performance and Linux-capable CPS devices.

Chapters 3, 4, and 5 are adapted from previous publications. In particular, Chapter 3 builds upon [31], Chapter 4 upon [32], and Chapter 5 upon [33, 34]. Chapter 2 was written to provide context for the work and establish an architectural baseline for the various contributions proposed in this thesis. Chapter 6 was written autonomously to complete the exploration of all major architectural domains of a modern CPS SoC and to present the work carried out during the final year of my Ph.D. While I was responsible for the majority of the concept formulation, manuscript writing, technical development, and experimental evaluation for Chapters 3 and 5, I also made substantial contributions to Chapter 4. In particular, I contributed to the manuscript writing, the design and implementation of the host processor extensions, the system-level integration enabling communication between the host processor and the RoT,

the FPGA-based evaluation, and the integration of the design within the SoC to extract area results. Accordingly, throughout this thesis the use of the pronoun "we" reflects the collaborative nature of the work and acknowledges the essential guidance, technical contributions, support, and feedback provided by my co-authors in the papers related to this thesis.

Figure 1.1 provides an overview of how the chapters of this thesis relate to each other.

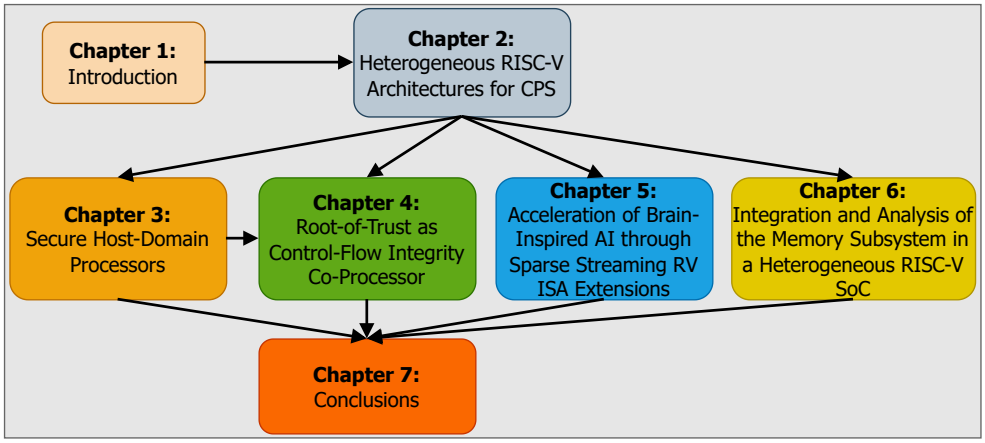


Figure 1.1. Overview of thesis structure and dependencies between chapters.

1.3 List of Publications

This list includes all publications that I authored or co-authored, both those whose content is included in this thesis and those that are not.

1. **S. Manoni**, E. Parisi, R. Tedeschi, D. Rossi, A. Acquaviva, A. Bartolini, "CVA6-CFI: A First Glance at RISC-V Control-Flow

Integrity Extensions", in *2026 IEEE International Symposium on Circuit and Systems*, IEEE, 2026, pp. 1–5.

2. **S. Manoni**, P. Scheffler, A. Di Mauro, L. Zanatta, A. Acquaviva, L. Benini, A. Bartolini, "NARS: Neuromorphic Acceleration through Register-Streaming Extensions on RISC-V Cores", in *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, ACM, 2024, pp. 79–82.

3. **S. Manoni**, P. Scheffler, L. Zanatta, A. Acquaviva, L. Benini, A. Bartolini, "SpikeStream: Accelerating Spiking Neural Network Inference on RISC-V Clusters with Sparse Computation Extensions", in *2025 Design, Automation & Test in Europe Conference (DATE)*, IEEE, 2025, pp. 1–7.

4. M. Grossi, **S. Manoni**, E. Parisi, M. Omaña, C. Metra, F. Alfonsi, A. Gabrielli, A. Acquaviva, "A PCI Express Based Data Acquisition System for the Monitoring of Code Traces of RISC-V Microprocessors", in *2024 8th International Conference on System Reliability and Safety (ICSRS)*, IEEE, 2024, pp. 10–15.

5. E. Parisi, A. Musa, **S. Manoni**, M. Ciani, D. Rossi, F. Barchi, A. Bartolini, A. Acquaviva, "TitanCFI: Toward Enforcing Control-Flow Integrity in the Root-of-Trust", in *2024 Design, Automation & Test in Europe Conference (DATE)*, IEEE, 2024, pp. 1–6.

6. L. Zanatta, F. Barchi, **S. Manoni**, S. Tolu, A. Bartolini, A. Acquaviva, "Exploring Spiking Neural Networks for Deep

Reinforcement Learning in Robotic Tasks", *Scientific Reports*, Nature Publishing Group, vol. 14, no. 1, 2024, Art. 30648.

7. U. Laghi, **S. Manoni**, E. Parisi, A. Bartolini, "Efficient Trace for RISC-V: Design, Evaluation, and Integration in CVA6", in *RISC-V Summit Europe*, 2025.

8. E. Venieri, **S. Manoni**, G. Ceccolini, G. Madella, F. Ficarelli, D. Gregori, D. Cesarini, L. Benini, A. Bartolini, "Monte Cimone v2: HPC RISC-V Cluster Evaluation and Optimization," in *ISC High Performance Conference*, Springer International Publishing, 2025.

Chapter 2

Heterogeneous RISC-V Architectures for CPS

2.1 Background

The sustained increase in transistor density, famously described by Moore's Law, drove the exponential growth in computing performance over the past several decades. Originally, transistor counts doubled roughly every 12–24 months, enabling ever more complex and higher-performing processors. However, by the mid-2000s, the scaling of voltage and power, described by Dennard scaling, began to break down: smaller transistors no longer automatically translated into higher clock frequencies or lower power consumption due to leakage currents and increased power density. As a result, simply shrinking transistors no longer guaranteed proportional gains in performance or energy efficiency, creating fundamental limits for traditional processor design [35]. General-purpose single-core CPUs had long been the workhorse of computing systems, providing highly versatile and programmable architectures. Their flexibility, however, came at a cost: significant silicon area is devoted to control logic, branch prediction, speculative execution, and multi-level caches,

leaving less area for actual computation. Consequently, single-core CPUs are suboptimal for highly parallel or data-intensive workloads, where computational density and energy efficiency are critical [36–38].

To overcome these limitations, the microprocessor industry shifted to multicore architectures, replicating CPU cores to increase throughput within the same power and area budgets. While multicore designs improved aggregate performance, their gains are limited by Amdahl's Law and the presence of "dark silicon," i.e., portions of the chip that must remain idle due to thermal and power constraints. Moreover, power consumption grows nonlinearly with added cores and higher clock rates, making conventional multicore scaling inefficient for modern high-throughput and low-power domains required in today's CPS [35].

These challenges motivated the rise of heterogeneous architectures, where general-purpose cores are combined with specialized accelerators to achieve higher computational density and energy-efficient processing for domain-specific workloads. Accelerators span a spectrum from highly domain-specific units, such as tensor processing units (TPUs) [39, 40] or neural network activation-function accelerators (e.g., softmax, ReLU, Leaky-Integrate and Fire) [41, 42], to more general-purpose multi-core clusters with Multiple-Instruction-Multiple-Data (MIMD) execution [43]. In these clusters, control logic and cache hierarchies are minimized, and speculative execution is largely removed compared to conventional CPUs, resulting in ultra-lightweight CPU cores that dedicate a larger fraction of silicon to arithmetic and data-path units [44]. Such clusters can be replicated to scale to hundreds of cores; however, software must explicitly manage inter-core communication and synchronization, making programming for massive parallelism more complex despite

the cores' general-purpose flexibility. For workloads demanding extreme parallelism, GPUs provide a complementary solution: they consist of hundreds to thousands of lightweight cores organized for Single-Instruction-Multiple-Data (SIMD or SIMT) execution. Unlike multi-core clusters, GPUs feature hardware-supported synchronization mechanisms and a programming model designed for highly parallel, data-parallel workloads, allowing efficient scaling to thousands of threads with minimal programmer intervention [45].

To efficiently implement such heterogeneous architectures, modern SoCs integrate multiple processing domains, each tailored to distinct performance, power, and programmability requirements. Examples of this approach can be found in recent industrial [43, 46] and research platforms [47–49]. These domains are interconnected through on-chip communication fabrics that balance scalability, latency, and energy efficiency. For large-scale or highly modular systems, Networks-on-Chip (NoCs) are typically employed to support concurrent communication across many domains [30, 50, 51]. NoCs offer excellent scalability and bandwidth, but at the cost of higher latency due to packet routing. In contrast, crossbar interconnects remain a preferred solution for smaller or more tightly coupled configurations, such as embedded, and mobile CPS platforms, offering lower latency and simpler arbitration [43, 52–55]. In summary, heterogeneous CPS architectures are divided into multiple domains that are interconnected through NoCs or crossbars, depending on the system's scale. The main domains can be broadly identified as follows:

- **Host Domain**: This domain contains one or more general-purpose processing cores, typically configurable at design time, and integrated within a coherent interconnect framework. Cores

may be 32- or 64-bit, with private L1 caches and optional shared L2 caches supporting cache-coherence protocols [47]. The host domain executes operating systems, middleware, and control logic, and provides standard memory and I/O interfaces to interact with accelerators, peripherals, and other system components [56]. Its design is modular and transparent, allowing cores to operate independently of the specific heterogeneous architecture while supporting secure and coordinated system management.

- **Memory Subsystem**: This domain provides one or more channels to off-chip memory and incorporates configurable on-chip memory. It may include partitions of the last-level cache (LLC) with corresponding directories to maintain coherence with processor caches. On-chip hierarchies can include multi-level caches and scratchpad memories to support both general-purpose processing and hardware accelerators. The memory subsystem integrates with standard interconnects and may use high-speed serial links for off-chip communication. This design enables transparent address-space partitioning and scalable, coherent access across processors and accelerators. Memory options range from compact, lower-speed solutions such as on-chip SRAM and HyperRAM to higher-performance LPDDR3/4/5 or DDR3/4/5, depending on throughput and energy requirements.

- **I/O Domain**: This domain hosts the SoC's shared peripherals and interfaces, connecting the platform to sensors, actuators, communication buses, and external devices. It typically includes network interfaces (e.g., Ethernet NICs), UARTs, timers, GPIOs, and display controllers, as well as DMA engines for

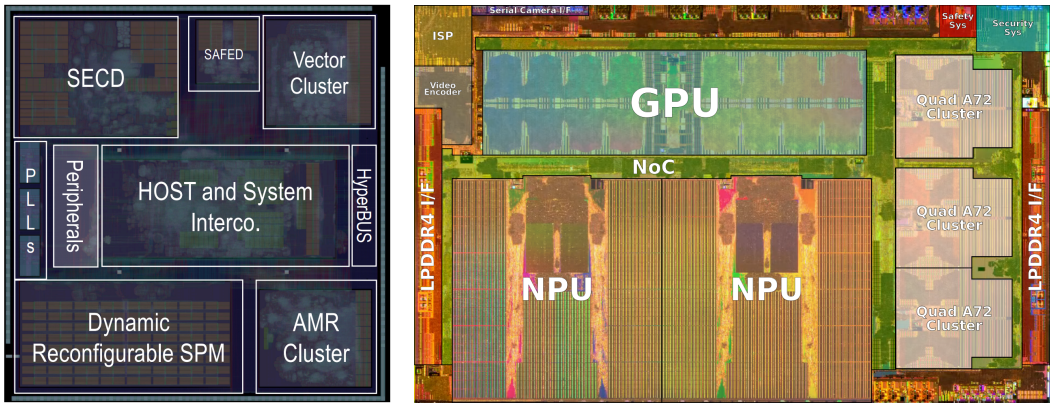
high-bandwidth data movement. The I/O domain integrates coherently with the memory subsystem and processor cores through dedicated proxies or interconnect adapters, supporting both memory-mapped and DMA-based access. It may also provide debug and monitoring interfaces, boot ROMs, and other auxiliary services to facilitate system management, configuration, and performance tracking. High-speed serial links or off-chip communication interfaces can be included and intersect with the Memory Subsystem to interact with external memories, enabling seamless integration into CPS platforms.

- **Accelerator Domain**: This domain contains compute units that range from highly domain-specific hardware to more flexible architectures such as GPUs or MIMD-style clusters of lightweight CPUs. These units typically execute coarse-grained or data-parallel tasks independently of the host-domain cores, exchanging data with the memory subsystem through either coherent or non-coherent channels. Integration interfaces generally provide mechanisms for configuration, control, and completion signaling, as well as support for DMA or high-bandwidth memory access [57]. Optional point-to-point communication between accelerators may be supported for direct data exchange without passing through shared memory. Run-time synchronization, coherence, and resource management can be handled via hardware protocols or software drivers depending on the accelerator type and system architecture [47]. This domain enhances throughput and energy efficiency for compute-intensive tasks such as control algorithms, digital signal processing, or AI/ML inference, while leaving the host domain free for general-purpose execution.

Individual domains can be customized for specific applications. For example, automotive and aerospace platforms often require strong security and fault tolerance. As discussed in Section 1.1, security often relies on foundational elements, such as a RoT, and protections, such as CFI, to safeguard critical code and data. Fault tolerance can be enhanced with modular redundancy, error detection, and recovery schemes, all of which are crucial for safety-critical automotive systems [58, 59].

In Figure 2.1, two annotated layouts are presented, representing prominent examples of modern systems-on-chip (SoCs) for mobile and autonomous computing, taken respectively from industry and academia. Both layouts highlight the architectural organization of the processing domains and the placement of major functional components.

The first SoC, Carfield [53], integrates a central host domain and multiple specialized accelerators designed to offload computation-intensive and safety-critical workloads. The safe domain (SAFED) includes a lockstepped multi-core cluster for enhanced reliability, featuring ECC-protected private instruction and data scratchpad memories that ensure deterministic access. The adaptive modular redundancy (AMR) cluster, located in the lower region of the layout, consists of several general-purpose RISC-V cores sharing a banked, ECC-protected L1 scratchpad memory connected through a high-bandwidth, low-latency interconnect. Within this cluster, custom RISC-V mixed-precision extensions enable SIMD operations across multiple data formats, accelerating integer, signal-processing, and AI workloads. A dedicated vector accelerator complements the AMR and safe domains, targeting high-throughput compute kernels through wide SIMD execution and mixed-precision arithmetic support. Soft real-time workloads with relaxed safety constraints



(a) Carfield SoC. (b) Tesla dojo.

Figure 2.1. Annotated layouts of modern SoCs for autonomous and mobile computing.

execute in the host domain, which is based on the Cheshire platform [56] and extended with a dual-core RISC-V processor supporting hardware-assisted virtualization. A secure domain (SECD), on the top-left of the layout, acts as the SoC's hardware root of trust, managing the secure boot process and providing cryptographic services for system integrity and data protection. Finally, the memory subsystem domain features a dynamically reconfigurable L2 scratchpad memory and a HyperRAM controller that provides access to off-chip memory.

In Figure 2.1b, the annotated layout of Tesla's Full Self-Driving (FSD) SoC [46] is shown. The figure highlights the main architectural domains, which broadly correspond to those identified in Carfield—namely, the host, accelerator (including safety system), memory, and security subsystems. The host domain is built around three quad-core clusters based on the ARM Cortex-A72 application-class processor, playing a role analogous to the Host domain in Carfield. The accelerator domain comprises a highly parallel com-

pute subsystem, including a GPU and a neural processing engine located in the central region of the layout, optimized for matrix- and convolution-based neural-network operations [46]. This domain also includes several additional smaller accelerators for digital-signal-processing tasks, located toward the top-left of the SoC, along with the safety subsystem in the top-right of the floorplan, which ensures reliable execution of critical workloads and serves a similar function to the SAFED in Carfield. The memory subsystem features an LPDDR4 memory controller for off-chip memory connectivity, while the L2 memory is integrated within the host domain. Finally, the security system occupies the upper-right section of the layout, performing the functions of a hardware root of trust comparable to the RoT in Carfield.

2.1.1 RISC-V ISA

Among the many architectural choices for implementing heterogeneous CPS SoCs and considering all the compute elements in an autonomous CPS, RISC-V has since its inception, attracted significant adoption across both academia and industry. This is largely due to its open and modular nature: By providing a compact, clean base ISA and a rich set of optional extensions, RISC-V allows system designers to implement cores tailored to each domain. high-performance application processors for the host, ultra-low-power microcontrollers for I/O management, or custom instruction set extensions that directly interface with accelerators. Its extensibility also enables domain-specific optimizations such as linear algebra instructions for AI, lightweight bit-manipulation units for control algorithms, or security extensions for protecting safety-critical workloads.

RISC-V is an open-standard instruction set architecture (ISA) based on a reduced instruction set computing (RISC) design. Its modular structure consists of a small base ISA that can be extended with optional standard or custom extensions, enabling scalable implementations across a wide range of performance, area, and energy targets. The base ISA supports integer arithmetic, control-flow operations, and basic memory access instructions, while extensions such as `M` (integer multiplication and division), `A` (atomic operations), `F` and `D` (single- and double-precision floating point), and `C` (compressed instructions) provide additional computational capabilities and code density optimizations.

RISC-V also defines privileged architecture specifications to support system-level functions, including memory management, exceptions, interrupts, and privilege modes. This enables the implementation of application-class cores capable of running full-fledged operating systems as well as microcontroller-class cores for lightweight, low-power embedded environments.

In this thesis, all the processors under consideration are based on the RISC-V ISA and on extensions of it.

2.2 Template architecture for mobile CPS

This section introduces an architectural template for mobile and autonomous CPS. It unifies the various hardware domains described in the previous section, establishing a baseline architecture that serves as the foundation for the contributions presented in this thesis. Subsequent chapters will systematically expand upon this baseline architecture to present and evaluate these contributions. As illustrated in Figure 2.2, the proposed architecture is organized

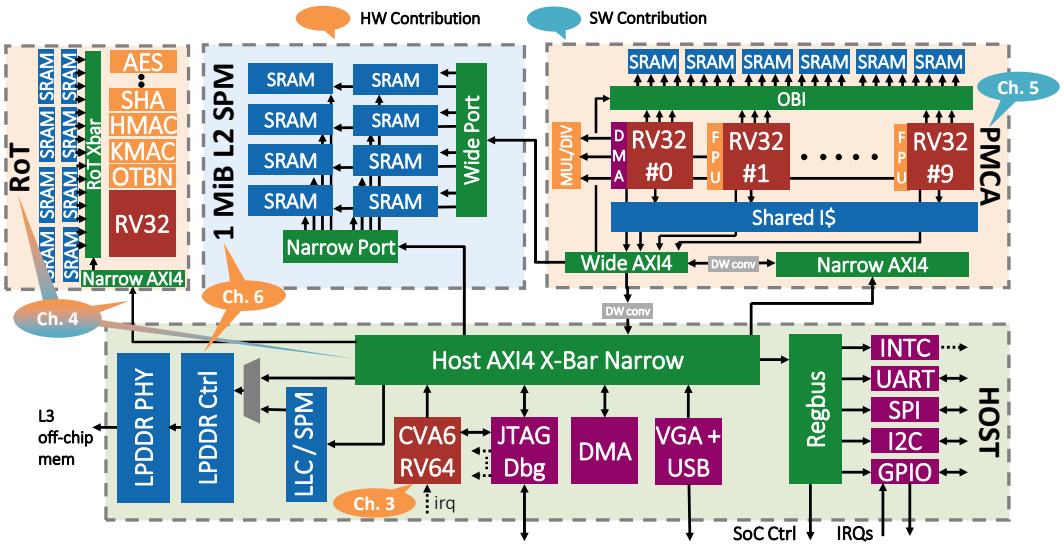


Figure 2.2. Proposed template architecture.

around a modular structure composed of multiple domains, as the one introduced in section 2.1 interconnected through a narrow 64-bit Advanced eXtensible Interface 4 (AXI4) crossbar, which serves as the main interconnect for the plugin of other domains, such as the accelerator, secure, and memory subsystems. The host domain includes an RV64 application-class processor and a set of I/O peripherals, serving as the central coordination and control unit. Connected to the host crossbar is the accelerator domain, which comprises a programmable multi-core accelerator (PMCA) built from lightweight RV32 cores sharing an instruction cache, a multi-banked scratchpad memory, and a dedicated DMA engine to enable efficient data movement. In parallel, the secure domain hosts a RoT that ensures platform integrity through secure boot, attestation, and dedicated cryptographic accelerators for hashing, authentication, and encryption. Finally, the memory subsystem integrates an L2 scratchpad memory and a LLC, interfacing the main AXI4 interconnect with an LPDDR4 memory controller to

provide scalable and efficient access to off-chip memory. Figure 2.2 also indicates the type of contribution (HW, SW, or both) and the domain addressed in the dedicated Chapter.

Chapter 3
Secure Host-Domain Processors

3.1 Introduction

RISC-V embedded platforms are experiencing rapid adoption across security and safety-critical industrial CPS. These computing systems frequently execute critical workloads implemented in memory-unsafe languages, while operating in untrusted environments exposed over the network. These features make them prime targets for cyberattacks [60]. In fact, an attacker could exploit memory corruption bugs to modify vulnerable code pointers in memory, hijack program control-flow, and trigger arbitrary code execution leveraging code-reuse attacks such as return-oriented programming (ROP), putting at risk the security of the whole system where the vulnerable device operates [61–63]. CFI has emerged as an effective defense mechanism against control-flow hijacking attacks by ensuring that programs execute only along valid and intended control-flow paths, thereby preventing unauthorized deviations at runtime. A typical CFI policy enforces protection on both "forward edges" and "backward edges" of control flow. Forward-edge policies protect indirect jumps where target addresses are computed dynamically during execution, such

as those resulting from function pointer dereferencing in C. Conversely, backward-edge protection secures function returns, whose target addresses depend on values retrieved from the stack memory. Various hardware-level approaches implement CFI, including ISA extensions and dedicated coprocessors, each offering different trade-offs between security and performance overheads [32, 64–66]. While commercial architectures like ARM, Intel and POWER10 have adopted CFI enforcement through proprietary ISA extensions and hybrid co-processor mechanisms [25, 67, 68], respectively, the RISC-V ISA has historically lacked native CFI support. However, RISC-V has recently ratified two CFI ISA extensions, `Zicfiss` and `Zicfilp`, which standardize an instruction-level mechanism for backward and forward-edge protection. This aims to mitigate code-reuse attacks through the introduction of shadow stacks and landing pads, respectively. To date, no commercial processor implements these extensions, and no study has evaluated their microarchitectural impact.

This chapter explores how the RISC-V CFI extensions impact power, performance, and area of RISC-V application-class embedded cores, our contributions are listed below:

- We design two hardware components implementing the RISC-V CFI extensions: a shadow stack unit implementing the `Zicfiss` extension for backward-edge protection and a landing-pad unit implementing the `Zicfilp` extensions for forward-edge protection.

- We integrate the designed components into CVA6, an embedded application-class core, presenting CVA6-CFI, a CVA6 variant with full CFI support. The complete implementation

is available open-source[1].

- We evaluate the impact of our RISC-V CFI extensions implementation, showing only 1.0% area overhead and upto 15.6% performance overhead on the enhanced CVA6 using the MiBench automotive benchmark subset.

3.2 Background

3.2.1 Commercial ISA Extensions for CFI

Many architectures, most notably Intel and ARM, have extended their ISA with additional instructions to mitigate control-flow subversion attacks. These instructions ensure that indirect control transfers execute only according to a pre-determined control-flow graph. Intel's CET includes two key features: Shadow Stack and Indirect Branch Tracking (IBT). The Shadow Stack is a write-protected memory page dedicated to storing return address pointers whenever a `CALL` instruction is executed. On function returns, an exception is raised if the target address does not match the caller address stored in the shadow stack. IBT introduced the `ENDBRANCH` instruction, which marks valid code targets for indirect calls and jumps. If an indirect control transfer attempts to target any invalid target, the processor raises an exception [68]. ARM architectures introduced two primary defense mechanisms: Branch Target Instructions (BTI) to protect indirect forward jumps and Pointer Authentication (PAC) for function return protection. Like Intel's `ENDBRANCH`, BTI restricts indirect branches to target only these designated instructions and raises an exception if this constraint is violated [69]. For function

[1] https://github.com/AlSaqr-platform/cva6/tree/pulp-v1-culsans

return protection, ARM uses pointer authentication, which utilizes the unused upper bits of 64-bit pointers to store a PAC. When a function is called, the return address is signed by generating and adding a PAC to its upper bits. Before the function returns, the return address must be authenticated using the stored PAC. If a ROP attack has counterfeited the return address, the authentication fails, an invalid address is returned, and an exception is triggered [67].

IBM POWER10 introduces a ROP protection mechanism based on cryptographic validation of function return addresses [25]. Instead of maintaining a separate shadow stack or embedding authentication codes within pointers, POWER10 computes a cryptographic hash of the return address at function entry and stores it in memory alongside the conventional stack frame. Upon function return, the hash is recomputed and compared against the stored value; any mismatch is treated as a control-flow integrity violation and triggers an exception. To support this mechanism, the Power ISA was extended with two dedicated instructions, `hashst` and `hashchk`, which are used at function entry and return, respectively. The `hashst` instruction computes a cryptographic hash over the function return address and the current stack pointer using a supervisor-managed secret key, and stores the resulting hash in memory alongside the stack frame. At function return, the `hashchk` instruction recomputes the hash and compares it against the stored value, triggering an exception in case of a mismatch. These instructions are implemented using dedicated hardware units in POWER10 to minimize performance overhead, and are treated as no-operations on processors that do not support the extension or when the feature is disabled, ensuring backward compatibility.

3.2.2 CFI RISC-V ISA extension

Recently, RISC-V extended its ISA to support CFI through the introduction of Landing Pads and Shadow Stack extensions. The `Zicfilp` extension introduces a labeled landing pad instruction (`lpad`) encoded as `auipc x0, label`, which serves as a PC-relative marker identifying valid targets for indirect control transfers without modifying architectural state. The execution of an indirect jump forces the processor into a state where the only instruction allowed to retire next is a `lpad` instruction with a matching label, or a software-check exception is triggered. The expected label is pre-loaded in register `x7` and must match the label encoded in the immediate field of the `lpad` instruction. The `Zicfiss` extension provides backward-edge protection via a shadow stack managed through dedicated instructions, including shadow stack push (`sspush`), pop and check (`sspopchk`), pointer read (`ssrdp`), and atomic swap (`ssamoswap`). Non-leaf functions save the link register into the shadow stack entry tracked by a dedicated shadow-stack pointer. Programs maintain both the regular and shadow stacks to protect return addresses from tampering. Unlike Intel's CET, the return address is not transparently pushed and popped from the shadow stack in conjunction with function calls and returns.

3.2.3 CVA6 microarchitecture

CVA6 is a 64-bit core designed for application-class systems. It features an in-order, single-issue, six-stage pipeline with hardware support for virtualization. The pipeline features two front-end stages responsible for generating the next program counter and interfacing with the instruction cache to fetch instructions. The decode stage

realigns and decodes the instructions, storing them in the issue queue. The issue stage contains the issue queue, scoreboard, and reorder buffer, and it issues instructions to the execute stage once all operands are ready. The execute stage integrates the core functional units, and the commit stage retires up to two instructions per cycle, updates the register file, and resolves write-back conflicts through the reorder buffer.

3.3 Methodology

To enhance CVA6 with CFI capabilities, we independently integrated the `Zicfi(ss/lp)` RISC-V extensions into the CPU pipeline. Subsection 3.3.1 details the `Zicfiss` extension and Subsection 3.3.2 details the `Zicfilp` extension.

3.3.1 Shadow Stack ISA extension design

Figure 3.1 illustrates the CVA6 pipeline, enhanced with the CFI extensions. The modules highlighted in blue have been introduced, or modified, to support the `Zicfiss` extension. The Control Status Registers (CSRs) have been extended to include the Shadow Stack pointer, and the environment configuration registers for machine, supervisor, and hypervisor modes have been integrated and expanded with a Shadow Stack enable field. This field is utilized in the frontend to compute the Shadow Stack-enabled state across various privilege levels and is then forwarded to the decode and execute stages. The decoder and compressed decoder are extended to handle the decoding of `sspush`, `sspopchk`, `ssprd`, and `ssamoswap` instructions, as well as their compressed counterparts. The `sspush` and `sspopchk` instructions have been mapped to store and load

instructions, respectively, with special operation tags to ensure that `Zicfiss` operations are differentiated from standard memory operations in later pipeline stages. The same applies to the `ssamoswap` instructions, which are tagged with dedicated Shadow Stack swap W/D identifiers. We have implemented a dedicated Shadow Stack Unit (SSU) in the execution stage, responsible for performing shadow stack pop checks and early instruction filtering before allowing the instructions to access the load-store unit (LSU). If the SSU detects a `ssamoswap` operation executing in machine mode, or any `Zicfiss` instruction running with Supervisor or Virtual address translation and protection disabled, and at a privilege level below machine mode, a store access fault exception is raised and forwarded directly to the scoreboard. If neither of these two conditions is met, access to the LSU is allowed. Furthermore, when the SSU detects a `sspopchk` instruction, it buffers the link register content and the load transaction ID in dedicated registers and then waits for the load result. It also sets a flag to indicate that a `sspopchk` is in progress. When a `sspopchk` is in-flight, the SSU compares the load transaction ID of each load result coming from the LSU with the buffered transaction ID. If these IDs match, the SSU then checks the recorded link register value against the load result. If there is a mismatch between these values, a software check exception is raised. Additionally, we extended the memory management unit (MMU) to enforce checks based on the type of page accessed. `Zicfiss` instructions can access only shadow stack pages, while non-`Zicfiss` can access only non-shadow stack pages. Any violation of this access policy results in a store access fault.

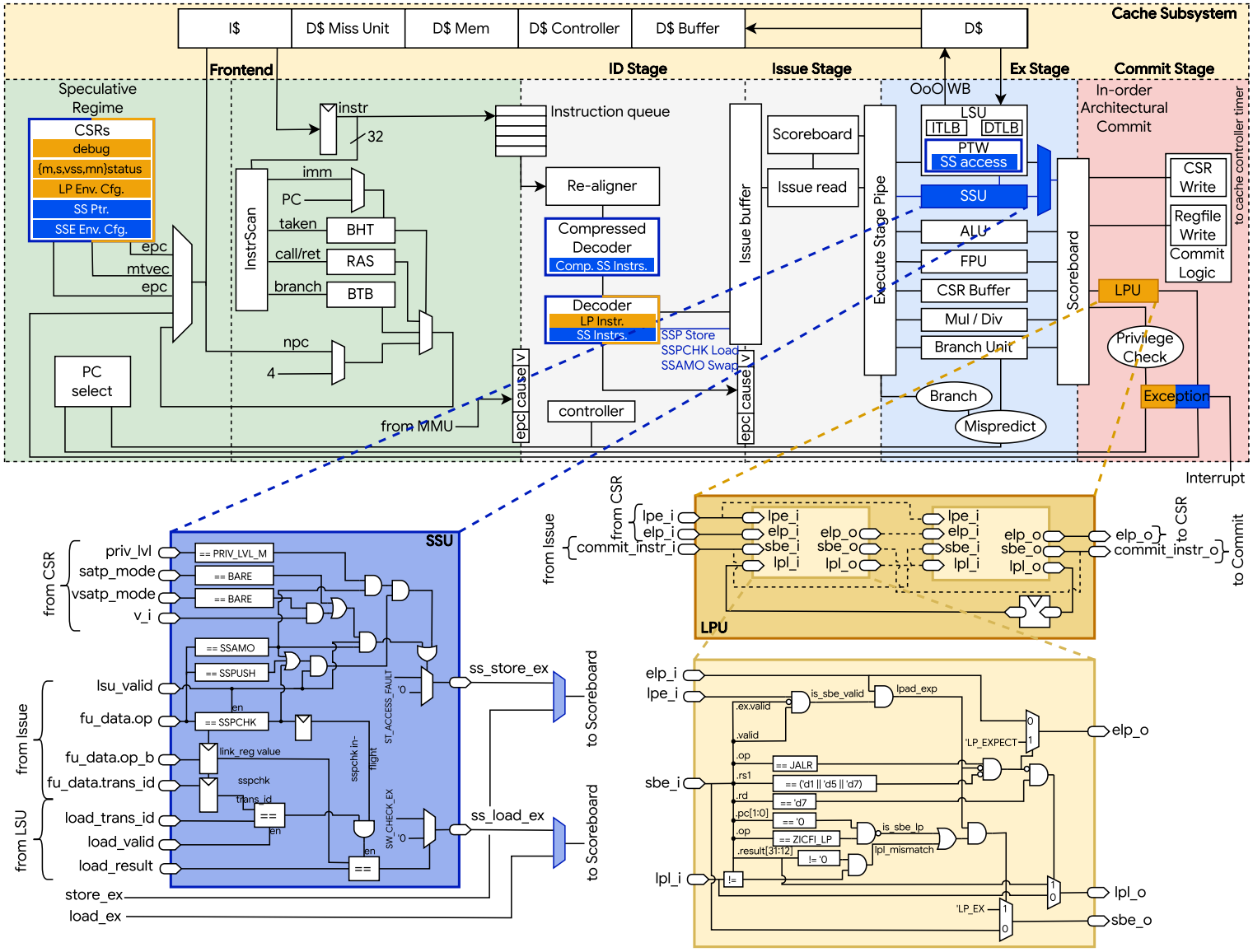


Figure 3.1. CVA6-CFI microarchitecture.

3.3.2 Landing Pads ISA extension design

The modules highlighted in orange in Figure 3.1 have been introduced, or modified, to support the `Zicfilp` extension. To enforce forward-edge CFI, RISC-V compilers emit a dedicated `lpad` as the first instruction of any indirect jump targets. To support this mechanism, we integrate a Landing Pad Unit (LPU) at the interface between the scoreboard and the commit stage of the core pipeline. The LPU monitors ordered, non-speculative, and valid instructions, and it checks two events. First, when an update to the `x7` register is detected, the LPU records this as the most recent valid landing pad label configured by the program. Second, when an indirect jump is observed, the pipeline transitions to a state in which only the `lpad` instruction matching the current landing pad label is permitted to execute. Upon reaching this state, if a matching `lpad` instruction is encountered, it is retired without side effects; if not, the LPU raises an exception. By observing fully executed, non-speculative instructions, the LPU design facilitates seamless integration into pipelines such as CVA6, which feature multiple commit ports capable of retiring more than one instruction per cycle. In these configurations, a chain of LPUs is instantiated, one per commit port, and each instance propagates information about `lpad` executions or landing pad label updates to the next unit. While this design choice increases propagation delay in the commit stage, it addresses corner cases, such as when an indirect jump and its landing pad are retired in the same cycle. Similarly to the `Zicfiss` extension, we extended the CSR register file and the decode stage with the necessary control and decoding logic to support the `Zicfilp` extension.

Table 3.1. CVA6 Area Overhead Analysis in 22nm FDX

CVA6 Pipeline	Area [μm^2]		Overhead [%]
	Baseline	w/ CFI ext.	
CSR RegFile	7831	8220	5.0
Fetch & Decode	1101	1124	2.1
Issue	25637	25788	0.6
Execute	61631	61854	0.4
Commit	182	372	103.6
Total	96382	97358	1.0

3.4 Results

To evaluate the area and performance overhead of the proposed control-flow integrity extensions, we synthesized the CVA6 core, with and without the `Zicfilp` and `Zicfiss` ISA extensions, using Synopsys Design Compiler targeting GF22FDX technology. The target operating frequency was fixed at 800 MHz under worst-case conditions (SSG corner, 0.72 V, –40°C to 125°C). On the software side, we evaluated the implementation using the MiBench automotive benchmark subset [70] compiled with the SiFive GCC 13.3 toolchain[2], which provides CFI-aware compilation support. Due to the absence of a stable CFI-patched Linux release, all benchmarks were executed on QEMU using a cycle-accurate model of the CPU calibrated against register transfer level (RTL) simulations performed with Questa 2023.1.

3.4.1 Hardware Characterization

[2] https://github.com/sifive/riscv-gnu-toolchain/tree/cfi-dev

Table 4.4 reports the area results for the CPU pipeline stages only. The cache subsystem is excluded to avoid masking the overhead introduced by our CFI extensions, as the cache remains identical between the baseline CVA6 and the CFI-enhanced version. Our implementation introduces low relative area overhead across most pipeline stages, except for the commit stage. Although this stage doubles in size due to the LPU unit, it is the smallest pipeline stage. Therefore, the overall area increase for CFI-enhanced CVA6 remains negligible at only 1.0%. Furthermore, our extensions do not impact the operating frequency. Figure 3.2 presents the area overhead comparison, extended from [71] between our proposed CFI design and prior hardware-centric CFI methods [66, 71–75], showing that CVA6-CFI achieves the smallest overhead.

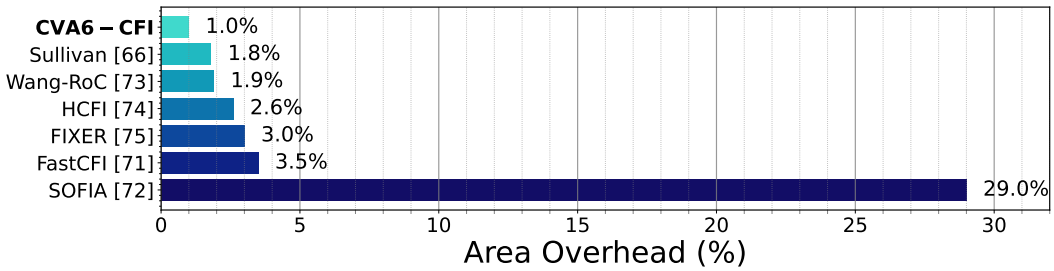


Figure 3.2. CVA6-CFI area overhead comparison.

3.4.2 Software Characterization

Figure 3.3 presents the code size overhead introduced by the CFI implementation across the MiBench automotive subset, compiled using the CFI-enabled toolchain. The results demonstrate consistent instrumentation costs with several key observations: The CFI extensions introduce a stable absolute overhead of approximately 22-23 kB across all benchmarks, representing a relative increase of

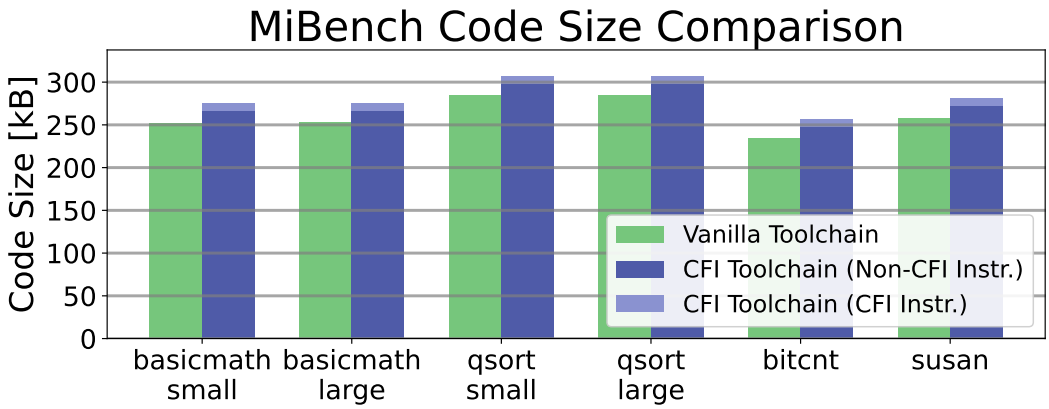


Figure 3.3. Code size overhead for MiBench Automotive benchmarks.

7.6-9.2%. This consistency suggests that the instrumentation adds a largely fixed cost independent of specific application characteristics. Notably, the overhead remains virtually identical between small and large input variants of the same benchmarks (*basicmath* and *qsort*), indicating that the CFI cost is primarily determined by program structure rather than data size or complexity. Figure 3.4 presents the number of executed instructions for MiBench programs compiled with the CFI-enabled toolchain. *basicmath_large* and *qsort_large* exhibit the highest CFI instruction counts, while *bitcount* and *susan* show minimal CFI activity. This pattern directly correlates with function call density: *basicmath_large*'s deeply nested loops calling a cubic equation solver repeatedly generate intensive shadow stack activity, while *qsort_large*'s recursive partitioning and frequent comparisons create substantial CFI overhead. Conversely, *bitcount*'s bitwise operations in tight loops and susan's image processing with direct function calls result in minimal CFI instructions execution. Across all benchmarks, CFI instructions represent less than 0.5% of total executed instructions. Notably, `ssamoswap` instructions are absent from all benchmarks as they are primarily required in the

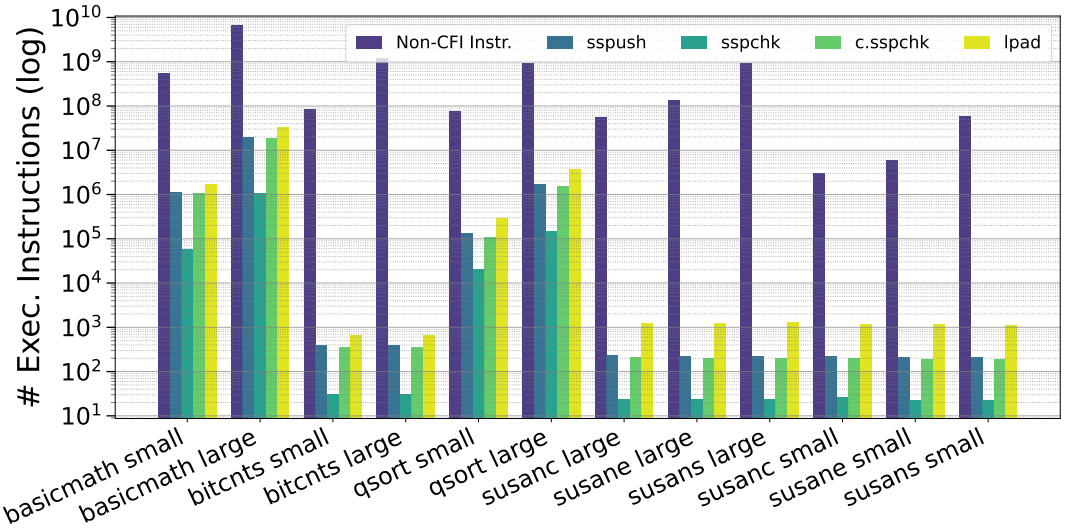


Figure 3.4. Number of CFI instructions executed for each MiBench program.

Linux kernel for context switching and user-space shadow stack management. Similarly, `ssrdp` only appear once during the initial benchmark setup to read the shadow stack pointer for stack frame establishment. Therefore, we did not include them in the figure. Figure 3.5 compares the cycle counts of programs compiled with the vanilla and CFI-enabled toolchains running on CVA6 and CVA6-CFI respectively. The CFI extension introduces a modest slowdown across all benchmarks, with relative overhead ranging from 1.0% to 15.6%. This overhead is primarily determined by shadow stack operations (`sspush`, `sspchk`), which account for all CFI-related memory accesses. Despite their prevalence, `lpad` instructions have a lower performance impact, as the dedicated LPU resolves them in a single cycle without requiring memory access when they reach the execute stage.

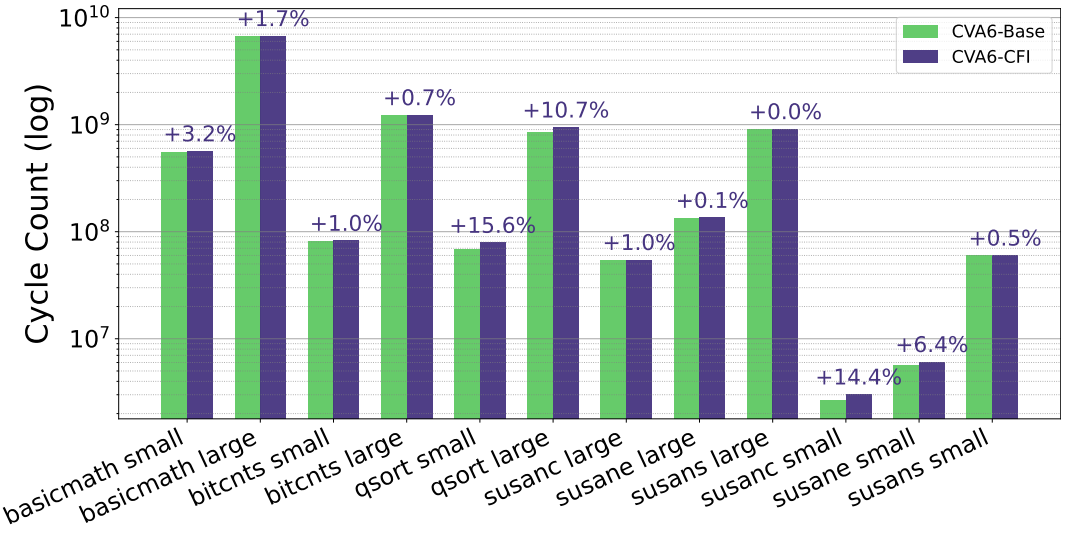


Figure 3.5. Performance comparison between CVA6 baseline and CVA6-CFI.

3.5 Summary

In this chapter, we presented the first design, integration, and evaluation of the RISC-V CFI extension. Our approach introduces two independent hardware units implementing backward-edge and forward-edge control-flow protection. The enhanced CVA6 incurs only 1.0% area and up to 15.6% performance overhead on the MiBench automotive subset, the full implementation is released open-source. Future work includes silicon prototyping and comparison with other hardware-based CFI solutions. The integration of the RISC-V CFI extension into the CVA6 CPU has been completed within the Alsaqrv2 platform, which features an architecture similar to the template architecture introduced in Section 2.2 and shown in Figure 3.6. The tape-out of this design is scheduled for Q2∼2026.

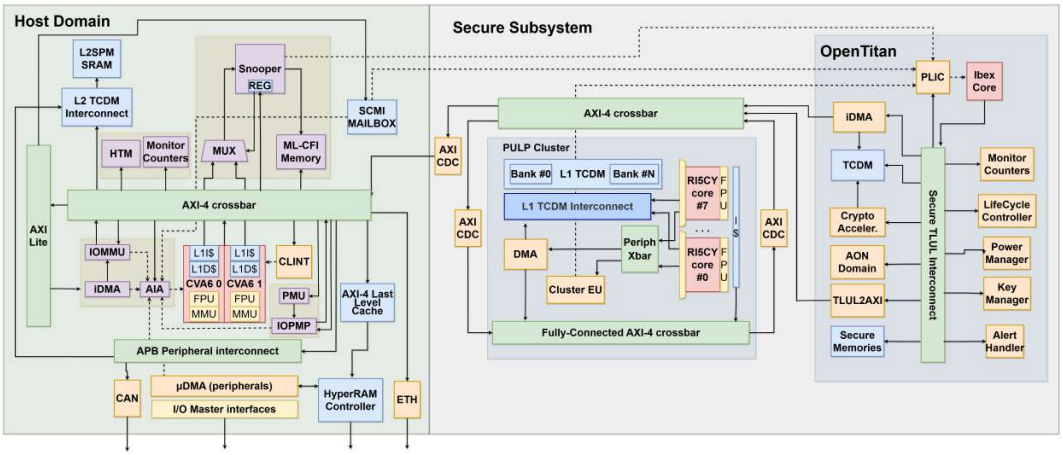


Figure 3.6. Alsaqr v2 SoC architecture.

Chapter 4
Root-of-Trust as Control-Flow Integrity Co-Processor

4.1 Introduction

Building upon the discussion of hardware-level CFI mechanisms in the previous chapter, this section explores an alternative approach that leverages existing trusted hardware components to enforce runtime control-flow protection. In particular, many modern RISC-V SoC designs integrate a RoT to provide a secure foundation for operations such as verified boot and attestation. The OpenTitan RoT, designed as an open-source silicon RoT, represents a prime candidate for extending such security capabilities beyond initialization time. While OpenTitan already ensures the authenticity and integrity of firmware during boot, its embedded processor and security primitives can be repurposed to provide runtime control-flow protection for the host system. In addition to the ISA extensions explored in the previous chapter, various other architectural strategies have been proposed to implement CFI at the hardware level, such as hardware monitors and programmable co-processors. As discussed in Chapter 3, ISA-based techniques generally incur minimal runtime

overhead but require toolchain modifications and recompilation of all software, which prevents protection of legacy binaries. Hardware monitors, on the other hand, integrate dedicated IPs within the processor pipeline to track control-flow events and enforce CFI policies directly in hardware [65]. These solutions operate transparently with respect to software execution but lack flexibility, as modifying the policy typically requires redesigning the hardware logic. In contrast, programmable co-processors [76, 77] implement the CFI enforcement logic in software running on an auxiliary core. This approach allows more adaptable and customizable policies but increases area overhead due to the additional processing unit. This chapter introduces an innovative co-processor-based architecture for implementing custom CFI policies on a modern RISC-V platform [78] that includes the OpenTitan RoT and a RV64GC CVA6 host processor [79]. Our approach relies on exploiting the OpenTitan RoT, that is already present on the platform to enable Secure Boot and Remote Attestation, as a CFI co-processor. The motivation behind this choice is to harness the RV32IMAC Ibex [80] core within OpenTitan to execute custom CFI policies in software. This approach avoids the area overhead associated with integrating a separate security monitor and maximizes the utilization of the RoT, which typically remains unused after the platform is initially set up. Moreover, our solution takes advantage of the security features provided by the RoT, including access to private tamper-proof storage and cryptography accelerators, to provide additional security guarantees with respect to the other SoA solutions. We demonstrate that our solution incurs negligible area overhead and imposes no, or $< 10\%$ overhead for the majority of the tested benchmarks.

To summarize, this chapter introduces the following contributions:

- We extend the architecture of a modern RISC-V SoC for autonomous vehicle [78] to (i) enable OpenTitan to observe the stream of control flow instructions retired by the host processor, and (ii) enhance the host core commit stage to filter the control-flow instructions, and enqueue them in a FIFO until the monitor marks them as 'safe'. Moreover, (iii) we extend the firmware running on the OpenTitan IBEX core to analyze the selected instructions and signal any control flow violations.
- We characterize the overhead of running CFI checks in the RoT by measuring the runtime overhead of our control flow enforcement scheme on a set of benchmarks applications, for different CFI check latencies.
- We prove the functionality of the designed system implementing a CFI return address protection policy based on a shadow stack. Moreover, we analyze the runtime penalty and hardware overhead of our solution and compare it to the original design.

4.2 Related Works

Among the several approaches proposed in the SoA to enforce CFI, the most relevant to our work are (i) FIXER [75], (ii) DExIE [65] and (iii) PHMon [76].

FIXER [75] is a security module that enforces CFI by implementing a shadow stack and a jump table [81]. The host core interfaces FIXER through a set of custom opcodes which signals function calls and returns, and indirect jumps. FIXER is an example of ISA-based CFI enforcement. While it has a lightweight execution overhead, FIXER requires access to a custom toolchain and rebuilding all the

sources that need to be protected. In contrast, TitanCFI works with a standard toolchain and can protect legacy binaries without the need for rebuilding.

DExIE [65] enforces CFI on the protected host core by implementing a hardware Enforcement Finite State Machines (EFSM) [81], coupled with a shadow stack. It represents an example of hardware monitor CFI enforcement. Similarly to FIXER [75] it uses custom-designed hardware IP to enforce the desired CFI policy, to minimize area occupancy and runtime overhead. Like our approach, it does not rely on any special opcode, and control-flow events are inferred by observing the stream of instructions executed by the host core. However, the policy requires custom hardware IP and interface toward the main core, whereas our solution uses software-defined policies which use standard bus interconnects.

PHMon [76] is a programmable hardware module which belongs to the category of the programmable co-processors. It consists of a match unit that transparently detects control-flow events by snooping the instructions retired by the host core and a programmable actions unit which enables the user to assign custom actions, like accessing memory or comparing values, to each detected pattern. Unlike the solution we propose, the action unit is not a fully programmable core, which limits the available CFI policies, and it exploits a custom interconnect to enable the communication between the host core and the hardware monitor. A significant difference, however, lies in the security guarantees offered by our solution. PHMon protects CFI metadata by storing them in a set of virtual memory pages, which the OS marks as reserved, but it lacks a mechanism to ensure the authenticity of such data in the event of an OS breach. On the other hand, our solution either keeps private data in the tamper-proof storage of the RoT, or, in case

of overflow, it exploits the available cryptographic accelerators to ensure authenticity of CFI metadata.

4.3 SoC Architecture

TitanCFI enhances the architecture of a SoA RISC-V platform for micro aerial vehicles [78]. The SoC features an host domain based on the CVA6 core, the OpenTitan RoT, and a programmable multi-core accelerator. The programmable accelerator is not considered for the purpose of CFI enforcement.

4.3.1 Host Domain Architecture

The host domain is built around the CVA6 core, whose microarchitecture is briefly described in Section 3.2.3 and discussed in more detail in [79]. In addition to CVA6, the host domain follows an architecture similar to the template presented in Section 2.2. It includes a peripheral subsystem, a 128 KiB LLC, and a crossbar interconnect that implements a 64-bit, low-latency AXI4 protocol, used to plugin the programmable accelerator, the OpenTitan RoT, and the L2 scratchpad memory.

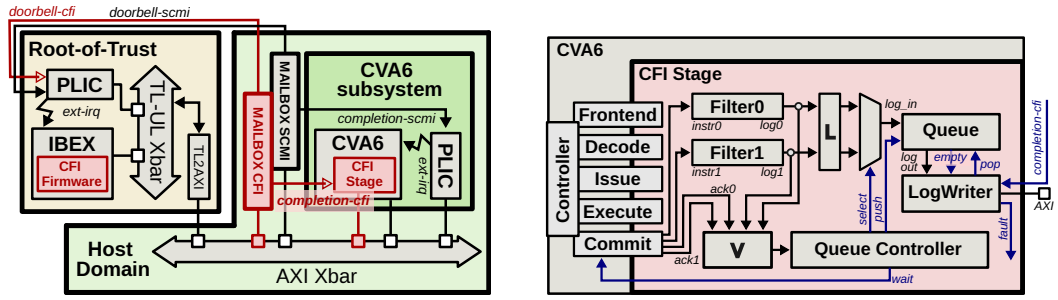
4.3.2 OpenTitan Root-of-Trust Architecture

OpenTitan is an open-source silicon RoT that implements a secure enclave aiming at securing sensitive data against hardware attacks, tampering, counterfeiting, and enabling the implementation of security mechanisms such as secure boot and remote attestation. The hardware architecture of OpenTitan includes (i) a secure microcontroller, (ii) on-board private memory, and (iii) a set of

cryptographic accelerators. The secure microcontroller is Ibex, a open-source RV32IMC MCU optimized for low-gate count, and designed for embedded control applications [80]. The cryptographic hardware accelerators efficiently execute compute-intensive security primitives, such as key generation, encryption/decryption, signature generation/verification, and hash calculation. OpenTitan is equipped with a 128KB scratchpad SRAM memory and an embedded flash memory enhanced with Error Correcting Code (ECC) and data & address scrambling, for enhanced security and reliability. In the reference system, OpenTitan is integrated in the SoC and it can access memory through a custom TileLink-to-AXI bridge, as detailed in [78]. Communications between the host domain and the RoT are mediated by a SCMI compliant mailbox. The mailbox consists of a set of general-purpose memory mapped registers meant for data sharing. Additionally, it features two registers, named Doorbell and Completion, which are meant to send an interrupt to the Ibex security microcontroller and to the CVA6 host core.

4.4 CFI Extensions and OpenTitan Firmware

This chapter proposes an enhancement to the reference SoC [78] and the host CVA6 core [79] to enable the CFI enforcement in the OpenTitan RoT. Figure 4.1 depicts the major architectural changes. First, we extend the CVA6 commit stage to (i) filter control-flow operations out of the stream of retired instructions, (ii) extract relevant metadata from the host core scoreboard, (iii) enqueue and forward them towards the RoT. Then, we expand the SoC communication system by adding a shared memory region dedicated to exchanging data between CVA6 and the RoT. Finally,



(a) SoC architectural modifications (b) CVA6 core modifications

Figure 4.1. Architecture of TitanCFI. The diagram highlights the proposed architectural modifications to the SoC (a) and the CVA6 core (b).

we develop a firmware to enforce CFI.

4.4.1 SoC Modifications — CFI Mailbox

CFI metadata extracted from the retired instructions are stored in a CFI Mailbox, where they can be read from the RoT. The design of the CFI Mailbox is analogous to the SCMI-like mailbox already present in the reference SoC [78]. We parametrize the general-purpose registers to be large enough to store the CFI metadata required to represent a single control flow instruction. When a new metadata is ready to be read, the enhanced CVA6 commit stage sets the doorbell register in order to trigger an interrupt in the RoT. Different from a regular SCMI-like mailbox, the completion register is not connected to the host domain interrupt controller, but directly to the commit stage of the CVA6 core. To indicate that a previously retired instruction has been checked, the result of the CFI enforcement policy can be read from the mailbox, signaling that the RoT is ready to read the next commit log.

4.4.2 Host Core Modifications

CFI Filters

The commit stage of the CVA6 core is extended to scan all the instructions retired by each commit port, and to select the control flow operations that need to be checked. Such operations are indirect jumps, function returns, and function calls. TitanCFI instanciates two CFI filters, one for each commit port. A CFI Filter takes a scoreboard entry as input, which is emitted by the commit port, and generates a commit log. A valid scoreboard entry represents an issued instruction which has been executed, and it is ready to be retired. From a scoreboard entry the CFI Filter verifies if the retired instruction is relevant to CFI, and it extracts useful metadata, called the commit log, which is needed for CFI enforcement. A commit log is a 224-bit packet containing four items of information: (i) instruction program counter, (ii) the uncompressed binary encoding, (iii) the next address, and (iv) the target address.

CFI Queue and Queue Controller

The CFI Queue is a FIFO which stores the commit logs extracted by the CFI Filters. The Queue Controller controls the CFI Queue push signal and, occasionally, it inhibits the CVA6 commit stage which eventually results in stalling the core. The Queue Control inhibits the commit stage if the CFI Queue is full, or if more that one commit port retires a control-flow instruction. This is required since the CFI Queue is implemented as a single entry FIFO, and it does not allow more than one commit log per cycle to be pushed into its internal storage. Since committing two control-flow instructions in the same cycle is a rare event, we do not expect it to impact the

performance of our design.

CFI Log Writer

The CFI Log Writer module implements a Finite State Machine (FSM) which pops commit logs from CFI Queue, and writes them to the CFI Mailbox through the SoC interconnect. When the FSM is idle, it waits for the CFI Queue to contain at least one commit log, and for the CFI Mailbox to be ready. Then, the Log Writer retrieves a commit log from the queue, divides it into data chunks of equal size, matching the interconnect data bus, which is 64 bits in our case, and initiates AXI transactions to transmit the commit log to the CFI Mailbox. The final AXI transaction sets the doorbell interrupt register and transitions the FSM into a waiting state and it remains there until the completion signal is asserted by the RoT firmware. Once the completion signal is received, the FSM reads the result of the CFI enforcement check from the CFI Mailbox and triggers an exception if any control flow violation is detected. Finally, the FSM returns to idle and it is ready to pop a new commit log from the CFI Queue.

4.4.3 OpenTitan Firmware Design

TitanCFI implements the desired CFI policy in C and embeds it into the interrupt service routine dedicated to the CFI Mailbox. Every CFI policy obeys the same common structure, which consists of three steps: (i) IRQ entry, (ii) policy enforcement, and (iii) IRQ exit. When the OpenTitan interrupt controller receives the CFI Mailbox interrupt, it wakes up the Ibex microcontroller, directs it to jump to the CFI interrupt service routine, reads the commit log from the CFI Mailbox, and extracts the commit log that describes

the control-flow event to be checked. Then, it parses the binary encoding of the control flow instruction to understand which control flow event it represents, among function call, function return, or indirect jump, and applies the programmed CFI enforcement policy. Finally, the Ibex writes a binary value to the first mailbox entry, to signal whether any control-flow violation has occurred, and sets the mailbox IRQ completion register, to signal that the check is over and OpenTitan is ready to read a new commit log from the mailbox.

4.5 Experimental Results

4.5.1 Experimental environment

We synthesized TitanCFI on FPGA, and simulated a set of benchmarks to estimate how our approach impact hardware utilization and runtime overhead. FPGA synthesis has been run on Xilinx Vivado 2020.2 targeting the Virtex UltraScale+ VCU118 system. The benchmarks comes from EmBench-IoT v1.0 and RISC-V-Tests, and were built with a standard RISC-V toolchain featuring GCC 12.2.0 and `-O3` optimization level.

4.5.2 OpenTitan firmware analysis

To measure the overhead of implementing CFI enforcement policies in the RoT we implemented a well-known return address protection policy called shadow-stack [82]. Our shadow-stack implementation parse the instruction binary to distinguish call from return instructions. In case of a call, the expected return address is extracted from the commit log and pushed into the shadow stack.

Table 4.1. Cycles required to implement the return address protection policy in OpenTitan

	Op.		Instructions [#]			Cycles [#]			Cycles [%]		
			IRQ	CFI	TOT	IRQ	CFI	TOT	IRQ	CFI	TOT
IRQ	CALL	Logic	8	15	23	59	27	86	23	10	33
		Mem. RoT	14	5	19	74	28	102	29	9	38
		Mem. SoC	2	4	6	22	48	70	10	19	29
		TOT	24	24	48	155	103	258	62	38	100
	RET.	Logic	8	15	33	59	45	104	21	16	37
		Mem. RoT	14	5	19	74	28	102	27	10	37
		Mem. SoC	2	4	6	22	48	70	9	17	26
		TOT	24	34	58	155	121	276	57	43	100
Polling	CALL	Logic	–	15	15	–	27	27	–	26	26
		Mem. RoT	–	5	5	–	28	28	–	27	27
		Mem. SoC	–	4	4	–	48	48	–	47	47
		TOT	–	24	24	–	103	103	–	100	100
	RET.	Logic	–	25	25	–	45	45	–	37	37
		Mem. RoT	–	5	5	–	28	28	–	23	23
		Mem. SoC	–	4	4	–	48	48	–	40	40
		TOT	–	34	34	–	121	121	–	100	100
Optimized	CALL	Logic	–	15	15	–	27	27	–	42	42
		Mem. RoT	–	5	5	–	5	5	–	08	08
		Mem. SoC	–	4	4	–	32	32	–	50	50
		TOT	–	24	24	–	64	64	–	100	100
	RET.	Logic	–	25	25	–	45	45	–	55	55
		Mem. RoT	–	5	5	–	5	5	–	06	06
		Mem. SoC	–	4	4	–	32	32	–	39	39
		TOT	–	34	34	–	82	82	–	100	100

If a return is detected, the return address is extracted from the commit log and compared with the value popped from the shadow stack. Any mismatch is reported as a security violation. In both scenarios, the shadow stack is checked for overflow or underflow and eventually saved (or restored) from main memory after having being authenticated using the cryptographic accelerators available in OpenTitan.

Table 4.1 reports the cost, in terms of instructions and cycles, of implementing the abovementioned policy in OpenTitan. It separates the instructions into `IRQ` vs. `CFI`, and `Logic` vs. `Memory-RoT` vs. `Memory-SoC`. The former split distinguishes instructions involved in IRQ handling, such as spilling registers to the stack and clearing the interrupt pending bit, from those implementing the CFI policy, such

as accessing the mailbox or pushing the return address in the stack. The latter split considers memory accesses, distinguishing between RoT private scratchpad and SoC memory, from other operations. The *IRQ* section of Table 4.1 shows the cost in cycles of implementing the OpenTitan firmware described in Subsection 4.4.3. The results shows that while the cost in cycles of implementing the CFI policy is non negligible, between 258 and 276 cycles per control-flow operation, the large part of it is overhead caused by the OpenTitan micro architecture. Two major non-idealities emerge from the result analysis: the high ratio of cycles spent in IRQ handling and the overhead imposed by memory accesses. Approximately 60% of the number of cycles involved in the implementation of the CFI policy is spent in IRQ handling. The analysis of the traces produced by the simulator shows that it takes 45 cycles from when the host core set the doorbell interrupt bit in the CFI mailbox to when the Ibex core wakes up from sleep. Moreover, since accessing the OpenTitan private scratchpad takes approximately 5 cycles per access, and Ibex spill and restore 6 registers, entering and leaving the IRQ results in 105 cycles overhead, independently from the CFI policy implemented. On the other hand, the number of instruction required to implement the CFI policy logic, between 48 and 58 opcodes, highlight how implementing return address protection in software is feasible and a potentially cheaper alternative to the design of fully customized hardware modules, assuming the RoT architecture is tuned to obtain a sufficiently high IPC.

The sections of Table 4.1 labelled *Polling* and *Optimized* report the RoT firmware performance when applying two possible optimizations targetting at optimizing the issues identified in the previous paragraphs. *Polling* firmware amortizes the cost of IRQ handling by performing some busy waiting cycles polling the doorbell bit of

the CFI Mailbox after an instruction has been checked. With such optimization, the CFI enforcement firmware does not pay the cost of executing expensive IRQ handling procedures and directly executes the CFI enforcement logic, in case a new control flow instruction is already ready in the CFI Queue. Such optimization requires no hardware modifications, and enforce the desired policy in 112 cycles on average, saving approximately 58% of overhead with respect to the approach which does not imply busy waiting loops.

A more aggressive optimization, represented by the *Optimized* section of Table 4.1, involves a partial redesign of the OpenTitan internal architecture to reduce the cost of accessing its private scratchpad. Substituting the internal OpenTitan interconnect with a low-latency interconnect would enable accessing the OpenTitan private memory and peripherals in a single cycle and the SoC memory in approximately 8 cycles, instead of 12. In this last scenario, enforcing return address protection requires 73 cycles on average, more that 70% less than the baseline *IRQ* firmware.

4.5.3 Runtime overhead

We run a set of benchmarks to assess the runtime overhead of software CFI enforcement in the RoT, and compare the results with the SoA. Slowdown is computed by simulating the RTL of the reference SoC [78] and extracting the cycle-accurate execution trace reporting the number of cycles required to commit each instruction. Then, we feed the obtained traces to a trace-driven model which emulates the latency required for CFI enforcement. We emulated three different latencies, matching the results obtained by the firmware analysis reported in Table 4.1, averaging the number of cycles required to process a function call and a function return: (i) 267 cycles, for the

Table 4.2. Runtime slowdown comparison with [65] and [75]

	Benchmark	Slowdown [%]				
		[65]	[75]	Opt.	Poll.	IRQ
EmBench	`aha-mont64`	48	n.a.	–	–	–
	`edn`	47	n.a.	1	1	2
	`matmult-int`	48	n.a.	–	–	1
	`ud`	48	n.a.	12	18	43
RISC-V Tests	`rsort`	n.a.		–	–	1
	`median`	n.a.		3	5	12
	`qsort`	n.a.	2	–	–	1
	`multiply`	n.a.		2	3	6
	`dhrystone`	n.a.		360	553	1318

IRQ firmware, (ii) 112 cycles, for the *Polling* firmware, and (iii) 73 cycles, for the *Optimized* RoT.

Table 4.2 compares the overhead of TitanCFI with the one imposed by two SoA architectures: DExIE [65] and FIXER [75]. In both cases, to establish a fair comparison, we constrained the CFI Queue to have depth 1, to emulate the behaviour of stalling the core as soon as a single control flow instruction is retired. The first section of Table 4.2 shows that TitanCFI imposes a runtime overhead lower than the one imposed by DExIE [65], for which we considered the best runtime overhead obtained fetching data from the relative paper. Notice that the authors of [65] report a reduction in the clock frequency of the tested cores, when interfaced with the DExIE security monitor. While compensating for clock reduction makes the overhead of DExIE [65] negligible, Table 4.2 highlights how enforcing CFI in the RoT leads to minimal overhead with respect to custom hardware monitor in 3 out of 4 benchmarks tested by [65].

Table 4.3. Statistics and slowdowns of EmBench-IoT and RISC-V Tests.

	Benchmark	Cycles	CF	Slowdown [%]		
				Opt.	Poll.	IRQ
EmBench	aha-mont64	2.51E+6	1.50E+1	–	–	–
	crc32	3.49E+6	1.50E+1	–	–	–
	cubic	1.10E+6	2.01E+4	46	107	390
	edn	4.23E+6	3.67E+2	–	–	–
	huffbench	3.49E+6	2.28E+3	1	3	11
	matmult-int	4.69E+6	2.05E+2	–	–	–
	minver	4.75E+5	4.50E+3	–	7	153
	nbody	1.21E+5	4.29E+3	163	301	849
	nettle-aes	5.20E+6	7.95E+2	–	–	–
	nettle-sha256	4.73E+6	8.57E+3	1	2	11
	nsichneu	5.24E+6	1.70E+1	–	–	–
	picojpeg	4.97E+6	2.14E+4	5	15	58
	qrduino	4.61E+6	4.35E+3	–	–	–
	sglib-combined	3.67E+6	2.62E+4	9	32	142
	slre	3.57E+6	6.69E+4	38	110	401
	st	1.47E+5	2.31E+2	–	–	2
	statemate	3.22E+6	2.75E+4	–	–	129
	ud	1.87E+6	2.98E+3	–	–	–
	wikisort	4.38E+5	7.69E+3	94	158	418
RISC-V Tests	dhrystone	4.57E+5	2.25E+4	260	452	1215
	median	2.53E+4	1.10E+1	–	–	–
	memcpy	1.20E+5	1.10E+1	–	–	–
	mm	1.41E+6	2.33E+5	1108	1752	4311
	mt-matmul	5.76E+4	2.38E+2	11	22	65
	mt-memcpy	4.08E+5	1.80E+1	–	–	–
	mt-vvadd	1.48E+5	3.30E+1	–	–	–
	multiply	3.72E+4	9.00E+0	–	–	–
	pmp	9.01E+5	5.90E+1	–	–	–
	qsort	2.68E+5	1.10E+1	–	–	–
	rsort	3.32E+5	1.10E+1	–	–	–
	spmv	1.67E+5	1.10E+1	–	–	–
	towers	2.01E+4	9.00E+0	–	–	–

Comparing TitanCFI with FIXER [75] is more complicated because the authors of [75] reports a 1.5% runtime overhead for their solution, without showing the breakdown of how such results were obtained. We observe that, with the exception of the `dhrystone` benchmark, TitanCFI has a mean overhead of 5% in the worst case, only marginally higher than the result reported by the authors of

[75].

While comparing the overhead of TitanCFI with the results obtained by [65] and [75] is useful to check how it compares with alternative SoA architecture for CFI enforcement, it does not provide a clear overview on the real performance degradation we can expect to get in a real-world scenario. We see two major criticalities: (i) only functions with a moderately low number of control-flow instructions are chosen, and (ii) the benchmark used are normally used to assess core performance, and they are not relevant to the problem of CFI enforcement. While this approach, like others in the SoA, does not face the second issue, it solves the first criticality by computing the overhead of TitanCFI on a larger pool of benchmarks, considering the whole EmBench-IoT suite, and most of RISC-V-Tests. Table 4.3 reports, for each benchmark, the number of cycles required to complete the execution, the number of control flow instructions retired, and the slowdown obtained by TitanCFI in the three version described in Subsection 4.5.2, using a CFI queue of size 8. While our solution requires tens of cycles to enforce CFI, and will require further optimization to be applied to benchmarks with a high number of control flow instructions, it gives $< 10\%$ overhead for most of the kernel tested. We see two main approaches to improve the performance of TitanCFI. First, ad-hoc static source code analysis techniques should be developed to recognise hot-spots in the code and there selectively inline function calls to alleviate the pressure on the CFI enforcement module. Additionally, TitanCFI should be enhanced to enforcing CFI per thread, to selectively protect only the processes exposed at the boundary of the system, dealing with potentially tainted data and inputs. All the benchmarks in the Table 4.3 which impose the highest overhead are not kernel which requires CFI enforcement, since they aim at performing linear

algebra or physics computation, and they are not expected to be protected in a real-world scenario.

4.5.4 Hardware utilization overhead

Table 4.4. Hardware resource utilization with respect to DExIE [65]

		w.o CFI	CFI	Δ	Overhead
Host	LUT	5.02E+4	5.14E+4	1.16E+3	+2.3 %
	Registers	3.04E+4	3.22E+4	1.77E+3	+5.8 %
	BRAM	6.60E+1	6.60E+1	–	–
SoC	LUT	4.41E+5	4.41E+5	1.33E+3	+0.3 %
	Registers	2.57E+5	2.58E+5	2.19E+3	+0.9 %
	BRAM	2.68E+2	2.68E+2	–	–
[65]	LUT	4.66E+3	8.02E+3	3.36E+3	+72.1 %
	Registers	3.09E+3	5.33E+3	2.24E+3	+72.5 %
	BRAM	1.36E+2	1.42E+2	6.00E+0	+4.4 %

From FPGA synthesis we report how many LUTs, registers, and BRAMs TitanCFI requires to evaluate how the architectural modification we propose impacts on the resource utilization of the SoC and of the host core. Table 3.1 shows that our architectural modifications have negligible impact on hardware resource utilization, equal to $< 1\%$ on the entire SoC, and $< 6\%$ considering only the host core. Comparing with the best result reported by [65], our solution uses 60% less LUTs, 2% less registers, and does not need any BRAM slice. Furthermore, our proposed modifications do not impact on the maximum operating frequency of the SoC.

4.6 Summary

This chapter introduced TitanCFI, a study to investigate the possibility of implementing CFI enforcement policies in software in the

RoT, enabling the possibility of implementing any policy without designing and integrating custom hardware monitors, or modifying the host core pipeline. Moreover, it indicates how to take advantage of the properties of RoT such as tamper-proof memory and cryptographic accelerators for enhancing the security guarantees provided by the CFI enforcement scheme. We prove our solution by extending the architecture of a modern RISC-V SoC for secure systems [78]. We demonstrate the proposed approach exhibits minimal hardware overhead and does not compromise timing. Tests on EmBench-IoT and RISC-V-Tests benchmark suites show that the majority of benchmarks impose no, or $< 10\%$ runtime overhead. The overhead of our system is comparable to the one imposed by alternative enforcement schemes in literature [65, 75] for most benchmarks.

Chapter 5

Acceleration of Brain-Inspired AI through Sparse Streaming RISC-V ISA Extensions

5.1 Introduction

The growing deployment of AI workloads at the edge places stringent constraints on energy consumption, memory footprint, and real-time responsiveness, which are particularly critical in embedded and CPS. In this context, AI models that can exploit temporal sparsity and event-driven computation are especially attractive, as they offer the potential to reduce both computation and data movement while preserving competitive task-level accuracy. SNNs are at the forefront of neuromorphic computing, providing compact and energy-efficient AI models [15, 83]. Recent studies have demonstrated that SNNs are capable of achieving competitive performance in image classification and object detection tasks with a reduced power envelope and memory footprint w.r.t. their non-spiking counterparts [42, 84, 85]. To address their event-driven nature, spike-based communication between neurons, and complex activation functions, several acceler-

ators and neuromorphic processors have been developed. Among these, both Von Neumann architectures and non-Von Neumann architectures have been proposed. The latter are composed of neurons and synapses arrays [86], and can be grouped into analog and mixed-signal processors [87, 88], Globally Asynchronous and Locally Synchronous (GALS) processors [16, 89], and more traditional Digital Fully-Synchronous (DFS) accelerators [14, 42, 84, 85, 90]. Analog neuromorphic processors struggle with precision and noise issues as their continuous signals are vulnerable to voltage and temperature variations [91], leading to less accurate computations compared to digital platforms. GALS processors are very complex to design due to the low maturity of asynchronous EDA tools [15]. Both types of processors also face barriers to integration with existing infrastructure, often requiring entirely new software stacks [15]. DFS accelerators face fewer integration issues but are often constrained by the types of SNN models they can support, limited by their network topology or their reliance on hardwired neuron models, which frequently use fixed-point arithmetic with highly-reduced precision [14, 85, 90].

Moreover, quantization techniques in SNNs are not yet as stable as their ANNs counterparts and can lead to significant accuracy degradation [12, 84]. Indeed, while achieving marginal accuracy reduction in simple tasks such as MNIST classification [16], Zanatta et al. [12] demonstrate that in a drone obstacle avoidance task, the SNN outperforms the ANN when using floating-point (FP) computation. When the SNN is quantized from FP32 to 4-bit integers, it is no longer able to complete the task. Therefore, FP computation is still beneficial in SNNs.

Additionally, the rapid evolution of the SNNs ecosystem requires the embrace of programmable and general-purpose (GP) solutions to

support flexible model exploration. However, traditional CPUs, face challenges with the event-driven nature of SNNs, resulting in sparse data structures [15] that cause irregular memory access patterns and poor utilization of processing elements [92].

Recently, *Stream Registers* (SRs) [92–94] have emerged as a hardware extension for CPUs to address the Von Neumann bottleneck and the associated inefficiencies with memory accesses. SRs map streams of memory access directly to reads or writes of architectural registers, with address generation and data movement handled by dedicated hardware. They maximize bandwidth utilization on memory-bound workloads by decoupling memory accesses from computation and continuously streaming useful data, freeing the host processor from address calculations and enabling high floating-point-unit (FPU) utilization and compute throughput. In addition to affine address patterns, SRs can also support *indirect* streams [92, 93] using a base address and an index array to gather or scatter data, which can significantly accelerate the irregular memory accesses of sparse workloads.

In this Chapter, we present SpikeStream, the first exploration of neuromorphic processing acceleration on a multi-core streaming architecture. In contrast to the current State-of-the-Art (SoA) that requires expensive and non-flexible hardware units, our approach is software-based and runs on programmable RISC-V processors with enhanced ISA. It leverages the low-overhead streaming, SIMD, and hardware-loop extensions of an RV32G parallel compute cluster [44], to accelerate SNN computation and maximize FPU utilization. We first implement a parallel FP SIMD baseline by adopting a compressed representation for the sparse input feature maps (ifmaps). We identify the *indirection* operation used to gather weights associated with input spikes as the main source of inefficiency in

neuromorphic processing on CPUs, leading to frequent address computations, irregular memory accesses, and loop control overhead. SpikeStream optimizes SNN computation by leveraging the available [92] extensions in the proposed architecture to address memory inefficiencies caused by the indirection operation. When compared with the parallel baseline implementation of the SNN computation, SpikeStream improves runtime inference on an S-VGG11 of 7.29× and introduces an energy-efficiency gain of 5.68× in FP8, increasing the FPU utilization from 9.28% to 52.3%.

When compared with neuromorphic processors, SpikeStream achieves competitive performances, outperforming Loihi by 1.31× in FP16 and 2.38× in FP8, and introducing an energy-efficiency gain over LSMCore of 2.37× in FP16 and 3.46× in FP8, demonstrating that our approach can significantly improve the performance, utilization, and energy-efficiency of neuromorphic processing even without ad-hoc hardware.

5.2 Background

5.2.1 Spiking Neural Networks

SNNs employ spiking neurons as their fundamental computational units. The Leaky Integrate-and-Fire (LIF) model is a widely adopted model of a spiking neuron since its simplicity is appealing in deep

learning tasks. The following equation governs the LIF model:

$$\begin{cases} i_m(t) = \sum_{n=1}^{N} s_{i,n}(t) w_n \\ v_m(t) = v_m(t-1)\alpha + r i_m(t) - v_{rst} s_{o,m}(t) \\ s_{o,m}(t) = \begin{cases} 1 & \text{if } v_m(t) \geq v_{th} \\ 0 & \text{otherwise} \end{cases} \end{cases} \tag{5.2.1}$$

where, $s_{i,n}(t)$ and $s_{o,m}(t)$ represent the input spike at synapse n and the output spike of the m-th neuron at time step t, respectively. w_n is the weight associated with synapse n, and N is the total number of synapses. $i_m(t)$ and $v_m(t)$ denote the input current and neuron state of the output neuron m, respectively. v_r^{st}, v_{th}, r (usually set to 1), and α are the reset potential, membrane threshold, membrane resistance, and decay factor, respectively.

5.2.2 The multi-core streaming architecture

The adopted multi-core streaming architecture is the open-source *Snitch* cluster [44], a programmable many-core accelerator targeting energy-efficient FP compute. It contains eight 32-bit RV32G *worker* cores, each featuring a SIMD-capable FP64 FPU kept busy by three SRs and a hardware loop that decouples the FPU and integer core. These two subsystems are synchronized by explicit move instructions, allowing Snitch to overlap independent integer and FP instructions. SRs also expose shadow registers to overlap configuration and computation. The cluster also provides a shared 128 KiB 32-bank scratchpad memory (SPM), accessible to worker cores through a single-cycle logarithmic interconnect, and a shared 8 KiB L1 instruction cache. An additional *DMA* core without SRs and FPU controls a 512-bit DMA engine used to asynchronously move

large tiles of data between the cluster's SPM and global memory.

The SRs map buffered streams to and from the cluster's SPM [94] directly to FP register reads and writes, handling all necessary address calculations in hardware. All three SRs in each worker core are capable of $\leq$4D affine streams, enabling near-constant FPU utilization on dense workloads exhibiting regular memory access patterns. Furthermore, two of them support 1D indirect streams with 8-, 16-, or 32-bit indices in SPM [92], which can significantly accelerate workloads involving sparse and irregular data structures. SRs introduce minimal overhead, taking up only 1.8% of total cluster area. Figure 5.1 shows a block diagram of the Snitch multi-core cluster.

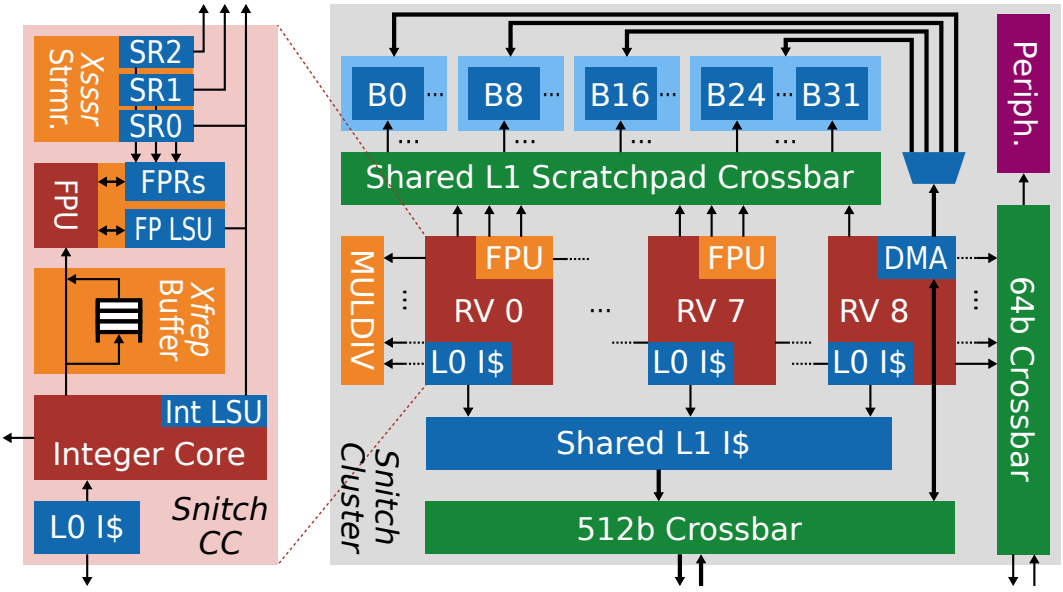


Figure 5.1. Snitch Cluster internal architecture.

5.3 SpikeStream Software Architecture

This section presents the proposed optimized SpikeStream SNN inference kernel leveraging the architecture presented in section 5.2.2. We progressively describe the key optimizations implemented: Tensor compression (TC - Section 5.3.1); Task parallelization (TP - Section 5.3.2); Data parallelization (DP - Section 5.3.3); Tiling and double buffering (DB - Section 5.3.4) and streaming acceleration (SA - Section 5.3.5).

Listing 1 Pseudocode of the spatial iterations on the receptive field (RF).

```
# core_rf_start, w_baddr set by workload-stealing scheduler
# Spatial iterations on RF
for i in range(k * k):
    # Select ifmap row
    if i % k == 0 and i != 0:
        if_rptr += 1
    # Compute spatial coordinate for s_ptr_i
    coo_ptr = if_rptr * if_w + core_rf_start + (i % k)
    # Compute stream/SpVA base address
    s_baddr = s_ptr_i[coo + 1]
    # Compute stream length
    s_len = s_baddr - s_ptr_i[coo]
    # Execute SpVA
    for j in range(s_len):
        # Accumulate on output neuron's input current
        ic += w[c_idcs_i[s_baddr + j] + w_baddr]
# Apply activation function
act_fun(ic, v, vth, c_idcs_o, s_ptr_o)
```

5.3.1 Tensors compression

In this manuscript, we propose to adopt a memory-efficient fiber-tree-based compression format for ifmaps, derived by the Compressed

Listing 2 RISC-V assembly implementation of the SpVA loop.

```
SpVA: lw     t0, 0(%c_idcs_i)
      slli   t0, t0, 3
      add    t0, t0, %w
      fld    ft1, 0(t0)
      addi   %c_idcs_i, %c_idcs_i, 2
      addi   %iter, %iter, 1
      fadd   %ic, ft1, %ic
      bne    %iter, %s_len, SpVA
```

Listing 3 SpikeStream SpVA pseudocode.

```
if s_len != 0:
    sr_set_indir(SR1, &w[w_baddr])
    sr_set_idcs(SR1, &c_idcs[s_baddr])
    sr_set_bound(SR1, s_len)
    frep 1, %s_len
        ic += sr_read(SR1)
```

Sparse Row (CSR) data representation. Indeed, we do not need to store the values of non-zero elements as they are all "1". In convolutional layers, binary channels are compactly represented using an index array (`c_idcs`) to mark the positions of active neurons, while a spatial pointer array (`s_ptr`) aggregates information on the spiking neuron count across spatial dimensions. In fully connected (FC) layers, compression is achieved with a single index array and a count of spiking neurons. Furthermore, by processing ifmaps sequentially, we avoid timestamping each input spike, leading to a more compact data representation compared to the address-event-representation (AER) format used in neuromorphic processors, which requires absolute coordinates and a timestamp for each spike [14]. This compressed format allows the replacement of costly multiply-accumulate operations with less power-hungry add operations. The neuron state tensor is held in a dense representation

as each value is updated at each timestep, resulting in a low degree of sparsity to motivate compression. In this work, we assume that all input tensors, including neuron states, weights and ifmaps are stored at the higher level of the memory hierarchy.

5.3.2 Task Parallelization

We parallelized the computational kernel to match the number of working cores in a Snitch Cluster. In the convolutional layers, each worker core is assigned a different receptive field (RF), which is processed in a depth-first manner. To evaluate $i_m(t)$, the kernel initially reads $v_m(t)$ into an FPU register and then computes a Sparse-Dense Vector Accumulation (SpVA) for each spatial iteration in the RF. For each SpVA, the weights associated with an input spike are retrieved through a sequence of indirection operations over the index vectors and accumulated in the register holding $v_m(t)$. The computation of each $i_m(t)$ involves $k_h \times k_w$ SpVAs, with k_h and k_w representing the spatial dimensions of the filters. The number of indirection operations required for each SpVA equals the number of spiking neurons across the channels in each spatial dimension. To address the workload imbalance resulting by using a compressed data representation of ifmaps, we introduced a workload stealing mechanism, in which each core, upon completion of its assigned RF, moves without halting to the next available unprocessed RF via an atomic tagging operation. Finally, to increase data reuse and reduce transfers to and from external memory, we implement layer fusion with the activation function. Listing 1 illustrates a streamlined pseudocode of the proposed baseline compressed convolution for SNNs. The two outer loops manage the control-heavy computations required to generate the base addresses for both the weights and

`c_idcs` which are then accessed by the innermost loop responsible for executing the SpVA. Without any optimizations, the SpVA yields to the assembly code in Listing 2, where only 1 instruction does useful computation, while the other 7 instructions are required for address calculations, memory accesses and loop control.

5.3.3 Data parallelization

Depending on the chosen precision, the SIMD capabilities of the FPU are leveraged to parallelize the activations across the output channels on each core's FPU lanes. To achieve this, we adopt a batched *HWC* memory layout for the weight tensor, arranging the weights from different filters in contiguous memory locations, with the number of batched weights corresponding to the SIMD width. Since the activations are parallelized across different FPU lanes, after thresholding, a series of SIMD-width bit-masking and branching operations are required to extract the output neuron result. In the case of a spike, the ofmap's `c_icds` and `s_ptr` SPM buffers are atomically updated.

Figure 5.2 shows an SNN convolutional at timestep t_0 layer with uncompressed tensors. Figure 5.3 shows the corresponding SpikeStream DP and TP dataflow using compressed ifmaps and ofmaps. The `next_rf` index is used to keep track of the RFs processed and to implement workload stealing scheme.

5.3.4 Tiling and double buffering

We implement double buffering in all kernels to mask the latency associated with memory transfers and maximize utilization. The kernel process begins with an initialization phase where a tile for

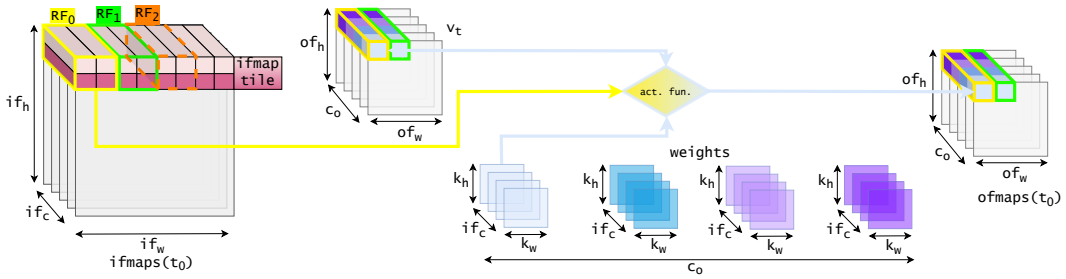


Figure 5.2. SNN convolutional layer at timestep t_0.

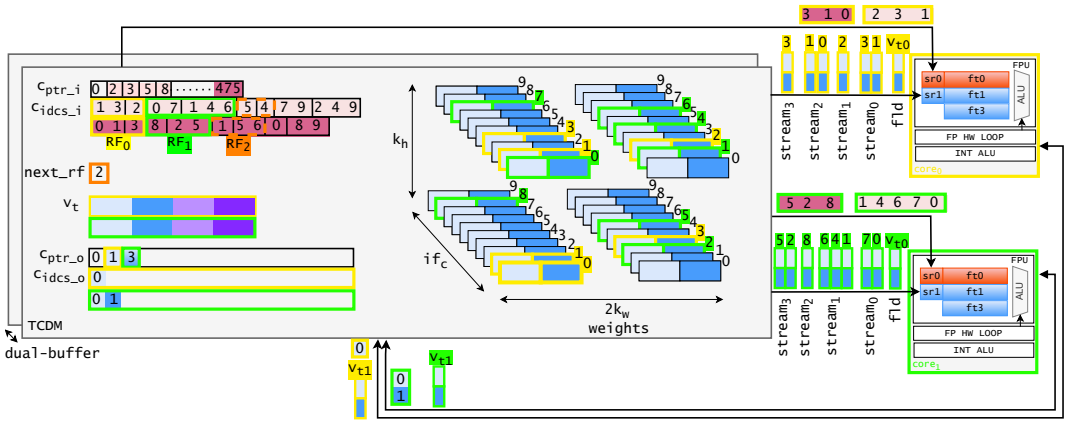


Figure 5.3. SpikeStream dataflow assuming $k_h = k_w = stride = 2$, two *worker cores*, and FP32 weights.

each input tensor is loaded into the cluster's SPM. In addition, according to the sizes of the input tiles, SPM buffers are allocated to host the compressed ofmaps computed by the cores. These buffers are sized for the worst-case scenario, assuming a zero-sparsity output. Once this initialization phase is complete, kernel computation begins. Simultaneously, the DMA core preloads the next data chunk from external memory into the SPM in preparation for the next computation phase. To avoid excessive fragmentation of the output `c_idcs` vectors and to reduce data movement associated with spatial pointers, we first double-buffer the weights and then the ifmaps. This approach ensures that the compressed ofmap tile is fully popu-

lated before it is transferred back to external memory. Once all the weights have been processed for the ifmap tile, the DMA core joins the ofmap `s_ptr` elements before copying out the results. Furthermore, the adopted compression format, which aggregates spiking neuron information into spatial dimensions, allows a compressed ifmap tile to be transferred within a single DMA request. The ofmap `c_idcs` vectors still have to be transferred upward individually due to the fragmented output buffer resulting from worst-case allocation caused by dynamic sparsity.

5.3.5 Streaming acceleration

We accelerate neuromorphic processing by reducing the overhead of repeated indirection operations within the SpVA loop, using the SRs to map all indirect weight loads to indexed stream reads. At the start of each SpVA, we configure an indirect SR to point to the base address of the ifmap `c_idcs` vector in the dedicated SPM buffer, corresponding to a spatial location within the RF, and the base address of the associated grouped weight tensor. We use `s_ptr` to compute the loop trip count to set the stream boundaries and configure the FP repetition buffer on an add operation to ensure continuous streaming and accumulation of the weights. This allows SR's hardware units to completely manage the SpVA's address generation and memory accesses. Meanwhile, the hardware-loop unit handles loop control and decouples the FPU from the integer core, allowing it to advance control processing outside of the SpVA to set up the next stream iteration via SR's shadow registers. Listing 3 reports SpikeStream SpVA pseudocode.

5.3.6 Spike encoding

When working with RGB images rather than event-cameras, it's necessary to convert the images into spikes. Most SNNs achieve this by using the first convolutional layer to handle the conversion, where the raw image values are interpreted directly as input currents of the layer [95–97]. In this case, we prefer a dense representation format with *HWC* storage for the first input tensor. In SpikeStream, we reshape this tensor on the fly through a 2D DMA transfer, reorganizing it with an *im2row* transformation and converting the convolution into a matrix multiplication (matmul). This allows for easier parallelization across cluster cores by output channel. Each dot product associated with an RF is accelerated using two affine SRs: One for the input current and the other for the weights.

5.4 Evaluation

We synthesized the Snitch cluster architecture in GlobalFoundries' 12LP+ FinFET technology using Fusion Compiler 2022.03,The placed and routed design is shown in Figure 5.4. Runtime measurements are based on cycle-accurate simulations of the Snitch cluster's register-transfer-level (RTL) description with Questasim 2022.3. All the code tested is compiled with a customized LLVM 12 toolchain for Snitch, with -O3 optimizations. Simulation traces are then used to extract runtimes and utilization metrics. Energy estimations are obtained by executing the computational kernels in a post-layout gate-level simulation at the cluster's clock speed of 1 GHz. The obtained switching activity is used to estimate power consumption in PrimeTime-2022.03 under typical operating conditions of 25 °C and a core supply voltage of 0.8 V.

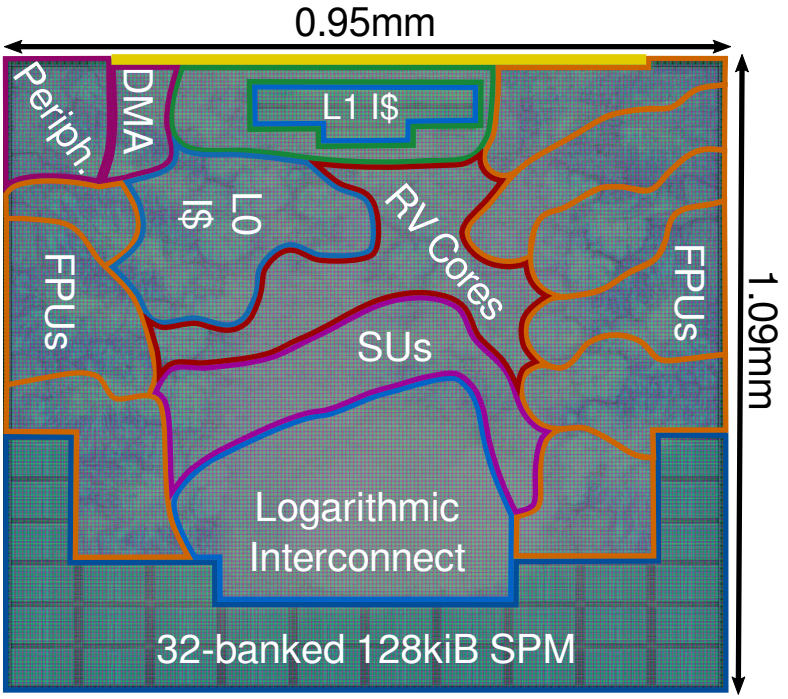


Figure 5.4. Place and Routed Snitch Cluster in GF12LP+ FinFET technology

In this section, we evaluate SpikeStream by comparing it against a multi-core SIMD baseline (implementing only the TC, TP, DP, and DB optimization presented in Section 5.3 to assess the speedup and energy savings achieved by the streaming ISA (SA - Section 5.3.5), and compare SNN inference energy and latency w.r.t. SoA neuromorphic accelerators (Section 5.4.3).

All the comparisons have been made on a low-latency, single-timestep, S-VGG11 architecture, trained with temporal backpropagation, for image recognition on the CIFAR10 dataset adapted on the work presented by authors [97]. The network performs spike encoding using the first layer. To account for the dynamic sparsity of the ifmaps, we conducted our evaluation over a batch of 128 input images, and we reported the computed standard deviations and the

average of each considered metric.

5.4.1 Performance and memory footprint

In this subsection, we evaluate the performance and memory footprint. Figure 5.5 shows the average memory footprint, measured in kB, required to store the ifmaps using both the AER format and our CSR-based format, along with the average firing activity across the various S-VGG11 layers. In both cases, we assume 16-bit values to represent indices and coordinates. The CSR format achieves an average memory footprint reduction of approximately 2.75× across the different layers of the network when compared to the AER format.

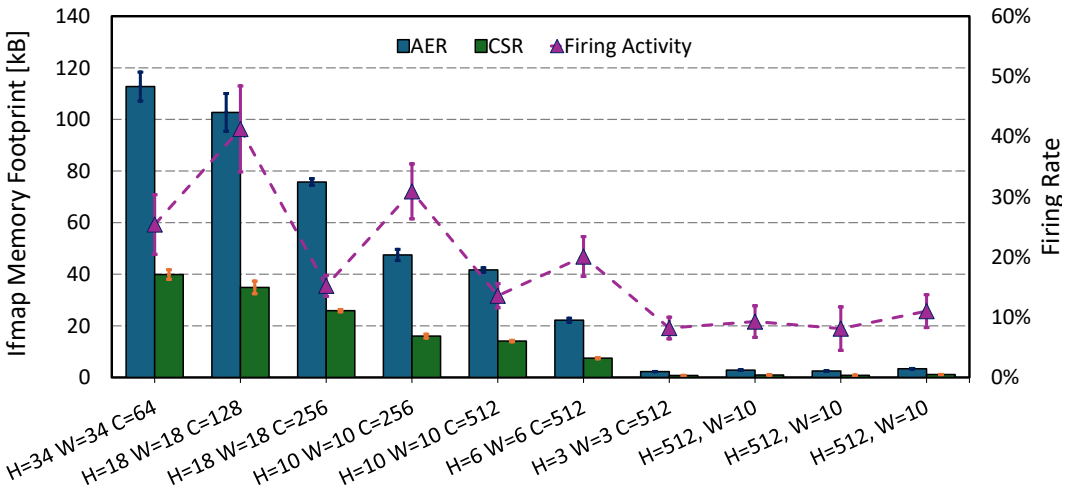


Figure 5.5. Average firing rate and memory footprint of ifmaps across S-VGG11 layers.

Figure 5.6 shows the average FPU utilization and IPC distribution across the layers of the network for both SpikeStream and the Baseline with FP16 arithmetic. The first layer, responsible for spike encoding, exhibits the highest FPU utilization for the baseline as

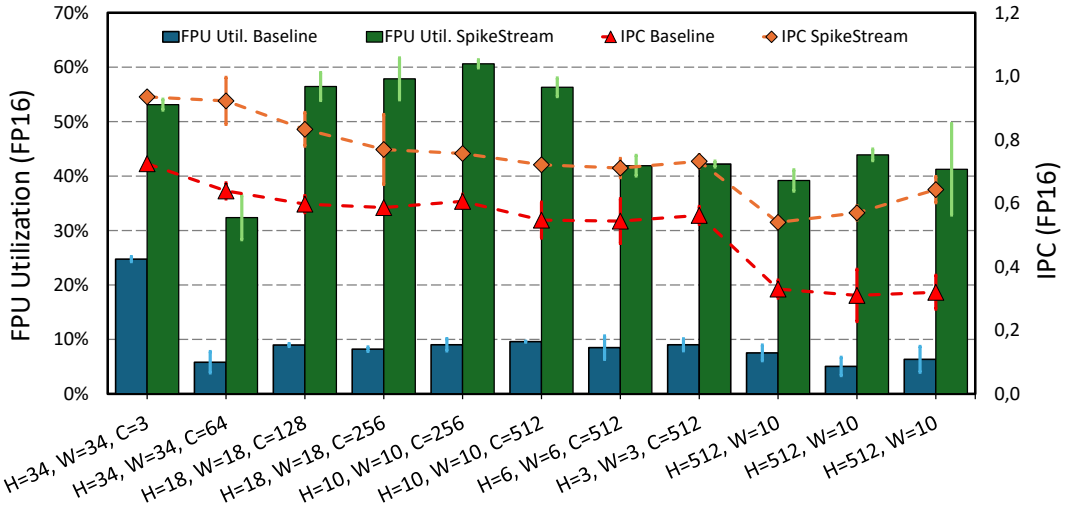


Figure 5.6. Average FPU utilization and per-core IPC for both code variants in FP16 across S-VGG11 layers.

convolution is implemented with dense matmul, leading to more regular memory accesses and reduced control computations for the integer core. SpikeStream, using two affine SRs, increases utilization from 24.8% to 53.1%. The second layer exhibits the lowest SpikeStream FPU utilization, primarily due to the reduced number of channels and the sparsity, which further shortens the SpVA stream, not allowing for complete computation overlap and the continuous issue of indirect streams. Indeed, low spike counts result in short stream lengths, leaving the FP pipeline underutilized and resulting in execution time being dominated by the integer pipeline of the snitch cores, which manages the workload-stealing scheduler and the computation of stream base addresses. In the other convolutional layers, the increasing sparsity is counterbalanced by the larger depth sizes, resulting in an average FPU utilization increase of 6.58×. In the FC layers, the extreme sparsity leads to a marginally lower utilization improvement of SpikeStream.

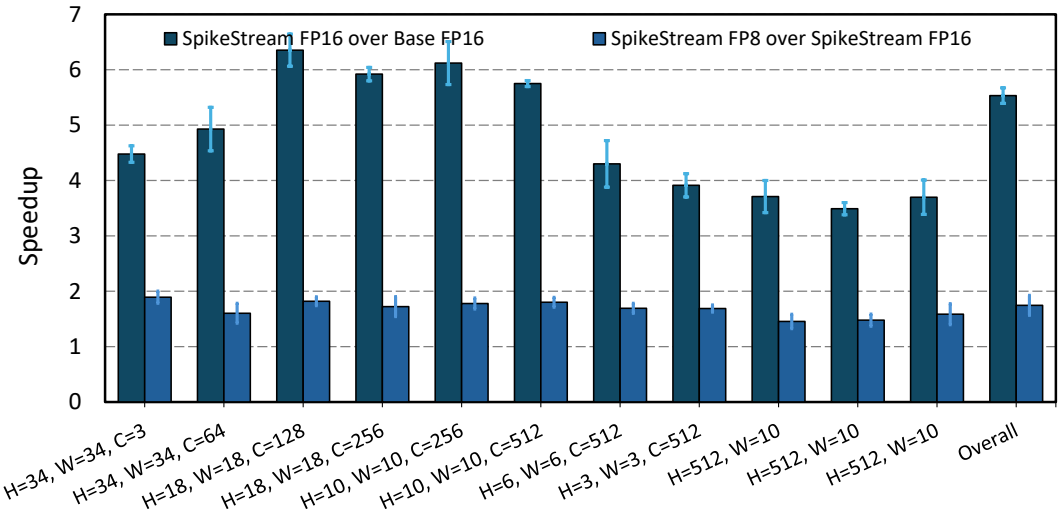


Figure 5.7. Average speedup across S-VGG11 layers.

Figure 5.7 shows the average speedup of SpikeStream when FP16 and FP8 are used w.r.t the baseline in FP16. SpikeStream FP16 achieves an average speedup of 5.62× w.r.t. the baseline for the full S-VGG11 inference, thanks to the SRs and hardware loops utilized in SpikeStream. Furthermore, in the first two layers of the network, we observe a smaller speedup in SpikeStream FP16 compared to deeper layers. This is primarily due to the limited decoupling between the integer core and FPU, caused by shorter stream lengths in these early layers, which result from their smaller depth sizes and are further reduced by sparsity in the second layer. From the third to the sixth network layer, we observe the highest speedup, approaching the ideal of 7×. Here, as with FPU utilization, the sparsity is counterbalanced by the larger depth sizes, allowing full overlap between the integer core and the FPU. The gap to the ideal speedup is mainly due to instruction cache misses, and conflicts in the SPM interconnect resulting from the random access patterns of indirection. The speedup obtained with SpikeStream

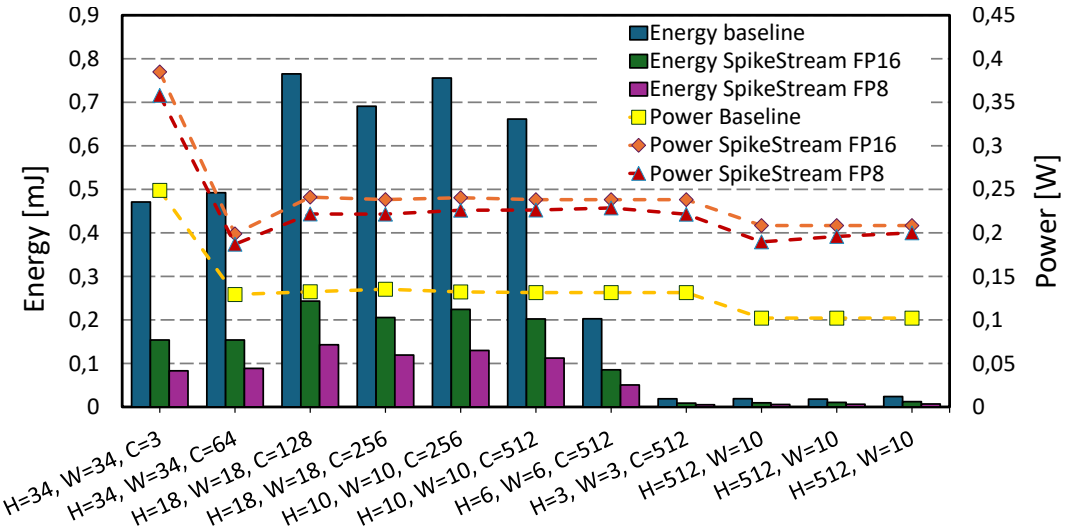


Figure 5.8. Average energy consumption and power for each layer of the network using both code variants in FP16 and FP8.

in FP8 compared to SpikeStream in FP16 is 1.71× for the overall network runtime, falling slightly below the ideal of 2×, as two additional iterations are required to unpack the output spikes after the activation function's thresholding.

5.4.2 Energy

Figure 5.8 shows the average energy and power consumption across the various layers of the network for the FP16 baseline, as well as SpikeStream in both FP8 and FP16 formats. The power required in the first layer is higher than in the other layers due to its increased computational intensity, as it involves matmul with multiply-accumulate operations, rather than simpler add operations as in the subsequent layers. Furthermore, the SpikeStream implementations use two affine SRs, instead of one indirect SR used in the following layers, leading to higher power consumption. Between the second and eighth layers, the power consumption for the evaluated

inference kernels (SpikeStream FP8, SpikeStream FP16, Baseline FP16) remains nearly constant, as the computational kernel is the same, with average values of 0.1319 W, 0.233 W, and 0.219 W for the FP16 baseline, SpikeStream FP16, and SpikeStream FP8, respectively. SpikeStream FP8 also consumes an average of 6.7% less power than SpikeStream FP16. This reduction is attributed to the way the FPU handles narrower data formats by separating execution units for each format and clock-gating idle slices, reducing power consumption.

Regarding energy consumption, it is primarily concentrated in the convolutional layers of the network, accounting for an average of 82.8% of the total energy consumption, since the FC layers are smaller and highly sparse. The average efficiency gains in total inference energy consumption are 5.67× for SpikeStream FP8 compared to the baseline, 3.25× for SpikeStream FP16 compared to the baseline, and 1.74× for SpikeStream FP8 compared to SpikeStream FP16.

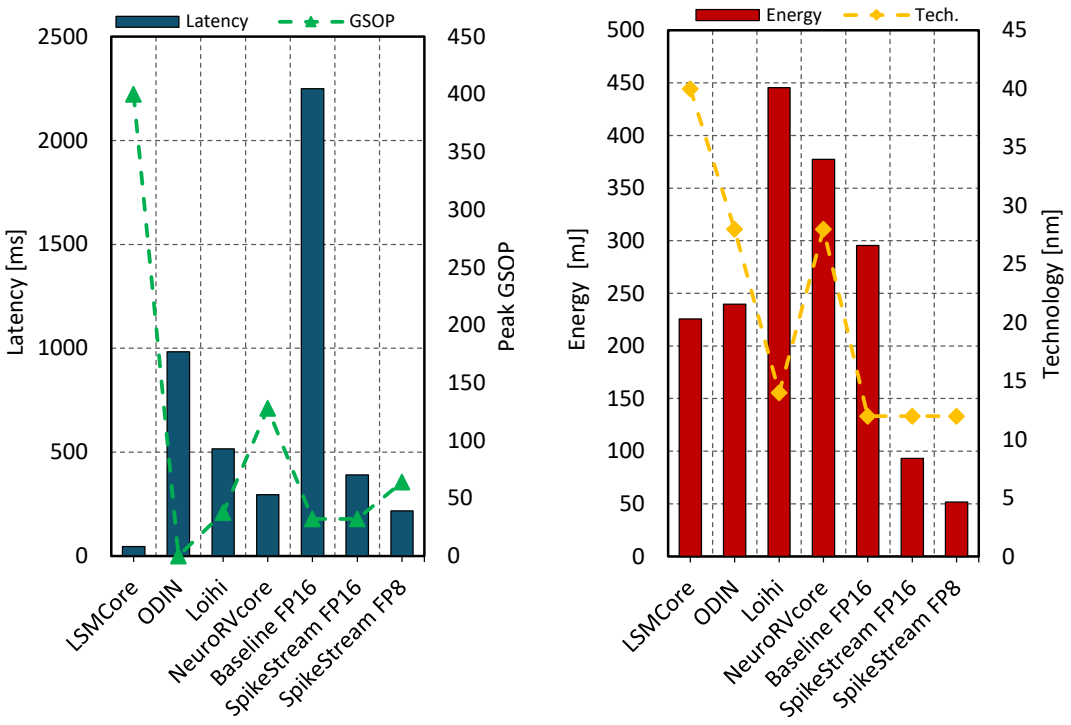
5.4.3 Comparison with SoA neuromorphic accelerators

In this section, we compare our results with those presented in [15], which evaluates the energy and runtime for the sixth layer of an S-VGG11 architecture on the CIFAR10 dataset over 500 timesteps on four SoA neuromorphic accelerators: Intel's Loihi (37.5 GSOP, 1-64 bits, 14-nm) [89], ODIN (0.038 GSOP, 4 bits, 28-nm) [16], LSMCore (400 GSOP, 4 bits, 40-nm) [90], and NeuroRVcore (128 GSOP, 4 bits, 28-nm) [15]. To ensure a fair comparison, we adapt the neural network accordingly and extend our simulations to match the specified number of timesteps. Figure 5.9a shows the performance comparison with neuromorphic accelerators. From the figure, we

can notice that among the SoA, LSMCore is the fastest while ODIN is the slowest. This can be easily explained by the peak GSOP of the two architectures, with LSMCore being more than four orders of magnitude more performing. When compared with the proposed architecture, which has 6.25× (w. FP8 arithmetic) lower peak SOP, the proposed FP16 baseline implementation is the slowest, with a runtime of 2516.72 ms. However, SpikeStream with FP8 arithmetic is the second fastest at 217.14 ms, and only 4.71× slower than LSMCore, which achieves a latency of only 46.08 ms. This significant result demonstrates the effectiveness of the proposed SR acceleration technique for SNNs. This is even more visible when comparing the energy consumption which is reported in Figure 5.9b. Our FP16 and FP8 SpikeStream implementations outperform LSMCore (which has the highest energy efficiency among SoA solutions) by consuming 2.37 and 3.46 times less energy, respectively. Despite the different numerical precision which is lower for LSMCore, this can be justified by the more efficient design and technology node of SpikeStream.

5.5 Summary

In this chapter, we presented the first exploration of SNN acceleration on a multicore streaming architecture. SpikeStream leverages (i) a memory-efficient compression format based on the CSR to represent the sparse ifmaps of SNNs, and (ii) an optimization technique that exploits both affine and indirect SRs combined with decoupling FP hardware loops to accelerate computation and maximize utilization. SpikeStream is able to increase utilization to an average of 52.3% in FP16, achieving a speedup of 7.29× in FP8 and an energy-efficiency gain of 5.68× in FP8 when compared to a



(a) Performance comparison.

(b) Energy comparison.

Figure 5.9. Comparison with neuromorphic processors on the 6^{th} layer of S-VGG11 over 500 timesteps.

non-streaming implementation.

Compared to SoA neuromorphic processors, SpikeStream achieves competitive performance in FP8 with a single-layer runtime of 217 ms, compared to 46 ms for LSMCore, which only operates on 4-bit integers. SpikeStream outperforms LSMCore with an energy-efficiency gain of 3.46× in FP8.

Future developments will focus on automatic SpikeStream code generation and enhancing SRs with strided indirect execution to enable higher degrees of computation overlap, further enhancing utilization with extremely sparse ifmaps.

Chapter 6

Memory Hierarchy Design for Programmable Acceleration in Heterogeneous RISC-V SoCs

6.1 Introduction

Memory subsystems in modern embedded and mobile SoCs are composed of multiple hierarchical levels of on-chip and off-chip storage, each serving distinct latency, bandwidth, and capacity roles. Tightly coupled on-chip SRAMs provide deterministic access for time-critical workloads, while higher levels of the hierarchy, ranging from shared on-chip buffers to external DRAM controllers, enable larger data footprints and sustained throughput.

Two principal paradigms dominate on-chip memory organization: cache-based and scratchpad-based hierarchies. Caches offer transparent data management and ease of programming, but introduce variability in latency and energy due to tag handling and dynamic replacement. SPMs, conversely, are explicitly managed by software or DMA engines, providing deterministic timing, lower area, and reduced energy consumption at the cost of increased

software responsibility [29]. For irregular workloads such as sparse matrix operations and event-driven neural computations—e.g., the spiking neural networks discussed in Chapter 5—traditional cache hierarchies often exhibit poor efficiency due to limited spatial and temporal locality, whereas SPM-centric or hybrid cache–SPM designs enable explicit data tiling, controlled reuse, and predictable access latency [33, 98, 99].

At the opposite end of the hierarchy, off-chip memory controllers interface the SoC with external DRAM devices, defining the effective capacity and bandwidth available to the system. For industrial mobile-class SoCs, LPDDR memories have become the de-facto standard thanks to their high data rate, moderate energy per bit. Unlike simpler serial interfaces such as HyperRAM, commonly adopted in academic CPS platforms [1, 53], LPDDR integration entails the use of mixed-signal PHYs, precise timing calibration, and protocol-compliant initialization sequences, significantly increasing integration complexity and cost but enabling substantially higher sustained bandwidth [29, 100].

This chapter addresses this challenge through the integration and characterization of an high-performance hybrid memory subsystem in a heterogeneous research RISC-V SoC platform. The subsystem combines a dual-port, software-managed L2 scratchpad—serving as the shared on-chip memory across computational domains—with an industry-grade LPDDR controller that provides access to high-capacity off-chip DRAM. Together, these components define the hierarchical memory structure of the platform and govern how host cores, accelerators, and security domains exchange data. The L1 SPM described in the template architecture is not analyzed here, as transfers between L2 and L1 memories are latency-hidden through double buffering, effectively decoupling computation and

communication as discussed in Section 5.3.4.

The contributions of this chapter are twofold:

- **Memory hierarchy design for software-defined acceleration:** architectural integration of a scratchpad-centric, software-managed memory hierarchy with a high-bandwidth LPDDR off-chip memory system, preserving explicit data movement and predictable access patterns.
- **Integration-driven characterization:** Evaluation of latency and bandwidth at the L2 scratchpad–LPDDR boundary, analyzing the impact of transfer granularity, buffering, and arbitration on sustained throughput.

6.2 Background

Modern heterogeneous SoCs rely on a combination of on-chip and off-chip memory technologies to balance performance, predictability, . In the context of software-defined acceleration, two classes of memories play a central role: SPMs, which provide deterministic and explicitly controlled on-chip storage, and LPDDR DRAM, which offers high-capacity and high-bandwidth off-chip memory. Understanding the complementary characteristics and inherent tensions between these two memory technologies is essential for the design of efficient memory hierarchies in heterogeneous RISC-V SoCs.

6.2.1 Scratchpad Memories

Scratchpad memories are on-chip SRAM blocks that are explicitly managed by software or by dedicated DMA engines, rather than by hardware-managed cache policies. Unlike caches, SPMs do not

perform tag lookup, coherence management, or dynamic replacement, resulting in lower access latency, reduced area and energy overhead, and fully predictable timing behavior. These properties make SPMs particularly attractive for real-time systems, accelerators, and software-defined execution models, where data placement and movement are orchestrated explicitly.

In heterogeneous SoCs, SPMs are commonly organized as shared L2 memories accessed by multiple computational domains, including host processors, DMA engines, and accelerators. Their explicit management enables software to implement data tiling, double buffering, and controlled reuse, which are key techniques for hiding memory latency and sustaining throughput in irregular workloads. However, the lack of hardware-managed locality places a greater burden on software and requires careful architectural integration when interfacing with off-chip memory systems.

6.2.2 LPDDR Memory Systems

Low-Power Double Data Rate (LPDDR) dynamic random-access memory is the dominant off-chip memory technology in mobile and embedded SoCs, balancing bandwidth, capacity, and energy efficiency. Since its first generation (LPDDR1) in the early 2000s, successive iterations up to LPDDR5 have progressively increased interface data rates—from hundreds of megatransfers per second (MT/s) to over 6.4 GT/s—while reducing operating voltage and standby power. LPDDR devices are widely adopted in energy-constrained systems such as smartphones, automotive controllers, and emerging heterogeneous RISC-V SoCs due to their compact package-on-package (PoP) integration and optimized I/O signaling.

Architecture Overview

An LPDDR device is organized hierarchically to exploit parallelism and improve bandwidth utilization. At the top level, the memory is divided into channels, each containing one or more ranks, which in turn are composed of multiple banks subdivided into subarrays of DRAM cells. This organization enables concurrent operations across independent structures, effectively overlapping activation, read/write, and refresh commands. The main structural elements are:

- **Channel**: A logically independent 16- or 32-bit data interface with its own command/address (CA) bus. Multi-channel configurations (e.g., 2 × 16-bit) allow concurrent access and higher aggregate bandwidth.
- **Rank**: A rank groups multiple DRAM dies that share command and control signals but provide independent data interfaces.
- **Banks and Subarrays**: Each bank consists of multiple subarrays of DRAM cells organized in rows and columns. A row buffer temporarily stores a full DRAM row after activation, enabling subsequent read/write operations at low latency as long as the same row remains open.

From a system-integration standpoint, an LPDDR subsystem relies on two tightly coupled hardware components: the LPDDR controller (CTRL) and the physical layer (PHY). The CTRL implements the transaction protocol, command scheduling, address mapping, refresh management, and power-state transitions, while exposing a standard SoC-facing interface (typically AXI) that manages

requests and Quality-of-Service (QoS) arbitration among masters. The PHY handles the low-level electrical interface to the DRAM devices, performing data serialization/deserialization, signal timing alignment, and calibration procedures such as DQS delay tuning and impedance matching. Communication between these components typically adheres to the DDR PHY Interface (DFI) standard. Owing to its mixed-signal nature, the PHY is highly sensitive to process, voltage, and temperature (PVT) variations, complicating portability and open-source integration.

Protocol and Operation

LPDDR devices operate in accordance with the JEDEC JESD209 family of standards [101], which define a command-based protocol managed by a dedicated memory controller. All transactions are issued over the command/address (CA) bus and are governed by strict timing constraints, including the row-to-column delay (tRCD), precharge time (tRP), row active time (tRAS), and refresh interval (tREFI).

The protocol distinguishes several command classes:

- **ACTIVATE (ACT):** Opens a specific row within a selected bank.
- **READ (RD):** Initiates a read transaction from the active row buffer.
- **WRITE (WR):** Transfers data from the controller to the active row buffer.
- **PRECHARGE (PRE):** Closes the active row.

- **REFRESH (REF):** Periodically restores charge in DRAM cells.

LPDDR employs double data rate (DDR) signaling and fixed-length burst transfers, supporting bank interleaving and multi-bank activation to hide activation and precharge latencies. While this design enables high sustained bandwidth, it also favors large, contiguous transfers and introduces protocol overhead that must be amortized through careful buffering and scheduling.

6.3 Memory Subsystem Integration in an Open RISC-V SoC

Figure 6.1 illustrates the memory subsystem integrated into the template SoC architecture introduced in Section 2.2, consisting of a dual-port L2 scratchpad memory [1] and an industry-grade LPDDR4 memory controller.

6.3.1 L2 SPM

The L2 SPM supports multiple independent request ports with two distinct data widths, referred to as narrow and wide. The narrow ports, typically 32 or 64 bits wide—matching the data width of processor cores—are optimized for low-latency access. Conversely, the wide ports, ranging from 256 up to 1024 bits, are designed for high-throughput transfers and can tolerate higher access latency. All requestors share a common memory address space; however, thanks to the interleaved bank organization of the memory island, access contention is minimized, allowing multiple ports to operate

[1] https://github.com/pulp-platform/memory_island

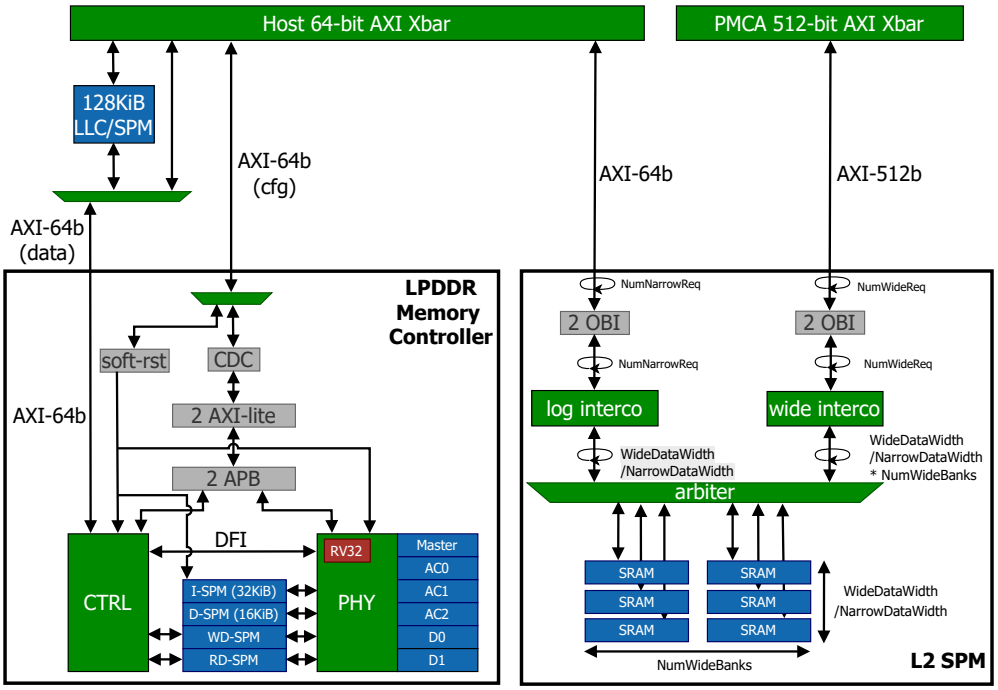


Figure 6.1. Memory Subsystem composed of LPDDR memory controller and L2 SPM.

concurrently. In the adopted configuration, the narrow port is 64 bits wide and interfaces with the host-domain crossbar, whereas the wide port is configured to 512 bits to match the data width of the PMCA. The AXI transactions are first converted into the Open Bus Interface (OBI) protocol and then routed through two dedicated interconnects: a logarithmic interconnect [102] for the narrow data port and a wide interconnect for the wide data port. Each interconnect distributes incoming requests across the instantiated memory banks, effectively partitioning the traffic according to the available bank parallelism. The design is parametric, allowing several architectural parameters to be tuned according to system requirements. The main configurable parameters are the number of

wide memory banks ($N_{\text{WideBanks}}$), the number of wide request ports (N_{WideReq}), the data width of the wide interface ($WideDataWidth$), the number of narrow request ports ($N_{\text{NarrowReq}}$), and the data width of the narrow interface ($NarrowDataWidth$). These parameters collectively define the degree of memory banking and port parallelism. The total number of instantiated SRAM blocks is determined by the relation:

$$N_{\text{SRAMs}} = N_{\text{WideBanks}} \times \left(\frac{WideDataWidth}{NarrowDataWidth} \right)$$

which reflects the subdivision of each wide bank into multiple narrower sub-banks to support concurrent accesses from different requestors. Before accessing the memory banks for read or write operations, the L2 SPM includes an arbitration crossbar with a configurable stall mechanism, which can temporarily delay wide-port requests to prioritize narrow-port accesses in case of contention.

6.3.2 LPDDR4 Memory Controller

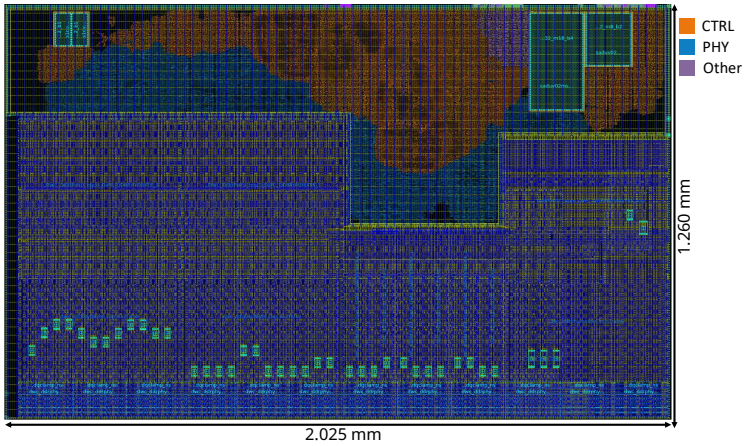
To interface with the external memory device, an industry-grade LPDDR4 memory controller is integrated into the system, preceded by a 128 KiB LLC that buffers transactions between the on-chip interconnect and the DRAM interface. Access to the LPDDR subsystem occurs through a dedicated AXI4 configuration port, which is demultiplexed to either write the soft-reset register—used to synchronously reset the entire LPDDR subsystem within a single clock cycle—or to traverse a conversion chain that sequentially converts AXI4 to AXI4-Lite and then to the APB protocol. The APB interface is employed to configure both the CTRL and the PHY, load the training firmware, and execute the initialization and

calibration procedures required to establish proper timing alignment and impedance matching between the controller and the LPDDR memory devices. At the top level of the LPDDR subsystem, a second 64-bit AXI4 data port is instantiated and connected to the LLC, enabling the main data transfers once link training and initialization are completed. This port is directly connected to the CTRL's AXI4 interface, which manages the translation from the AXI4 protocol to the DFI (DDR PHY Interface) protocol used to communicate with the PHY. The PHY itself consists of two main parts: a soft component and a modular macro component. The soft part includes a lightweight RV32 processor that executes the training and calibration firmware, which is stored in a dedicated instruction scratchpad memory (I-SPM) and loaded during the boot phase. This firmware handles the initialization routines, write leveling, read DQS alignment, and impedance calibration necessary to establish a stable link between the SoC and the LPDDR device. The macro part is composed of multiple modular building blocks—namely, the Master macro, Address/Command (AC) macros, and Data (D) macros. The Master macro manages the global timing generation, initialization sequencing, and interface control with the controller. The AC macros distribute the reference clocks, command, and address signals across the memory channels. The D macros implement the data-path circuitry, including SerDes, DQS gating, and on-die termination logic for the DQ/DQS lines. Depending on the memory configuration, multiple instances of these macros can be instantiated, supporting up to six AC units and four D per Master macro to scale bandwidth and data-bus width according to the target LPDDR device. In the implemented configuration, a single Master macro is instantiated together with three AC and two D macros, resulting in a single-channel LPDDR4 interface with a 64-bit data bus. At the

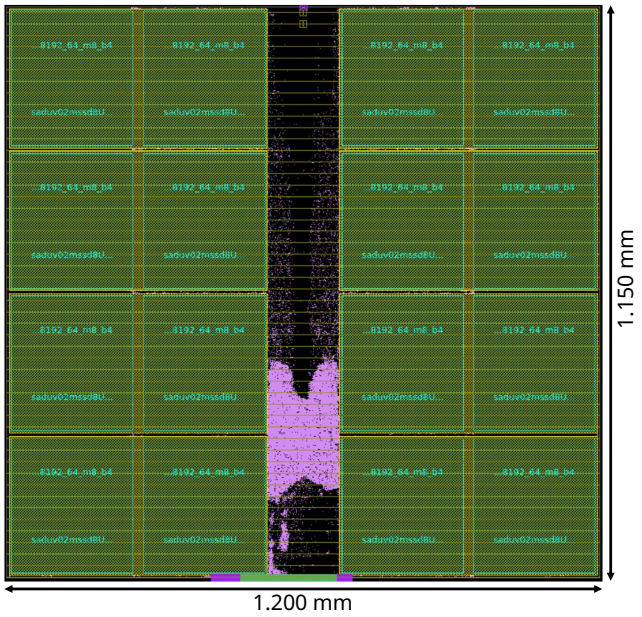
nominal LPDDR4 data rate of 3.2 Gb/s per pin. Within the memory controller, two dedicated SRAMs manage write data buffering and retry handling. The wdataRAM acts as a staging buffer, decoupling the AXI domain from the DRAM interface to sustain throughput during stalls and enabling efficient command scheduling through burst reordering and QoS arbitration.

6.4 Physical Design & Results

To implement the memory subsystem, we used Questa 2024.1 for RTL development, verification, and system integration, and Synopsys Fusion Compiler 2025.2 for the physical implementation in GlobalFoundries 22FDX technology. Figure 6.2a shows the placed-and-routed layout of the LPDDR subsystem. The CTRL cells are highlighted in orange, while the PHY cells—including the AC, D, and Master macros located at the bottom of the floorplan—are highlighted in blue. The memory macros located at the top-left and top-right of the layout correspond respectively to the I/D-SPMs and the wdataRAM and retryRAM blocks within the memory controller. The LPDDR subsystem has been implemented within a 2.025×1.260 mm^2 area, operating with a reference DFI clock of 600 MHz and an SoC clock of 500 MHz. Figure 6.2b shows the placed-and-routed layout of the L2 SPM, implemented in a 1.150×1.200 mm^2 area and operating at a clock frequency of 500 MHz. Figure 6.3 shows latency and bandwidth utilization at the LPDDR–L2 SPM interface as a function of transfer size. Bars on the left y-axis report read latency in nanoseconds, while the curves on the right y-axis show bandwidth utilization for reads (blue) and writes (magenta), defined as the achieved throughput normalized to the theoretical



(a) LPDDR4 subsystem.



(b) L2 SPM subsystem.

Figure 6.2. Placed-and-routed layouts of the LPDDR4 and L2 SPM subsystems implemented in GF22FDX technology.

peak of the LPDDR channel. The x-axis is $\log_2$-scaled across transfer sizes from 16 B to 256 KiB. Measurements are taken between the L2 SPM and the LPDDR subsystem, after SPM buffering and controller

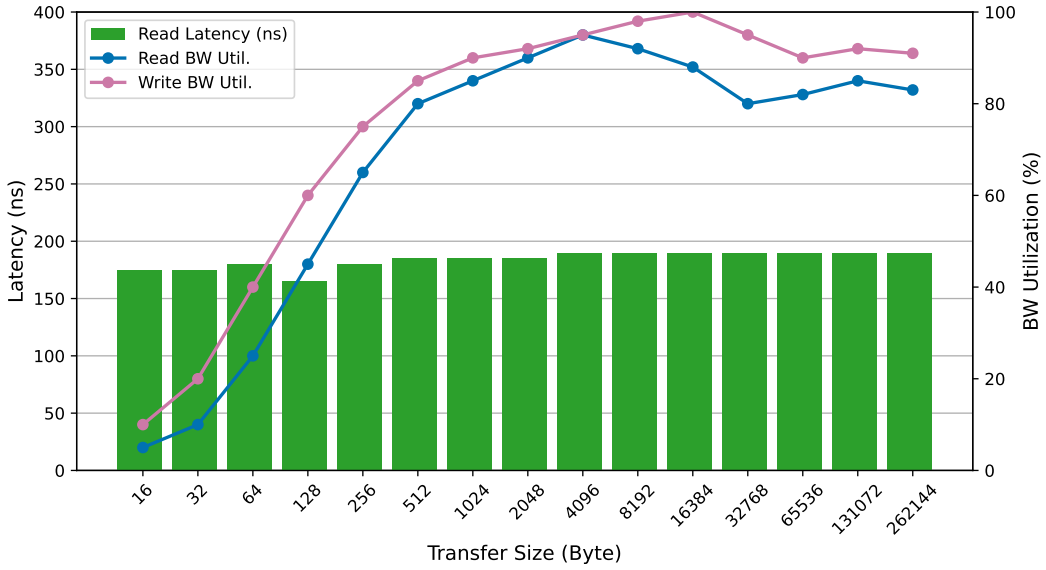


Figure 6.3. Latency and bandwidth utilization at the LLC–LPDDR interface measured with a one-dimensional streaming benchmark as a function of transfer size.

arbitration but before PHY signaling, thereby characterizing the effective performance of the external memory interface as observed by SoC initiators such as DMA engine or PMCA. The results were obtained using a synthetic one-dimensional streaming benchmark that performs contiguous reads and writes between the L2 SPM and the LPDDR subsystem. This configuration isolates the intrinsic transfer efficiency of the memory subsystem, providing a fair and reproducible evaluation of latency and sustained bandwidth under ideal sequential access conditions. All tests were triggered from the CVA6 host core, which orchestrated the data transfers to and from the LPDDR controller through the standard AXI interface. The benchmark was executed after PHY initialization, performed via a dedicated firmware developed to configure the controller registers, execute link training, and calibrate the PHY. The complete

setup was run in cycle-accurate simulation, using vendor-provided behavioral models for the LPDDR hard macros and a Virtual IP (VIP) to emulate the external memory device, ensuring accurate timing and protocol behavior consistent with silicon implementation. The figure shows that increasing the transfer size progressively amortizes the fixed protocol overhead associated with DRAM activation, precharge, and command turnaround. The read latency remains nearly constant, between 165 and 190 ns, reflecting the intrinsic timing of the LPDDR device and controller arbitration delays. Conversely, bandwidth utilization improves sharply with larger transfers: writes reach near-peak efficiency earlier, while reads saturate slightly later due to turnaround penalties and refresh cycles. Beyond a few kilobytes, both curves converge toward the controller's sustainable throughput, with only a minor decrease at the largest sizes caused by queue depth limitations and refresh overhead. Overall, the one-dimensional transfer benchmark highlights the steady-state behavior of the LPDDR–L2 SPM data path, demonstrating that the implemented subsystem achieves stable latency and high bandwidth utilization for burst sizes representative of DMA and PMCA workloads.

The designed subsystem has been integrated into a heterogeneous SoC, shown in Fig. 6.4. The SoC includes a host-domain processor based on CVA6, similar to the one described in Chapter 3, but without CFI capabilities. Flamingo targets mixed-criticality applications such as robotics, automotive, and industrial control. It integrates a Linux-capable 64-bit RISC-V host processor with virtualization support and hardware mechanisms for time predictability. AI-intensive workloads are accelerated by a compute cluster composed of two multi-precision floating-point RISC-V vector engines, achieving up to 64 GFLOPS at FP8 precision. The LPDDR interface is located

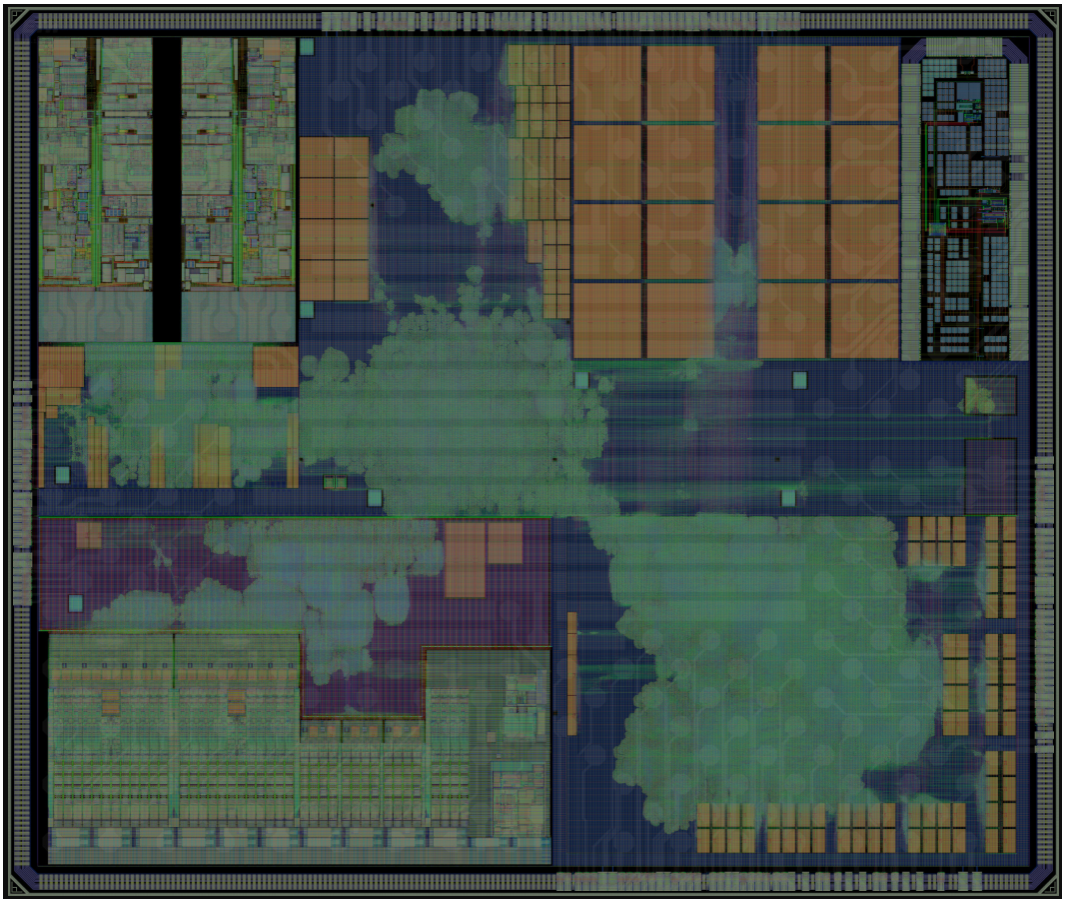


Figure 6.4. Flamingo SoC.

in the bottom-left region of the chip, while the L2 SPM domain occupies the upper-right area.

6.5 Related Work

Low-power heterogeneous SoCs increasingly require off-chip DRAM–equipped memory hierarchies to support Linux-class software stacks and data-intensive workloads, but the choice of memory technology and interface architecture has major implications for integration ef-

fort, achievable bandwidth, energy efficiency, and system scalability. Several academic works explore a wide range of design points that trade integration complexity, bandwidth, and architectural flexibility through different off-chip memory technologies and hierarchy organizations.

6.5.1 Low-Cost Fully Digital DRAM Integration

Several open and research-oriented platforms prioritize reduced integration complexity by adopting fully digital off-chip memory interfaces. HULK-V, for instance, targets Linux-capable ultra-low-power computing by integrating HyperRAM as its external memory technology [29]. This choice avoids the mixed-signal complexity of LPDDR PHYs and enables relatively simple and portable integration, while exposing up to 512 MB of external memory to the host processor. The platform exemplifies a design point where off-chip memory primarily serves to extend capacity beyond on-chip SRAM, with achievable performance constrained by the characteristics of the low-pin-count DRAM interface and its relatively limited bandwidth.

Within the same class of fully digital memory systems, Cheshire explores a higher-throughput design point by adopting a reduced pin-count (RPC) DRAM interface. While still avoiding mixed-signal PHY integration, Cheshire reports an achievable off-chip bandwidth of up to 750 MB/s at a 200 MHz interface frequency, enabled by a wider and more aggressively clocked digital interface. The reported PHY footprint is approximately 3.5 kGE, with an interface energy of about 250 pJ/B, illustrating a deliberate trade-off between increased interface complexity and substantially higher sustained memory throughput compared to simpler digital DRAM solutions. Together, HULK-V and Cheshire illustrate how fully

digital DRAM interfaces span a spectrum of architectural trade-offs between integration effort and achievable memory performance in open heterogeneous SoCs.

6.5.2 Off-Chip Memory Integration via FPGA Platforms

A third class of off-chip connectivity exploits FPGA-mediated DRAM access, where external memory controllers are hosted on an FPGA and connected to the SoC through wide source-synchronous links. In this class of systems, off-chip DRAM is integrated as a distributed resource tightly coupled to the on-chip interconnect rather than as a single centralized endpoint. EPOCHS-1 exemplifies this approach by adopting a tile-based memory hierarchy in which multiple LLC tiles each integrate a channel to external DRAM accessed through an FPGA-hosted DDR controller. This organization enables off-chip bandwidth to scale with the number of memory tiles and distributes traffic across the network-on-chip. Due to the high latency of FPGA-mediated DRAM access, EPOCHS complements the coherent cache hierarchy with large, software-managed scratchpad tiles primarily used to expand effective on-chip storage for OS-supported workloads, reflecting a capacity- and scalability-driven design point rather than a focus on peak off-chip throughput.

A similar FPGA-mediated DRAM connectivity approach is also adopted in several research SoCs designed within the Chipyard ecosystem [48]. In particular, [103, 104] rely on external DRAM accessed through FPGA-hosted DDR controllers connected via high-speed chip-to-FPGA interfaces. In these systems, the FPGA provides the DRAM PHY and controller functionality, while the chip integrates on-chip caches and local SRAMs to tolerate the high latency of the off-chip path. As in EPOCHS, this organization

prioritizes system bring-up flexibility, large memory capacity, and support for complex software stacks over minimizing off-chip access latency, placing these designs in the same capacity- and scalability-driven class of FPGA-mediated off-chip memory architectures. Such FPGA-mediated memory solutions are practical mainly for test-chip and prototyping platforms, as they rely on high-end FPGAs for DRAM access, whereas deployable systems require the significantly higher cost and engineering effort of integrating a PHY-based off-chip memory hierarchy directly on silicon.

6.5.3 PHY-Based Off-Chip Memory Integration in Open-Source SoCs

A final class of research SoCs demonstrates fully integrated off chip memory hierarchies based on real DRAM controllers and PHYs in silicon, targeting high performance and bandwidth while retaining architectural openness on the main compute architecture. Occamy [105] represents a SoA example of this approach, implementing a large scale RISC-V manycore system with directly integrated HBM2E memory stacks through on chip controllers and PHYs in a 2.5D chiplet assembly. Unlike FPGA mediated designs, Occamy treats off chip memory as a first class architectural component, tightly coupling multiple HBM channels to the on chip interconnect to provide high sustained bandwidth and predictable access characteristics. While PHY based memory controllers are the de-facto standard in industrial high-end products, their complexity has traditionally limited their adoption in academic designs, making Occamy one of the first academic SoCs to integrate such a memory subsystem. Table 6.1 compares the proposed memory subsystem, integrated within the Flamingo SoC with those of other SoA academic SoCs.

Table 6.1. Comparison of off-chip memory connectivity in open and academic SoCs.

Platform	Tech.	Domain	Off-chip	Cap.	Ch./W	PHY	Mem. Ctrl. Area	BW	On-chip buffering
This work	22 nm	Edge	LPDDR4	$\leq$ 8 GB	1×64 b	Mixed	2.55 mm^2	25.6 GB/s	LLC + L2 SPM
HULK-V [29]	22 nm	IoT	HyperRAM	$\leq$ 512 MB	1×8 b	Digital	0.27 mm^2	n.s.	LLC + L2 SPM
Cheshire [56]	65 nm	IoT	RPC DRAM	n.s.	1×32 b	Digital	0.42 mm^2	750 MB/s	LLC
EPOCHS-1 [30]	12 nm	Edge	DDR (FPGA)	n.s.	4 × n.s.	n.a.	n.s.	n.s.	LLC tiles + SPM
Chipyard (Vec.) [103]	16 nm	Edge	DDR (FPGA)	n.s.	1 × n.s.	n.a.	n.s.	n.s.	Caches + SRAM
Chipyard (Het.) [104]	22 nm	Edge	DDR (FPGA)	n.s.	1 × n.s.	n.a.	n.s.	n.s.	Caches + SRAM
Occamy [105]	12 nm	HPC	HBM2E	$\leq$ 16 GB	8×128 b	Mixed	18.3 mm^2	512 GB/s	Hierarchical SPMs

6.6 Summary

This chapter presented the design, integration, and analysis of the memory subsystem within the heterogeneous RISC-V SoC platform, completing the architectural exploration on modern embedded mobile CPS platforms initiated in the previous chapters. The subsystem combines a dual-port L2 scratchpad memory with an industry-grade LPDDR4 controller, bridging the high-bandwidth off-chip memory domain with the low-latency on-chip computational domains. The L2 SPM provides deterministic and software-managed access across heterogeneous requestors, supporting both latency-sensitive host traffic and wide high-throughput accelerator transactions. Its parametric design allows flexible scaling of data width and banking to match system requirements while maintaining predictable timing behavior. On the other hand, the LPDDR4 controller and PHY integration demonstrate how complex, mixed-signal interfaces can be effectively incorporated into an open RISC-V ecosystem. Cycle-accurate simulations, driven by the CVA6 host processor and validated through vendor models and a VIP memory device, confirmed the correct operation of the integrated memory subsystem and characterized its performance. The results showed stable read latency and near-peak bandwidth utilization for transfer sizes in the kilobyte range, indicating that the implemented architecture sustains high throughput under realistic DMA and PMCA workloads.

Overall, this chapter provides both methodological and quantitative insights into the integration of on-chip and off-chip memory hierarchies in heterogeneous SoCs.

Chapter 7
Conclusions and Future Directions

This thesis presented the design and evaluation of secure and efficient heterogeneous RISC-V architectures for next-generation CPS, addressing the combined challenges of runtime security, energy-efficient AI acceleration, and the integration of industry-grade memory subsystems. The proposed SoC template integrates three tightly coupled domains: a secure host processor, a programmable accelerator cluster, and a hierarchical LPDDR-based memory subsystem, providing a unified foundation for trustworthy and high-performance embedded computing.

The first contribution focused on CFI as a hardware-assisted defense against runtime attacks. Two alternative approaches were explored: (i) CVA6-CFI, the first hardware-extended processor implementing the RISC-V `Zicfiss` and `Zicfilp` standards for fine-grained control-flow protection; and (ii) TitanCFI, an hybrid hardware/software-based CFI mechanism that leverages the OpenTitan RoT as a co-processor for runtime verification. These represent alternative design points. Pipeline-level enforcement offers strong protection with minimal performance and hardware overhead,

but it requires toolchain modifications and software recompilation. RoT-based enforcement retains low hardware cost and requires no software recompilation or toolchain modifications. However, it incurs non-negligible runtime overhead on a subset of Embench IoT benchmarks.

The second major contribution, demonstrated that sparse and brain-inspired workloads such as SNNs can be efficiently accelerated on programmable RISC-V cores using lightweight streaming ISA extensions for indirect memory access. This software-defined framework achieved up to 7.3× speedup and 5.7× energy savings compared to SIMD baselines, matching or surpassing specialized neuromorphic processors while retaining full programmability and avoiding overspecialization.

Finally, the integration of an industry-grade LPDDR4 controller and an L2 SPM within the template heterogeneous RISC-V SoC demonstrated a complete RTL-to-physical design flow, bridging industry-grade memory IP with an open academic platform. Implemented in GlobalFoundries 22FDX technology, the simulated subsystem demonstrated high sustained bandwidth and stable latency across transfers between the L2 scratchpad and the external DRAM VIP.

Overall, this work demonstrates that open RISC-V architectures can effectively balance security, flexibility, and efficiency for emerging CPS and emerging AI workloads. By bridging secure system design, programmable acceleration of emerging AI workloads, and high-performance memory integration, this thesis lays the groundwork for trustworthy and energy-aware RISC-V platforms capable of supporting the next generation of intelligent, autonomous CPS. Future work will focus on the end-to-end deployment of SNN workloads using the optimized library introduced in Chapter 5, integrated into

the Linux software stack as a user-space runtime, and on consolidating the architectural domains introduced in Chapter 2 into a single, silicon implementation of the proposed heterogeneous CPS SoC template—realizing in hardware the secure, efficient, and flexible platform envisioned throughout this thesis.

7.1 Lessons Learned and Broader Implications

Beyond the individual technical contributions, this doctoral work enabled a set of cross-cutting insights on the design of secure, efficient, and programmable heterogeneous SoCs for cyber-physical systems.

A first key lesson concerns the trade-off between security enforcement strength and system integration cost. While tightly integrated, pipeline-level security mechanisms such as CVA6-CFI provide strong protection with limited runtime overhead, they introduce non-negligible complexity in the toolchain and software ecosystem. Conversely, Root-of-Trust–based enforcement shifts complexity away from the processor microarchitecture and preserves software compatibility, at the expense of higher runtime overhead in specific workloads. This highlights that, in practical CPS platforms, deployability and ecosystem compatibility can be as decisive as raw performance or area efficiency when selecting security mechanisms.

A second insight relates to programmability versus specialization in AI acceleration. The results obtained with SpikeStream demonstrate that, for emerging workloads such as SNNs, carefully co-designed hardware–software abstractions can achieve efficiency comparable to specialized accelerators while retaining flexibility and long-term adaptability. This challenges the assumption that

energy-efficient AI at the edge necessarily requires fixed-function hardware, and suggests that software-defined acceleration, when paired with lightweight ISA extensions and explicit data orchestration, represents a sustainable design point in rapidly evolving AI domains.

Memory-system design emerged as a third decisive dimension. Integrating an industry-grade LPDDR memory controller within an open, scratchpad-centric SoC exposed the significant area and integration costs required to sustain high off-chip bandwidth, especially when compared to simpler interfaces such as HyperRAM, RPC-based DRAM of FPGA based links. This experience reinforced the view that memory hierarchies are architectural choices defined by explicit trade-offs between area, bandwidth, and integration effort, rather than being purely performance-driven components.

Across all contributions, a recurring theme is the importance of architectural templates over isolated optimizations. Rather than optimizing individual components in isolation, this thesis shows that robust CPS platforms benefit from co-design across processors, accelerators, memory systems, and security primitives. In this sense, the proposed heterogeneous RISC-V template is not a fixed design, but a structured framework within which different trade-offs can be explored and instantiated depending on application requirements.

Bibliography

1. Valente, L. *et al.* A Heterogeneous RISC-V Based SoC for Secure Nano-UAV Navigation. *IEEE Transactions on Circuits and Systems I: Regular Papers* **71,** 2266–2279 (2024).
2. Olowononi, F., Rawat, D. B. & Liu, C. Resilient Machine Learning for Networked Cyber Physical Systems: A Survey for Machine Learning Security to Securing Machine Learning for CPS. *arXiv preprint arXiv:2102.07244* (2021).
3. Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S. & Kepner, J. *AI and ML Accelerator Survey and Trends* in *2022 IEEE High Performance Extreme Computing Conference (HPEC)* (2022), 1–10.
4. Giri, D., Chiu, K.-L., Di Guglielmo, G., Mantovani, P. & Carloni, L. P. *ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning* in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2020), 1049–1054.
5. Han, S., Mao, H. & Dally, W. J. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding* in *International Conference on Learning Representations (ICLR)* (2016).

6. Li, H., Kadav, A., Durdanovic, I., Samet, H. & Graf, H. P. *Pruning Filters for Efficient ConvNets* in *International Conference on Learning Representations (ICLR)* (2017).

7. Jacob, B. *et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2018).

8. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R. & Bengio, Y. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv preprint arXiv:1602.02830* (2016).

9. Elsen, E., Dukhan, M., Gale, T. & Simonyan, K. *Fast Sparse ConvNets* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020).

10. Narang, S., Undersander, E. & Diamos, G. *Block-Sparse Recurrent Neural Networks* in *International Conference on Learning Representations (ICLR)* (2017).

11. Gou, S. *et al.* Dynamic spatio-temporal pruning for efficient spiking neural networks. *Frontiers in Neuroscience* **14,** 1545583 (2025).

12. Zanatta, L., Di Mauro, A., Barchi, F., Bartolini, A., Benini, L. & Acquaviva, A. Directly-trained Spiking Neural Networks for Deep Reinforcement Learning: Energy efficient implementation of event-based obstacle avoidance on a neuromorphic accelerator. *Neurocomputing* **562,** 126885 (2023).

13. Zanatta, L., Barchi, F., Manoni, S., Tolu, S., Bartolini, A. & Acquaviva, A. Exploring spiking neural networks for deep

reinforcement learning in robotic tasks. *Scientific Reports* **14** (2024).

14. Di Mauro, A., Prasad, A. S., Huang, Z., Spallanzani, M., Conti, F. & Benini, L. *Sne: an energy-proportional digital accelerator for sparse event-based convolutions* in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2022), 825–830.

15. Yang, Z. *et al.* Back to Homogeneous Computing: A Tightly-Coupled Neuromorphic Processor With Neuromorphic ISA. *IEEE Transactions on Parallel and Distributed Systems* **34,** 2910–2927 (2023).

16. Frenkel, C., Lefebvre, M., Legat, J.-D. & Bol, D. A 0.086-mm^2 12.7-pJ/SOP 64k-Synapse 256-Neuron Online-Learning Digital Spiking Neuromorphic Processor in 28-nm CMOS. *IEEE Transactions on Biomedical Circuits and Systems* **13,** 145–158 (2019).

17. Roy, K., Sinanoglu, O. & Karri, R. Cybersecurity in Cyber-Physical Systems: A Challenge for Autonomy. *Proceedings of the IEEE* **104,** 1039–1051 (2016).

18. Xu, Y., Han, S., Liu, Y. & Jantsch, A. Secure and Trustworthy Cyber–Physical Systems: A Review of the State of the Art. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* **41,** 5373–5386 (2022).

19. Humayed, A. *et al.* Cyber-Physical Systems Security–A Survey. *IEEE Internet of Things Journal* (2017).

20. Abadi, M., Budiu, M., Erlingsson, Ú. & Ligatti, J. *Control-Flow Integrity: Principles, Implementations, and Applications*

in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)* (2005), 340–353.

21. Burow, N., Carr, S. A., Nash, J., Payer, M., Larsen, P. & Franz, M. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)* **50,** 1–33 (2017).

22. Zhang, Z., McCamant, S., Regehr, J. & Carbin, M. *HCFI: Hardware-Enforced Control-Flow Integrity* in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), 526–538.

23. Intel Corporation. Intel Control-Flow Enforcement Technology (CET): Mitigating ROP and JOP Attacks. *Intel White Paper* (2020).

24. Oliveira, B., Martins, J., Cunha, T., Pinto, S. & Tavares, A. *Practical Experiences with Pointer Authentication on ARMv8.3-A* in *Proceedings of the IEEE International Conference on Computer Design (ICCD)* (2020).

25. Moreira, J. E., Chatterjee, D., Ekanadham, K. & Flores, A. *Return-oriented programming protection in the IBM POWER10* in *Proceedings of the 19th ACM International Conference on Computing Frontiers* (Association for Computing Machinery, Turin, Italy, 2022), 173–176.

26. Ciani, M. *et al.* Unleashing OpenTitan's Potential: a Silicon-Ready Embedded Secure Element for Root of Trust and Cryptographic Offloading. *ACM Trans. Embed. Comput. Syst.* **24** (2025).

27. Prasanna, R., Banerjee, A. & Chattopadhyay, A. A Survey on Hardware Roots of Trust for Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)* **20,** 1–26 (2021).

28. LowRISC Contributors. *OpenTitan: Open Source Silicon Root of Trust* in *OpenTitan Project Documentation* (2020).

29. Valente, L. *et al. HULK-V: a Heterogeneous Ultra-low-power Linux capable RISC-V SoC* in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2023), 1–6.

30. Zuckerman, J. *et al.* EPOCHS-1: A 12 nm Highly Heterogeneous Open-Source SoC With Distributed Coin-Based Power Management and Integrated Hybrid Voltage Regulation. *IEEE Journal of Solid-State Circuits,* 1–19 (2025).

31. Manoni, S., Parisi, E., Tedeschi, R., Rossi, D., Acquaviva, A. & Bartolini, A. CVA6-CFI: A First Glance at RISC-V Control-Flow Integrity Extensions. *arXiv preprint arXiv:2602.04991* (2026).

32. Parisi, E. *et al. TitanCFI: Toward Enforcing Control-Flow Integrity in the Root -of- Trust* in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2024), 1–6.

33. Manoni, S., Scheffler, P., Zanatta, L., Acquaviva, A., Benini, L. & Bartolini, A. *SpikeStream: Accelerating Spiking Neural Network Inference on RISC-V Clusters with Sparse Computation Extensions* in *2025 Design, Automation & Test in Europe Conference (DATE)* (2025), 1–7.

34. Manoni, S. *et al. NARS: Neuromorphic Acceleration through Register-Streaming Extensions on RISC-V Cores* in *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions* (Association for Computing Machinery, Ischia, Italy, 2024), 79–82.

35. Cota, E. G. *Scalable Emulation of Heterogeneous Systems* PhD thesis (2019), 186.

36. Jouppi, N. P. *et al. In-Datacenter Performance Analysis of a Tensor Processing Unit* in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), 1–12.

37. Patterson, D. A. & Hennessy, J. L. *Computer Architecture: A Quantitative Approach* 6th (Morgan Kaufmann, 2017).

38. Giri, D., Mantovani, P. & Carloni, L. P. Accelerators & Coherence: An SoC Perspective. *IEEE MICRO* **38,** 30–39 (2018).

39. Tortorella, Y., Bertaccini, L., Benini, L., Rossi, D. & Conti, F. RedMule: A mixed-precision matrix–matrix operation engine for flexible and energy-efficient on-chip linear algebra and TinyML training acceleration. *Future Generation Computer Systems* **149,** 122–135 (2023).

40. Jouppi, N. *et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings* in *Proceedings of the 50th annual international symposium on computer architecture* (2023), 1–14.

41. Belano, A., Tortorella, Y., Garofalo, A., Benini, L., Rossi, D. & Conti, F. A Flexible Template for Edge Generative AI With High-Accuracy Accelerated Softmax and GELU. *IEEE*

Journal on Emerging and Selected Topics in Circuits and Systems **15,** 200–216 (2025).

42. Lee, J.-J., Zhang, W. & Li, P. *Parallel Time Batching: Systolic-Array Acceleration of Sparse Spiking Neural Computation* in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022), 317–330.

43. Flamand, E. *et al. GAP-8: A RISC-V SoC for AI at the Edge of the IoT* in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2018), 1–4.

44. Zaruba, F., Schuiki, F., Hoefler, T. & Benini, L. Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads. *IEEE Transactions on Computers* **70,** 1845–1860 (2021).

45. Huerta, R., Shoushtary, M. A., Cruz, J.-L. & González, A. Analyzing Modern NVIDIA GPU cores. *arXiv preprint arXiv:2503.20481* (2025).

46. Kellogg, B., Bannon, P., *et al. Tesla Full Self-Driving (FSD) Computer and SoC* in *Hot Chips 31 Symposium (HCS)* (2019).

47. Mantovani, P. *et al. Agile SoC development with open ESP* in *Proceedings of the 39th International Conference on Computer-Aided Design* (Association for Computing Machinery, Virtual Event, USA, 2020).

48. Amid, A. *et al.* Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* **40,** 10–21 (2020).

49. Rossi, D. *et al. PULP: A parallel ultra low power platform for next generation IoT applications* in *2015 IEEE Hot Chips 27 Symposium (HCS)* (2015), 1–39.

50. Zhao, J., Agrawal, A., Nikolic, B. & Asanović, K. *Constellation: An Open-Source SoC-Capable NoC Generator* in *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)* (2022), 1–7.

51. Kim, S. *et al. MAVERIC: A 16nm 72 FPS, 10 mJ/Frame Heterogeneous Robotics SoC with 4 Cores and 13 INT8/FP32 Accelerators* in *2025 Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)* (2025), 1–3.

52. Moons, B., Uytterhoeven, R., Dehaene, W. & Verhelst, M. *14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI* in *2017 IEEE International Solid-State Circuits Conference (ISSCC)* (2017), 246–247.

53. Garofalo, A. *et al.* A Reliable, Time-Predictable Heterogeneous SoC for AI-Enhanced Mixed-Criticality Edge Applications. *arXiv preprint arXiv:2502.18953* (2025).

54. Schiavone, P. D. *et al. X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller* in *Proceedings of the 20th ACM International Conference on Computing Frontiers* (Association for Computing Machinery, Bologna, Italy, 2023), 379–380.

55. Tedeschi, R. *et al.* Culsans: An efficient snoop-based coherency unit for the cva6 open source risc-v application processor. *arXiv preprint arXiv:2407.19895* (2024).

56. Ottaviano, A., Benz, T., Scheffler, P. & Benini, L. Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In. *IEEE Transactions on Circuits and Systems II: Express Briefs* **70,** 3777–3781 (2023).

57. Conti, F., Palossi, D., Marongiu, A., Rossi, D. & Benini, L. *Enabling the heterogeneous accelerator model on ultra-low power microcontroller platforms* in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2016), 1201–1206.

58. Barbirotta, M., Angioli, M., Mastrandrea, A., Menichelli, F., Cheikh, A. & Olivieri, M. *Dual-Modular-Redundancy Voting Circuits for Single-Event-Transient Mitigation* in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2024), 1–6.

59. Barbirotta, M., Minervini, F., Morales, C. R., Cristal, A., Unsal, O. & Olivieri, M. Enhancing Fault Tolerance in High-Performance Computing: A Real Hardware Case Study on a RISC-V Vector Processing Unit. *IEEE Open Journal of the Computer Society* **5,** 553–565 (2024).

60. Pałka, M., Woodruff, J., Chisnall, D., Roe, M. & Moore, S. W. *Trends in Memory Unsafety* in *IEEE Symposium on Security and Privacy* (2024).

61. Li, Y., Liu, S., Liu, B., Huang, W., Wang, P. & Liu, L. *SoK: The Progress, Challenges, and Perspectives of Directed Grey-Box Fuzzing* in *USENIX Security Symposium* (2024).

62. Yang, S. *et al. SoK: All You Ever Wanted to Know about x86/x64 Binary Disassembly But Were Afraid to Ask* in *IEEE Symposium on Security and Privacy* (2021), 957–974.

63. Shacham, H. *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)* in *Proceedings of the ACM Conference on Computer and Communications Security* (2007), 552–561.

64. Kumar, R. *et al. Cats: Composable and agile tracking for security* in *IEEE Symposium on Security and Privacy (SP)* (2020), 819–836.

65. Spang, C., Lavan, Y., Hartmann, M., Meisel, F. & Koch, A. *DExIE - An IoT-Class Hardware Monitor for Real-Time Fine-Grained Control-Flow Integrity* in *Workshop on Design and Architectures for Signal and Image Processing (14th Edition)* (Association for Computing Machinery, Budapest, Hungary, 2021), 26–34.

66. Sullivan, D. *et al. Efficiently Enforcing Control-Flow Integrity with a Security Coprocessor* in *IEEE Secure Development Conference (SecDev)* (2016).

67. Liljestrand, H., Nyman, T., Wang, K., Perez, C. C., Ekberg, J.-E. & Asokan, N. *PAC it up: towards pointer integrity using ARM pointer authentication* in *Proceedings of the 28th USENIX Conference on Security Symposium* (USENIX Association, Santa Clara, CA, USA, 2019), 177–194.

68. Shanbhogue, V., Gupta, D. & Sahita, R. *Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity* in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Association for Computing Machinery, Phoenix, AZ, USA, 2019).

69. Yue, T. *et al.* Efficient Forward-Edge Control-Flow Integrity for COTS Binaries via Arm BTI. *IEEE Transactions on Information Forensics and Security* **20,** 7137–7152 (2025).

70. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T. & Brown, R. *MiBench: A free, commercially representative embedded benchmark suite* in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)* (2001), 3–14.

71. Wang, W., Feng, L., Shi, Z., Zhuo, C. & Chen, J. *Hardware-Assisted Control-Flow Integrity Enhancement for IoT Devices* in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2024), 1–6.

72. Clercq, R. D., Herrewege, A. V. & Verbauwhede, I. SOFIA: Software and control flow integrity architecture. *Computers & Security* **68,** 16–35 (2017).

73. Feng, L. *et al.* FastCFI: Real-Time Control-Flow Integrity Using FPGA Without Code Instrumentation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* **26,** 1–25 (2021).

74. Christoulakis, N., Christou, G., Ioannidis, S. & Papadogiannakis, A. *HCFI: Hardware-enforced control-flow integrity* in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY)* (ACM, 2016), 37–48.

75. De, A. *et al. FIXER: Flow Integrity Extensions for Embedded RISC-V* in *Design, Automation and Test in Europe (DATE)* (IEEE, 2019), 348–353.

76. Delshadtehrani, L. *et al. PHMon: A Programmable Hardware Monitor and Its Security Use Cases* in *29th USENIX* (2020).

77. Delshadtehrani, L. *et al.* Nile: A Programmable Monitoring Coprocessor. *IEEE Computer Architecture Letters* (2018).

78. Ciani, M. *et al. Cyber Security aboard Micro Aerial Vehicles: An OpenTitan-based Visual Communication Use Case* in *2023 IEEE ISCAS* (2023).

79. Zaruba, F. & Benini, L. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **27,** 2629–2640 (2019).

80. Davide Schiavone, P. *et al. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications* in *PATMOS* (2017).

81. De Clercq, R. *et al.* A survey of Hardware-based Control Flow Integrity (CFI). *CoRR* (2017).

82. Burow, N. *et al. SoK: Shining Light on Shadow Stacks* in *2013 IEEE S&P* (2019).

83. Zanatta, L., Barchi, F., Manoni, S., Tolu, S., Bartolini, A. & Acquaviva, A. Exploring Spiking Neural Networks for Deep Reinforcement Learning in Robotic Tasks. *Scientific Reports* **14,** 30648 (2024).

84. Lien, H.-H. & Chang, T.-S. Sparse Compressed Spiking Neural Network Accelerator for Object Detection. *IEEE Transactions on Circuits and Systems I: Regular Papers* **69,** 2060–2069 (2022).

85. Kim, S. & Yoo, H.-J. C-DNN V2: Complementary Deep-Neural-Network Processor With Full-Adder/OR-Based Reduction Tree and Reconfigurable Spatial Weight Reuse. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* **13,** 1026–1039 (2023).

86. Schuman, C. D., Kulkarni, S. R., Parsa, M., Mitchell, J. P., Date, P. & Kay, B. Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science* **2** (2022).

87. Moradi, S., Qiao, N., Stefanini, F. & Indiveri, G. A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems* **12,** 106–122 (2018).

88. Benjamin, B. V. *et al.* Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations. *Proceedings of the IEEE* **102,** 699–716 (2014).

89. Davies, M. *et al.* Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro* **38,** 82–99 (2018).

90. Wang, L. *et al.* LSMCore: A 69k-Synapse/mm2 Single-Core Digital Neuromorphic Processor for Liquid State Machine. *IEEE Transactions on Circuits and Systems I: Regular Papers* **69,** 1976–1989 (2022).

91. Ottati, F. *et al.* To Spike or Not to Spike: A Digital Hardware Perspective on Deep Learning Acceleration. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* **13,** 1015–1025 (2023).

92. Scheffler, P., Zaruba, F., Schuiki, F., Hoefler, T. & Benini, L. Sparse stream semantic registers: A lightweight ISA extension accelerating general sparse linear algebra. *IEEE Transactions on Parallel and Distributed Systems* (2023).

93. Domingos, J. M., Neves, N., Roma, N. & Tomás, P. *Unlimited Vector Extension with Data Streaming Support* in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), 209–222.

94. Schuiki, F., Zaruba, F., Hoefler, T. & Benini, L. Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores. *IEEE Transactions on Computers* **70,** 212–227 (2021).

95. Bellec, G. *et al.* A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature communications* **11,** 3625 (2020).

96. Xiao, M., Meng, Q., Zhang, Z., He, D. & Lin, Z. Online training through time for spiking neural networks. *Advances in neural information processing systems* **35,** 20717–20730 (2022).

97. Zhu, Y., Yu, Z., Fang, W., Xie, X., Huang, T. & Masquelier, T. *Training spiking neural networks with event-driven backpropagation* in *Proceedings of the 36th International Conference on Neural Information Processing Systems* (Curran Associates Inc., New Orleans, LA, USA, 2024).

98. Li, X., Jiang, J. & Gao, M. *SeaCache: Efficient and Adaptive Caching for Sparse Accelerators* in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture* (Association for Computing Machinery, 2025), 944–957.

99. Ganesan, V., Sen, S., Kumar, P., Gala, N., Veezhinathan, K. & Raghunathan, A. *Sparsity-Aware Caches to Accelerate Deep Neural Networks* in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2020), 85–90.

100. Feldmann, J., Lappas, J., Esmaeilpour, M., Abdo, H., Weis, C. & Wehn, N. Enhanced LPDDR4X PHY in 12 nm FinFET. *arXiv preprint arXiv:2503.11654* (2025).

101. JEDEC Solid State Technology Association. *Low Power Double Data Rate 4 (LPDDR4)* tech. rep. JESD209-4 (JEDEC Solid State Technology Association, 2014).

102. Rahimi, A., Loi, I., Kakoee, M. R. & Benini, L. *A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters* in *2011 Design, Automation & Test in Europe* (2011), 1–6.

103. Gonzalez, A. *et al. A 16mm2 106.1 GOPS/W Heterogeneous RISC-V Multi-Core Multi-Accelerator SoC in Low-Power 22nm FinFET* in *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference (ESSCIRC)* (2021), 259–262.

104. Schmidt, C. *et al. 4.3 An Eight-Core 1.44GHz RISC-V Vector Machine in 16nm FinFET* in *2021 IEEE International Solid-State Circuits Conference (ISSCC)* **64** (2021), 58–60.

105. Scheffler, P. *et al.* Occamy: A 432-core dual-chiplet dual-hbm2e 768-dp-gflop/s risc-v system for 8-to-64-bit dense and sparse computing in 12-nm finfet. *IEEE Journal of Solid-State Circuits* **60,** 1324–1338 (2025).