POLITECNICO DI TORINO
Repository ISTITUZIONALE

Ditching the Queue: Optimizing Coprocessor Utilization with Out-of-Order CPUs on compact Systems on Chip

Original
Ditching the Queue: Optimizing Coprocessor Utilization with Out-of-Order CPUs on compact Systems on Chip / Caon, M., Masera, G., Martina, M.. - In: ELECTRONICS. - ISSN 2079-9292. - ELETTRONICO. - 13:15(2024). [10.3390/electronics13153018]

Availability:
This version is available at: 11583/2991126 since: 2024-07-23T13:53:23Z

Publisher:
MDPI

Published
DOI:10.3390/electronics13153018

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

09 September 2026

electronics

MDPI

Article

Ditching the Queue: Optimizing Coprocessor Utilization with Out-of-Order CPUs on Compact Systems on Chip

Michele Caon, Guido Masera and Maurizio Martina *

Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy; michele.caon@polito.it (M.C.); guido.masera@polito.it (G.M.)
* Correspondence: maurizio.martina@polito.it

Abstract: The growing demand for high-performance and energy-efficient processing in edge-oriented Systems-on-Chip is driving the adoption of dedicated integrated circuits that accelerate computationally intensive workloads. To minimize area and performance overhead, low-power, general-purpose CPUs are often tightly coupled with domain-specific coprocessors implementing custom instructions, thereby delivering higher throughput and reduced memory traffic. However, commonly used in-order CPUs are not optimized for instruction-level parallelism, leading to stalls in the instruction stream while waiting for long-latency coprocessor operations and under-utilization of the coprocessor while executing other instructions. This work investigates the benefits of replacing simple in-order cores with a more complex out-of-order architecture to dynamically schedule instructions for the main core and coprocessor, optimizing resource utilization and reducing execution time. To ensure generality, an in-depth analysis was carried out by offloading instructions to a custom dummy coprocessor capable of emulating iterative and pipelined operations with arbitrary latency. Various workloads simulating real-world applications were executed on two variants of an open-source microcontroller equipped with a recent out-of-order core and the state-of-the-art CV32E40X in-order core, respectively. Results from Register Transfer Level simulations show that the former configuration executes up to 60% more instructions per cycle, with a modest 12% system area overhead on a 65 nm CMOS technology node.

Keywords: CPU microarchitecture; out of order; RISC-V; edge computing; coprocessors

check for updates

Citation: Caon, M.; Masera, G.; Martina, M. Ditching the Queue: Optimizing Coprocessor Utilization with Out-of-Order CPUs on Compact Systems on Chip. *Electronics* **2024**, *13*, 3018. https://doi.org/10.3390/electronics13153018

Academic Editor: Ricardo Martins

Received: 29 May 2024
Revised: 22 July 2024
Accepted: 23 July 2024
Published: 31 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

With the recent shift towards a data-driven computing paradigm, the demand for higher processing capabilities and better energy efficiency in edge-oriented Systems-on-Chip (SoCs) to overcome the bandwidth and latency limitations of the existing centralized computing infrastructure has dramatically increased. Artificial Neural Networks (ANNs) are increasingly being embedded in Internet of Things (IoT) devices to provide private and low-latency advanced functionalities. Health monitoring [1], robotics [2], and autonomous driving [3] are some of the possible applications of ANNs on resource- and energy-constrained devices.

In this context, heterogeneous SoCs have been proposed as a promising solution to address the inherent inefficiency of the von Neumann architecture, which has been the mainstay of embedded computers since their introduction. These systems accelerate computationally intensive parts of the workload by offloading them to specialized, tightly coupled coprocessors implementing domain-specific Instruction Set Architecture (ISA) extensions. Semantically rich instructions replace long sequences of scalar instructions, significantly reducing the pressure on the memory hierarchy and execution time, ultimately leading to higher throughput and energy efficiency at the system level. This benefit is amplified by the preference for Reduced Instruction Set Computer (RISC) instruction sets in low-power embedded systems, where executing computationally intensive tasks

on the Central Processing Unit (CPU) requires several additional instructions to move operands from the main memory into the CPU General-Purpose Registers (GPRs) and to construct potentially complex operations using elementary arithmetic instructions. In contrast, domain-specific instructions are usually designed to apply Single-Instruction Multiple-Data (SIMD) or vectorized operations, possibly implementing complex data flow patterns, in a single instruction. Consequently, the latency of instructions offloaded to the coprocessor is often significantly higher than that of scalar instructions executed in the CPU pipeline. Therefore, the effectiveness of such an approach partially depends on the ability of the host CPU to efficiently offload accelerated instructions without causing stalls in the main program while waiting for the coprocessor to produce the instruction results. To this end, exploiting Instruction-Level Parallelism (ILP) in addition to Data-Level Parallelism (DLP) is crucial. One possibility is to take advantage of strategies to leverage ILP that are already implemented in general-purpose, application-class microprocessors. In particular, Out-of-Order (OoO) instruction execution is a well-known technique to optimize the Instructions Per Cycle (IPC). It exposes a large pool of instructions to the CPU execution engine, allowing it to select those without data hazards with previous instructions for immediate execution, possibly breaking the original program order. Superscalar execution, which allows more than one instruction to be executed simultaneously, offers additional opportunities to exploit ILP, especially when coupled with OoO execution. Though the main purpose of OoO superscalar microprocessors is to achieve a superior IPC with conventional scalar instruction sets, their architectural solutions can also be beneficial in hiding the latency of long-latency coprocessors by reordering the program instructions and executing scalar instructions in parallel with those offloaded to the coprocessor. This work elaborates on this concept to propose an analysis of the performance benefits and limitations of employing an OoO-capable CPU to drive coprocessors with arbitrary latency as an alternative to the simple in-order CPUs employed in most edge-oriented scenarios. In particular, the main contributions of this article are twofold:

- It defines a general strategy to comprehensively evaluate the benefits of OoO instruction execution in the context of tightly coupled coprocessors using a variable-latency module and an automatic generator of test applications with different instruction compositions and instruction dependency patterns.
- It demonstrates the effectiveness of an existing open-source OoO CPU in covering the latency of long-latency coprocessors in a wide selection of workloads.

The rest of this paper is organized as follows: Section 2 discusses relevant use cases of tightly coupled coprocessors in edge applications and provides examples of existing OoO CPUs; Section 3 focuses on key aspects of the CPU microarchitecture selected for the experiments; Section 4 elaborates on the experimental setup and discusses the results obtained; finally, Section 5 concludes the paper.

2. Background and Related Works

2.1. Tightly Coupled Coprocessors

To motivate the need for better ILP exploitation in coprocessor-based computing systems, this section presents a brief review of some relevant examples of state-of-the-art tightly coupled coprocessors. While loosely coupled, memory-mapped accelerators typically offer superior peak performance and energy efficiency with large workloads, they lack flexibility and area efficiency, which are crucial for devices targeting low-effort application deployment across various domains. Similar considerations apply to reconfigurable solutions like Field-Programmable Gate Arrays (FPGAs) and Coarse-Grained Reconfigurable Arrays (CGRAs). Accordingly, this work focuses on small, versatile, general-purpose SoCs where maximum flexibility is paramount, driving the need for more fine-grained, instruction-level acceleration to enhance performance while maintaining programmability. In this context, custom instruction set extensions, implemented by tightly coupled coprocessors that cover a subset of common operations in the data flow graphs of the target applications, are a more suitable solution compared to the aforementioned alternatives [4],

and have gained traction since the recent diffusion of extendable ISAs [5]. As argued in [4], the effectiveness of a tightly coupled coprocessor can be measured by the reduction in overall program execution latency when using the custom instructions. Generally, the speedup is proportional to the amount of computation that is atomically implemented by a single offloaded instruction, albeit with some trade-offs in terms of flexibility. Moreover, physical constraints such as the available area and power budget or the target operating frequency often result in a multi-cycle instruction execution latency due to the iterative or pipelined implementation of custom instructions. This ranges from a few cycles for simple arithmetic operations, such as those in floating-point or ANN-focused coprocessors [6,7], to tens or hundreds of cycles for more complex data manipulation like that performed by cryptographic accelerators [8] or vector processing units [9]. While custom instructions alone significantly reduce the overall execution time, their latency creates opportunities for deeper ILP exploitation. Scalar instructions following the accelerated ones can be issued and executed in parallel, provided there are no data dependencies between them. This is a common scenario where scalar instructions perform housekeeping tasks or control operations. Dynamic instruction scheduling capabilities in the CPU can thus maximize the utilization of computing resources in both the CPU and the coprocessor, leading to further performance and energy efficiency improvements.

2.2. Out-of-Order Central Processing Units

To demonstrate the effectiveness of dynamic instruction scheduling in coprocessor-accelerated systems, the analysis presented in Section 4 compares the performance of the state-of-the-art in-order CPU with an OoO core when offloading instructions with variable latency to a configurable dummy Configurable-Latency Coprocessor (CLC), described in Section 4.1.2. The well-known CV32E40X microprocessor [5] was selected as the reference in-order core due to its design, which is specifically optimized for IPC, thus representing a best-case baseline for the experiments. On the other hand, selecting a candidate OoO core was challenging due to the limited availability of open-source designs and deployment examples. Three alternatives were considered: the Berkeley Out-of-Order Machine (BOOM) [10], Alibaba's OoO RISC-V core [11], and the LEN5 microprocessor [12]. Ultimately, LEN5 was chosen for its modularity and scalability, and its most relevant architectural features are described in Section 3. In particular, LEN5 has the following features:

- LEN5's modular microarchitecture facilitates the straightforward deployment of custom instruction set extensions and coprocessors. This modularity greatly simplified the integration of the CLC in the system, whereas the centralized execution control scheme used by the available OoO cores would have required significant modifications.
- Compared to other OoO cores, LEN5's architecture prioritizes scalability over performance, resulting in a more area-efficient design. Conversely, BOOM and Alibaba's cores are optimized for superscalar instruction execution, featuring wider issue windows and multiple execution units for each instruction class. While these features yield superior IPC when executing sequences of scalar instructions, they are less advantageous for the purposes of this work. When handling long-latency accelerated instructions, maximizing the number of scalar instructions executed per cycle could result in the CPU Execution Units (EUs) idling while awaiting the completion of offloaded instructions, thus not justifying the additional area and power overhead.
- The base variant of LEN5 targets bare-metal applications without a cache hierarchy, thereby offering a simpler interface with the host system bus. This interface is compatible with common bus protocols used in low-power Microcontroller Units (MCUs) like the OBI bus in the X-HEEP platform [13] selected for the experiments. Adapting the interface of other available cores would have required additional efforts.

Apart from the above considerations, the conclusions drawn from the analysis in Section 4 are general and applicable to any CPU with OoO capabilities. The minor differences in the obtained IPC due to the specific architectural choices do not impact the overall justification for considering OoO cores in coprocessor-accelerated systems.

3. Out-of-Order Central Processing Unit Microarchitecture

This section elaborates on the general concepts of OoO execution that are relevant to achieving optimal exploitation of the available computing resources when executing long-latency instructions using the LEN5 microprocessor presented in [12] and briefly introduced in Section 2.2 as a reference. Regardless, most of the insights discussed in this section are applicable to any OoO microprocessor.

In general, the key to hiding the latency of in-flight instructions is threefold:

1. It enables sufficient entry of instructions into the execution engine of the core, regardless of their readiness for execution. This approach maximizes the chances of identifying instructions that are independent of the previous ones, allowing for their immediate scheduling and execution, irrespective of the original program order.
2. It facilitates the parallel execution of multiple instructions (a superscalar design) so that if one instruction requires a prolonged time to complete, another can be dispatched to different EUs and executed concurrently.
3. It ensures the prompt retirement of completed instructions, potentially out of program order, to allow new instructions to enter the execution engine, thereby maintaining the EUs's productivity.

Item 1 leverages the generally valid assumption that a program is typically composed of two types of instructions: those implementing the computation core of the algorithm and those performing housekeeping tasks, such as updating loop iteration indexes or memory addresses. Generally, these two categories of instructions do not depend on each other, allowing them to be executed concurrently. However, this assumption does not hold in scenarios where the program control flow or memory access patterns are dependent on the computed results, as is common in iterative search algorithms. These scenarios are typically more challenging to predict, leading to significant performance penalties unless additional hardware resources and energy are allocated to more complex branch predictors. Nevertheless, the majority of modern data-intensive applications predominantly involve workloads that fit the former category, which is the focus of this work. LEN5 implements Item 1 and Item 2 with its OoO execution engine, which is based on Tomasulo's algorithm [14], extensively discussed in [15]. This implementation is enhanced with support for precise exceptions and efficient handling of speculative instructions. To promptly free the decode stage, newly decoded instructions are moved into designated buffers, known as Reservation Stations (RSs), until all associated execution conditions are met, such as the availability of the input operands from previous instructions or the resolution of previous speculative branches. A ReOrder Buffer (ROB) then accumulates the produced results and selects instructions ready for retirement, potentially out of program order, if no Write-After-Write (WAW) dependencies are present. It is important to note that the OoO commit strategy employed in the current version of LEN5 does not fully guarantee Item 3. Although newer instructions can be retired while older ones are still completing, the ROB allocation is still sequential, preventing the reassignment of freed entries to new issued instructions. The effects of this limitation are further discussed in Section 3.2.

As previously noted, LEN5's design philosophy emphasizes extendability and modularity, granting system implementers the flexibility to adapt its internal components to a broad spectrum of physical implementation constraints, operating conditions, and expected workloads. This adaptability is facilitated by a system-wide *valid-ready* handshake protocol, enabling each internal component to function independently of the latency affecting other parts of the architecture. To mitigate potential bottlenecks, LEN5 incorporates several internal buffers of configurable size that queue outstanding requests from upstream modules when downstream hardware is engaged, aiming to minimize stalls in the fetch and issue stages that would otherwise lead to system-level delays due to their in-order nature.

Similarly, RSs provide a flexible interface between the dynamically scheduled execution pipeline and the EUs. They enable straightforward integration of new processing elements tailored to specific domain or application needs with minimal modifications to

the core architecture. This is in contrast with microarchitectures relying on centralized control and instruction tracking mechanisms.

These principles also facilitate the optional inclusion of hardware enhancements like the *M* and *F/D* RISC-V extensions, which support integer operations and floating-point calculations, respectively. LEN5's design effectively minimizes the latency impact of these operations, which is critical for applications relying on complex calculations. In our analysis (Section 4), a dummy coprocessor was employed to maintain generality, though similar results were observed when dispatching long-latency Floating-Point Unit (FPU) instructions in data-intensive kernels.

To keep the EUs continually operational, LEN5 incorporates a speculative frontend with a configurable *gshare* branch predictor, Branch Target Buffer (BTB), and Return Address Stack (RAS), detailed in [12]. Figure 1 showcases LEN5's backend architecture and exemplifies resource allocation during execution, highlighting the system's adaptability and efficiency.

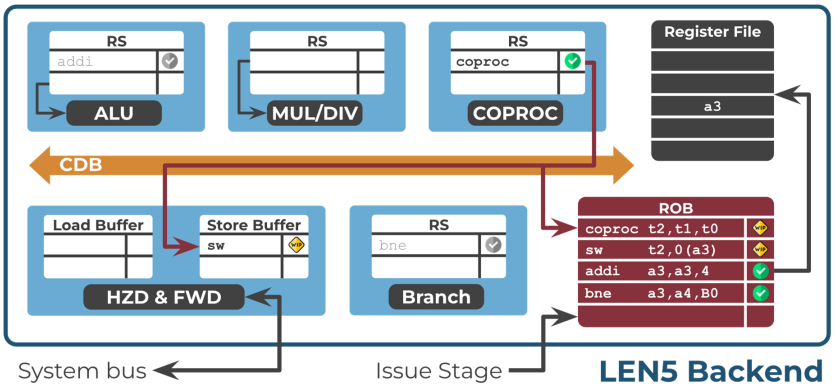

Figure 1. Block diagram of the backend of LEN5 with examples of OoO execution and commit. In the in-flight sequence of instructions (shown in the ROB), the `sw` is waiting for the result (`t2`) produced by the long-latency `coproc` instruction (Read-After-Write (RAW) hazard) that has just completed. The result is therefore broadcast to the `sw` instruction and the ROB through the Common Data Bus (CDB). In the meantime, the `addi` and `bne` instructions, which do not depend on `coproc`, have already completed their OoO execution (grayed out in their RS) and are eligible for commit.

3.1. Out-of-Order Instruction Execution

Building on the overview presented in [12], this section delves into the microarchitecture of LEN5's OoO execution engine, elucidating how its design contributes to the IPC improvements discussed in Section 4.

The decode stage of LEN5 handles several critical tasks:

- Translating incoming instructions into commands for the associated RS and EU.
- Allocating an available ROB entry to buffer the result of the instruction once it completes. From this moment, the instruction is uniquely tagged with the index of the assigned ROB entry.
- Fetching the operands for the instruction from the Register File (RF) or the ROB, if available.

A dedicated module monitors the status of each register, forwarding their values from the RF or the ROB based on whether the last instruction that wrote to a specific register has been committed. If the operand is from an in-flight instruction that has not yet completed, no forwarding occurs at issue time. Regardless, the instruction is dispatched to the target RS. Concurrently, its assigned tag (i.e., the ROB index) is linked with the instruction's destination register, effectively implementing a renaming mechanism that eliminates false Write-After-Read (WAR) dependencies. This strategy obviates the need for dedicated register renaming resources, as the RSs and ROB collectively function as a distributed register file.

Each instruction is uniquely associated with an ROB entry, setting the maximum number of in-flight instructions to the adjustable ROB size. If, at issue time, either the ROB or the destination RS is full, the processor experiences a stall. This is a primary cause of performance degradation when dealing with instructions with a very long execution latency, as discussed in Section 4. The size of an RS determines the maximum number of outstanding instructions of the same type that can be dispatched. This sets an upper limit on LEN5's capability to mask the latency of long-latency instructions, even when they are pipelined. Specifically, filling the RS causes a stall at issue time if instructions of varying types are not fetched subsequently. A full RS, when combined with the fetch of an instruction of the same type, leads to performance degradation similar to what is observed due to a full ROB.

Once an instruction is loaded into the target RS, and its operands are available, it is ready for execution. The order of selection at this stage is irrelevant for program coherence since the forwarding and renaming mechanism previously discussed ensures that all RAW hazards are correctly resolved. To prevent starvation, a *round-robin* approach determines the sequence of execution inside each RS; no new instructions are considered until all previously eligible instructions have been issued to the EU. The Branch Unit (BU) is an exception as its instructions must execute sequentially to accurately recover from branch mispredictions. Similarly, the Load–Store Unit (LSU) enforces additional checks to handle memory hazards correctly, employing store-to-load forwarding managed internally until additional space is needed, effectively creating a *level-zero* caching mechanism inspired by [16].

Once an instruction completes its execution, its result is stored in the RS, awaiting acceptance by the CDB to be forwarded to the ROB and other RSs while releasing the EU for other instructions of the same kind. Branch instructions have the highest priority on the CDB since their resolution has the greatest potential to unlock new commit and memory update operations, thereby freeing up resources for new instructions. All other instructions are subject to a *round-robin* selection policy. The CDB implements an efficient forwarding mechanism that connects one instruction's output to potentially all other in-flight instructions through a simple 1-to-N interconnect rather than a fully parallel N-to-N crossbar. The tag of the instruction producing the result is checked against the one associated with each operand of a waiting instruction to trigger the forwarding. Since instructions' operands are tagged at issue time in program order with the tag of the *latest* instruction that wrote to the corresponding register, this mechanism effectively ensures the correct resolution of RAW hazards. It is possible for multiple instructions to become ready for execution as a result of operand forwarding. In such cases, they may be selected for execution within the same cycle, provided that sufficient EUs are available.

3.2. Out-of-Order Instruction Commit

As mentioned in the previous section, an ROB entry is allocated for each new instruction at issue time, in program order. Once the result of a completed instruction is received on the CDB, it is buffered within the allocated ROB entry. Subsequently, any instruction that requires the destination register of the completed instruction as a source operand has its value forwarded from the ROB, marking it as immediately ready for execution.

Once buffered in the ROB, an instruction becomes eligible for commit into the physical register file, if necessary, and for retirement. LEN5's ROB includes two commit slots, which facilitate instruction commitment in both program order and out of program order. Out-of-order commit is permitted under the following conditions for a given instruction:

- Its execution is complete, making the instruction result available in the ROB.
- It is no longer speculative, meaning all previous branch predictions have been validated.
- It did not trigger any exceptions.

- There are no newer instructions eligible for commit that would write to the same destination register (WAW hazard). In such cases, the older instruction is simply retired without updating the RF.

Due to these stringent conditions, the out-of-order commit slot never advances past mispredicted branches or exception-raising instructions, ensuring consistency with the program order. The in-order commit slot consistently points to the oldest instruction awaiting commitment. If a mispredicted branch or an exception-raising instruction is committed through the in-order slot, the entire execution pipeline, including the ROB, is flushed, and execution restarts. To minimize the penalty associated with branch mispredictions, the BU immediately communicates the correct branch target to the frontend upon resolving a misprediction.

However, the LEN5 OoO commit process does not permit the reallocation of ROB entries freed by the out-of-order commit slot to new instructions. Instead, ROB entries are allocated in a fixed sequence. Nevertheless, the OoO commit process expedites the clearing of the ROB when an older instruction, waiting for its result, is selected for commitment. This reduces the issue stage's backpressure, thus diminishing the stalls caused by an overcommitted ROB. An enhanced OoO commit strategy, currently under development, would eliminate the in-order allocation constraint, enabling the issuance of a greater number of independent instructions after a very-long-latency instruction than the ROB size permits, thereby addressing the performance bottleneck discussed in Section 4.

4. Experimental Results

Assessing the effectiveness of employing an OoO-capable CPU to optimize the offloading process necessitates a comprehensive test plan and simulation environment incorporating a diverse array of applications, instruction set extensions, and coprocessor characteristics. While there are several open-source examples of tightly coupled coprocessors, their variations in communication protocols and physical integration with their intended host CPU pose significant challenges in creating a unified test plan. Moreover, the generality of the results is constrained by the specific combinations of CPU and Instruction Set Architecture (ISA), coprocessor architecture, and instruction-level dependency patterns, demanding substantial technical effort and time to achieve sufficient comprehensiveness. Instead, this work aims to provide general guidance on the benefits and limitations of using an OoO-capable CPU to drive coprocessors with arbitrary architectures (i.e., iterative or pipelined data flow) and latencies. To this end, this study employs a synthetic approach composed of two main components: a Configurable-Latency Coprocessor (CLC) whose latency can be specified at runtime and an automatic code generator that assembles test applications featuring various combinations of instructions (i.e., scalar and accelerated) and data dependencies. The LEN5 and CV32E40X [5] CPUs, representing an OoO and an in-order core, respectively, are used as host CPUs in the experiments. Their average IPC is utilized to evaluate their capacity to continue program execution while waiting for offloaded instructions to complete. A higher IPC indicates more effective exploitation of coprocessor resources and, consequently, a reduced overall execution time. As noted in Section 2.2, both cores were selected for their modular architecture, ease of extendability, and interface compatibility with the X-HEEP platform, chosen as an example Microcontroller Unit (MCU). Additionally, their focus on optimizing the IPC within their respective CPU classes plays a crucial role. The differences in IPC performance when managing instruction offloading primarily stem from their contrasting execution paradigms—dynamic for LEN5 and static for CV32E40X. While low-level microarchitectural details influence the IPC achieved in the experiments, they do not significantly alter the conclusions of this work, which are generalized at the end of this section.

The subsequent sections will detail the software and hardware configurations used in the experiments and discuss the results obtained.

4.1. Experimental Setup

4.1.1. System Configuration

For the experiments, LEN5 parameters were set to align with the *Max Performance* configuration as described in [12]. This setup includes a 32-entry ROB, an 8-entry Arithmetic Logic Unit (ALU) RS, a 4-entry BU, a 4-entry multiplication and division unit featuring a 2-stage pipelined multiplier and a serial divider, a 16-entry Store Buffer (SB), and an 8-entry Load Buffer (LB). This configuration, whose post-synthesis characteristics are detailed in Table 1, imposes a minimal 12% area overhead on the host system compared to the CV32E40X core in a basic 256 KiB X-HEEP configuration. The choice of the *Max Performance* configuration was intended to maximize the benefits of out-of-order execution, thereby offering the best-case performance improvements across a comprehensive set of test applications. While optimized LEN5 configurations or other OoO CPU microarchitectures might provide even better trade-offs in terms of area and power consumption, these are not the focus of this work. A dedicated bridge was developed to adapt LEN5's data memory requests to X-HEEP's 32-bit bus. The bridge splits 64-bit requests into two 32-bit ones and ensures correct alignment and handshaking. Because the application software used in the experiments exclusively relies on 32-bit data, most of the memory accesses complete in a single cycle, making the performance impact of the bridge negligible. Apart from these changes in the CPU subsystem, no other modifications were made to the X-HEEP platform.

Table 1. Area and clock frequency of the LEN5 *Max Perf* variant from [12], implemented on a low-power 65 nm CMOS technology node.

	LEN5 *Max Perf*	**CV32E40X**
Clk Freq. [MHz]	438	360
Area [1×10^3 µm^2]	423	49
Area [kGE] [a]	294	34
Relative System Area [b]	1.12	1.00

[a] GE is the 2-input drive, strength-one NAND gate equivalent area. [b] X-HEEP host system with 256 KiB memory.

4.1.2. Configurable-Latency Coprocessor Architecture

The CLC was engineered to comprehensively emulate the behavior of coprocessors with arbitrary latencies, accommodating both iterative and pipelined instructions. The operational mode—iterative or pipelined—and the latency are configurable at runtime using a dedicated control signal and one of two input operands. This input operand specifies which of the internal pipeline registers serves as the output for the coprocessor. A Finite-State Machine (FSM) manages the handshaking with the CPU according to the selected mode. Specifically, in pipelined mode, the coprocessor can accept a new input transaction every cycle. In contrast, in iterative mode, it must wait to accept a new input until the CPU has acknowledged the output of the previous transaction. The maximum latency and the number of pipeline stages are adjustable Register Transfer Level (RTL) parameters. The second operand is propagated to the output after a predetermined number of cycles, facilitating experiments with various data dependency patterns in the software. Additionally, the coprocessor adheres to a *valid-ready* handshake protocol for managing both input and output transactions.

From a software standpoint, the CLC is controlled by a custom RISC-V ISA extension that defines two I-type instructions for the iterative and pipelined operating modes, respectively. Each accepts an input GPR as the input operand and a 12-bit immediate value encoding the desired latency or pipeline register to be used as output.

```
xdummy.iter rd, rs1, imm # iterative mode, imm is the latency
xdummy.pipe rd, rs1, imm # pipelined mode, imm is the number of stages
```

This architecture allows for a pure software-controlled configuration of the CLC, making it possible to explore a wide range of scenarios without the need to modify the RTL description and recompile the simulation model, resulting in a streamlined and efficient exploration process.

The CLC is integrated into LEN5 through the following key modifications to the core architecture:

1. The new custom instructions are incorporated into the main decoder, specifying the expected control signals for the CLC, the necessary source operands, and the result type so that the CPU can correctly manage dependencies and commit.
2. A new FOUR-entry RS is adapted from the ALU one and added to LEN5's backend with negligible impact on the overall area.
3. The CLC is connected to the dedicated RS. Dynamic synchronization between the CLC and the CPU is inherently achieved by the system-wide *valid-ready* handshake protocol.

On the other hand, the integration into the CV32E40X CPU is achieved through the CORE-V eXtension Interface (CV-X-IF) [17] using a bridge to convert the simple *valid-ready* interface of the CLC into compliant instruction-offloading transactions.

4.1.3. Configurable Test Applications

As previously outlined in Section 3, instructions within a typical data-intensive workload fall into two distinct categories: those that form the core of the processing algorithm, and those that perform housekeeping tasks, such as updating iteration counters and managing the memory addresses of input and output data. The processing core generally consists of a loop that processes input data iteratively, where each iteration may or may not depend on the results of the previous one. Housekeeping tasks, on the other hand, are typically independent of the processing core as loop indices and memory addresses usually do not directly relate to the outcomes of the algorithm. For instance, linear algebra vector kernels and general-purpose algorithms that include floating-point operations are practical examples that reflect this bifurcation. Tasks such as loop maintenance, exception handling, and memory management are executed through sequences of instructions that lack direct data dependencies with the computational results produced by the core code. To assess the impact of coprocessor latency on the execution times of workloads with varying dependency patterns, an automatic application generator was developed.

This tool generates the main function of a C program containing a loop with a single CLC instruction followed by a block of single-cycle arithmetic and logic instructions derived from the base RISC-V ISA. The Number of Instructions per Block (NIB) and the latency of the CLC are both configurable, enabling detailed exploration of how increasing coprocessor latencies and different quantities of other instructions affect performance. The simplest configuration, which illustrates the CLC operating independently of other instructions, is presented in Listing 1a.

Listing 1. Example of assembly code with dummy instructions.

```
B4:
xdummy.iter a0, a1, 5
add         a2,a4,a3
xor         a2,a5,a4
addiw       a5,a5,-1
bnez        a5, B4
```

(a) Loop with independent iterations.

```
B4:
xdummy.iter a0, a0, 5
add         a2,a4,a5
xor         a2,a4,a5
addiw       a3,a3,-1
bnez        a3, B4
```

(b) Loop with dependent iterations.

```
B4:
xdummy.iter a0, a0, 5
jal         [bookeeping]
add         a0,a0,s1
addiw       s0,s0,-1
bnez        s0, B4
```

(c) Loop with dependent iterations and housekeeping.

To emulate algorithms with dependent loop iterations, such as reduction and accumulation operations, the output register of the CLC instruction in a certain loop iteration can optionally be used as an input register for the same instruction in the next loop iteration, as shown in Listing 1b. The tool can also add to the main loop a function call to a routine that performs a configurable number of instructions that are independent of the offloaded one. The appropriate compiler directives were employed to ensure that the housekeeping routine is never inlined, to evaluate the impact of LEN5 branch prediction over the in-order, non-speculative CPU. An example of such a case is reported in Listing 1c.

4.2. Instructions per Cycle Analysis

As explained in Section 3.1, LEN5's dynamically scheduled execution engine can select instructions as soon as their operands are ready. To evaluate the effectiveness of OoO execution in the presence of long-latency instructions, LEN5's IPC is compared to that of CV32E40X while executing a large set of automatically generated test applications. In particular, tests were conducted with the CLC latency varying from one to twenty cycles. For each latency value, the NIB was swept from one to twenty as well. Several test program configurations were considered, each time repeating the test with the CLC in iterative and pipelined mode. The collected data are reported in the following paragraphs.

All the tests reported in the following sections were compiled for both CPUs using GCC with the `-O2` optimization level. Only 32-bit variables were used to generate analogous assembly code for the 64-bit LEN5 core and the 32-bit CV32E40X core. The IPC was measured as a ratio between the values of the `minstret` and `mcycle` control and status registers, accessed immediately before and after the test application code. The C++ RTL simulation model used to run the simulations was compiled using Verilator.

The first set of tests is conducted using the simplest kind of workload: a loop where a single CLC instruction is followed by a block of independent single-cycle instructions. Each loop iteration produces a result that does not depend on the previous operations. The IPC obtained by LEN5 and CV32E40X is reported in Figure 2. As shown, the performance degradation on the in-order CPU is significant even for the lowest latency values and only marginally improves with a higher NIB. A similar trend is observed with iterative (left) and pipelined (right) instructions. In contrast, LEN5 is able to keep an optimal IPC close to 1 even for the highest latency values, provided that there are enough single-cycle instructions to execute while waiting for the CLC instruction to complete.

In LEN5, the pipelined case shows significantly better performance even with low NIB values thanks to the possibility of issuing multiple coprocessor instructions to the four-entry reservation station before having to stall the issue stage. With a twenty-stage pipeline, a new coprocessor instruction can be accepted every five cycles. Therefore, four single-cycle instructions, including the `add` and `bnez` instructions implementing the loop, are, in theory, enough to prevent stalls due to a full RS. However, the internal handshake between the RSs, the CDB, and the issue stage adds a few cycles before an entry in the RS is actually freed, which motivates the performance degradation observed for NIB values below 5 instead of 2.

Despite this, LEN5 succeeds in dynamically scheduling multiple iterations of the for loop at the same time, pipelining their instruction execution. In the iterative case, while it is possible to hold up to four coprocessor instructions from the four initial loop iterations in its RS, they cannot be executed in parallel, causing the issue to stall on subsequent iterations because of the structural hazard on the full RS. Therefore, in the iterative case, a high IPC is possible only if the number of single-cycle instructions equals the latency of the coprocessor minus one (the coprocessor instruction itself). Regardless, LEN5 achieves an IPC which is up to 50% higher than CV32E40X with both iterative and serial instructions. This test also proves how the LEN5 branch predictor is able to correctly predict the loop condition, avoiding expensive flushes and keeping the IPC close to 1.

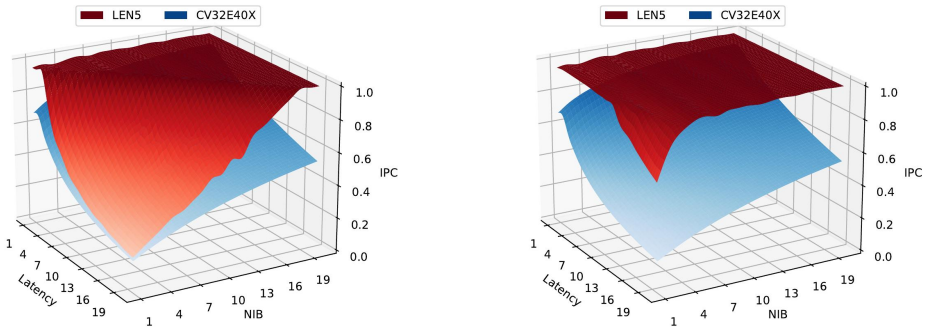

Figure 2. IPC comparison between LEN5 and CV32E40X using an example application without housekeeping and dependencies when using iterative (**left**) and pipelined (**right**) CLC.

If the system running the application is required to perform housekeeping actions while the accelerator is running, then the ability to execute those independent instructions becomes crucial. In this experimental setup, the housekeeping routine, containing a configurable number of single-cycle instructions, is invoked while the CLC runs. As illustrated in Figure 3, the trend in both the CPUs resembles the one from the previous experiment except, this time, the performance penalty with a low NIB is lower due to the presence of the additional instructions to call and execute the bookkeeping routine.

Like before, the in-order processor must pause before executing the subsequent loop iteration, whereas LEN5 can continue queuing new instructions. For low NIB values, LEN5 achieves more than twice the IPC of the in-order code.

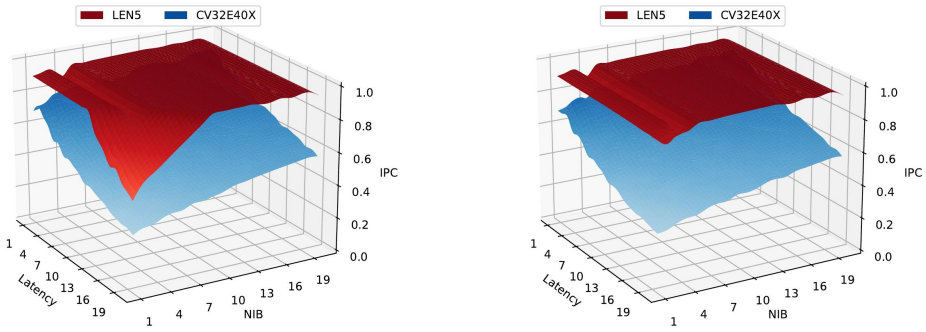

Figure 3. IPC comparison between LEN5 and CV32E40X using an example application with housekeeping and without dependencies when using iterative (**left**) and pipelined (**right**) CLC.

The next test emulates a scenario where the loop iterations are interdependent, creating RAW hazards between consecutive coprocessor instructions. Such cases limit the exploitation of pipelined coprocessors, resulting in the same performance penalty as when relying on an iterative coprocessor. This effect is clearly shown by Figures 4 and 5, which report the IPC when executing the exact same applications with iterative and pipelined CLC instructions, respectively. In both cases, the positive effect of housekeeping instructions observed in Figure 2 is visible, offering more instructions to cover the latency of the coprocessor and thus increasing the IPC.

The left examples for both experiments, where no housekeeping was performed, show that the two CPUs reach the same IPC for the combination of highest latency and lowest NIB. In this case, the body of the for loop solely contains the coprocessor instruction, offering no opportunities for the OoO core to perform any operation besides resolving the branch condition and updating the loop counter. This scenario is also true for the four-stage CV32E40X, which is able to fetch and execute two additional scalar instructions before stalling in the writeback stage, waiting for the external coprocessor to provide its result.

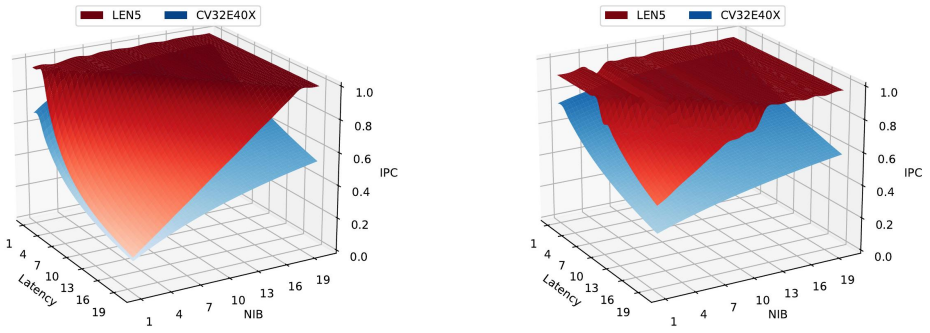

Figure 4. IPC comparison between LEN5 and CV32E40X using an example application with dependencies when using iterative CLC without housekeeping (**left**) and with it (**right**).

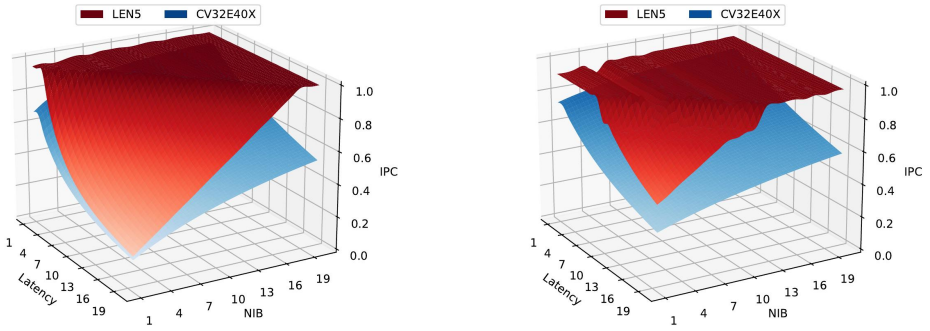

Figure 5. IPC comparison between LEN5 and CV32E40X using an example application with dependencies when using pipelined CLC without housekeeping (**left**) and with it (**right**).

One final experiment is set up to highlight the limitations of the current LEN5 commit stage when dealing with very-long-latency coprocessors, such as those implementing Post-Quantum Cryptography (PQC) cryptographic functions. In these cases, the coprocessor latency exceeds the ROB size. As previously noted, due to the in-order allocation of the ROB, it is not possible to issue more instructions than those that can fit inside the ROB. The simplest test case is run with the NIB set equal to the coprocessor latency to demonstrate the performance degradation resulting from this limitation. This setup matches the condition where, in all previous tests, LEN5 could reorder instructions and completely hide the coprocessor's execution latency. The experiment is repeated for a wide range of latency values, with the results reported in Figure 6. As shown, once the latency value approaches the ROB size (32), the IPC of LEN5 starts to degrade, eventually reaching an asymptotic limit of 0.5, similar to the in-order CPU. This degradation occurs because, once the ROB is full, the issue stage must stall until the oldest in-flight instruction, the coprocessor instruction, is retired.

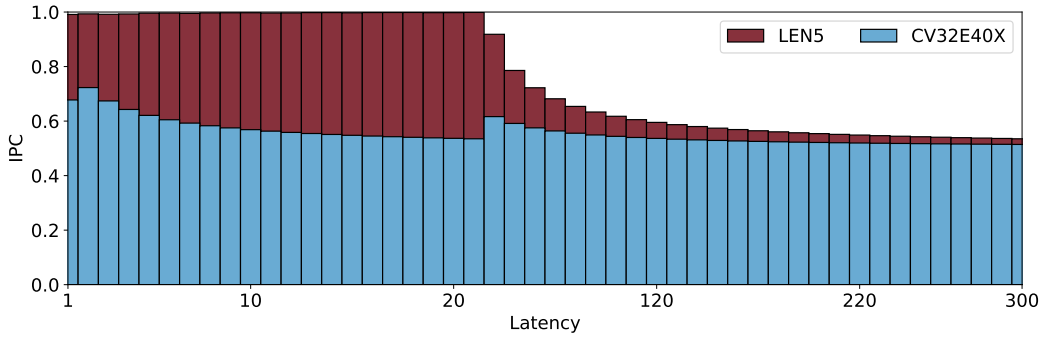

Figure 6. IPC comparison between LEN5 and CV32E40X as the CLC latency and NIB increase.

The value of the asymptotic limit can be verified by expressing the IPC as a function of the coprocessor latency L, the ROB size R, and the number of other instructions N ($N+1$ instructions in total). At the beginning of the program, one coprocessor instruction and $R-1$ single-cycle instructions can be issued. The single-cycle instructions are executed while waiting for the coprocessor result. After this point, the issue stage stalls, and no more instructions can be issued until the coprocessor completes. When that happens, the coprocessor instruction is removed from the ROB, which becomes empty, allowing the remaining $N-(R-1)$ single-cycle instructions to be issued and executed one per cycle until the next coprocessor instruction is fetched, and the cycle repeats. The IPC during one iteration can therefore be expressed as follows:

$$IPC = \frac{1+N}{L+N-R+1}$$

In the hypothesis of this experiment, $N = L$ (the instructions implementing the loop are neglected); therefore,

$$\lim_{L=N\to\infty} IPC = 0.5$$

Because compensating for large values of the latency L by increasing the ROB size R is not sustainable from a hardware resources standpoint, a better solution is to manage the ROB in such a way that new issuing instructions can be allocated to free ROB entries regardless of their position, overcoming the limitations of the current in-order ROB allocation policy. This approach guarantees a possibly infinite window of candidate instructions to be selected for execution while waiting for the long-latency instruction to complete. However, the additional hardware resources needed to ensure the correct resolution of WAW hazards and consistently recover from branch mispredictions and exceptions are justified only if the control, dependency patterns, and instruction count in the expected workload provide enough instructions eligible for execution before structural hazards emerge in the accelerator reservation station. In the case of pipelined accelerators, this last problem can be mitigated by removing instructions from an RS as soon as they are selected for execution, relying on the accelerator to propagate the information required to commit the result in the ROB and to handle the CDB handshake.

5. Conclusions

This work investigates the performance benefits and limitations of replacing conventional in-order cores with OoO implementations within edge-oriented SoCs. The study demonstrates the performance gains achievable through dynamic instruction scheduling by comparing the IPC of the OoO LEN5 RISC-V CPU with that of the state-of-the-art CV32E40X in-order CPU. A variety of workloads with variable-latency coprocessor-accelerated instructions and differing data dependency patterns were executed. Utilizing a Configurable-Latency Coprocessor (CLC) and an automatic code generator, the study simulated a broad spectrum of use cases, revealing substantial performance improvements across all workloads when compared to the in-order core, even when managing instructions with extensive latencies. LEN5 consistently achieves near-optimal IPC in parallelizable workloads devoid of data dependencies and markedly mitigates performance degradation in loops with interdependent iterations. Overall, the experimental results indicate that employing cores with OoO capabilities and speculative branch prediction can enhance the IPC by up to 60% compared to in-order cores.

Considering the modest system-level area overhead introduced by the added complexity of the OoO CPU, this study advocates for the integration of OoO execution in edge-oriented SoCs as a viable solution to fully leverage the benefits of tightly coupled coprocessors. This approach contributes significantly to reducing overall execution time and energy consumption. Furthermore, the study exposes a notable limitation of LEN5 when handling instructions whose latency surpasses the size of the ROB. In such scenarios, the processor struggles to utilize unrelated instructions to mask the coprocessor latency,

leading to diminishing returns in terms of IPC gains. This effect asymptotically approaches the performance level of an in-order core for extremely-long-latency operations. These findings suggest that more sophisticated OoO instruction commit schemes could unlock additional performance improvements without necessitating the complexities associated with enlarging the ROB or expanding other internal buffers.

In light of these findings, future research on extendable, CPU-based embedded systems should investigate the balance between performance optimization and the area and power overhead introduced by OoO execution engines. The insights gained from this study indicate that further microarchitectural enhancements are necessary to fully capitalize on the energy efficiency and performance benefits associated with the increasingly prevalent paradigm of tightly coupled coprocessor acceleration in edge SoCs.

Author Contributions: Conceptualization, methodology, software, validation, and writing, M.C.; funding acquisition, review, and supervision, G.M. and M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by NODES project no. ECS00000036, which has received funding from the MUR-M4C2 1.5 of PNRR funded by the European Union NextGenerationEU. This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union NextGenerationEU.

Data Availability Statement: The original contributions presented in the study are included in the article. The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ALU Arithmetic Logic Unit
ANN Artificial Neural Network
BTB Branch Target Buffer
BU Branch Unit
CDB Common Data Bus
CGRA Coarse-Grained Reconfigurable Array
CLC Configurable-Latency Coprocessor
CPU Central Processing Unit
DLP Data-Level Parallelism
EU Execution Unit
FPGA Field-Programmable Gate Array
FPU Floating-Point Unit
FSM Finite-State Machine
GPR General-Purpose Register
ILP Instruction-Level Parallelism
IoT Internet of Things
IPC Instructions Per Cycle
ISA Instruction Set Architecture
MCU Microcontroller Unit
NIB Number of Instructions per Block
OoO Out-of-Order
PQC Post-Quantum Cryptography
LB Load Buffer
LSU Load–Store Unit
RAS Return Address Stack
RAW Read-After-Write
RISC Reduced Instruction Set Computer
RF Register File
ROB ReOrder Buffer

RS	Reservation Station
RTL	Register Transfer Level
SB	Store Buffer
SIMD	Single-Instruction Multiple-Data
SoC	System-on-Chip
WAR	Write-After-Read
WAW	Write-After-Write

References

1. Daoud, H.; Bayoumi, M.A. Efficient Epileptic Seizure Prediction Based on Deep Learning. *IEEE Trans. Biomed. Circuits Syst.* **2019**, *13*, 804–813. [CrossRef] [PubMed]
2. Hoshino, S.; Kubota, Y. Mobile Robot Motion Planning through Obstacle State Classifier. In Proceedings of the 2023 62nd Annual Conference of the Society of Instrument and Control Engineers (SICE), Tsu, Japan, 6–9 September 2023; pp. 120–126.
3. Wang, Y.; Jiang, J.; Li, S.; Li, R.; Xu, S.; Wang, J.; Li, K. Decision-Making Driven by Driver Intelligence and Environment Reasoning for High-Level Autonomous Vehicles: A Survey. *IEEE Trans. Intell. Transp. Syst.* **2023**, *24*, 10362–10381. [CrossRef]
4. Galuzzi, C.; Bertels, K. The Instruction-Set Extension Problem: A Survey. *ACM Trans. Reconfigurable Technol. Syst.* **2011**, *4*, 1–28. [CrossRef]
5. Gautschi, M.; Schiavone, P.D.; Traber, A.; Loi, I.; Pullini, A.; Rossi, D.; Flamand, E.; Gürkaynak, F.K.; Benini, L. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 2700–2713. [CrossRef]
6. Mach, S.; Schuiki, F.; Zaruba, F.; Benini, L. Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2020**, *29*, 774–787. [CrossRef]
7. Garofalo, A.; Tagliavini, G.; Conti, F.; Rossi, D.; Benini, L. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 9–13 March 2020; pp. 186–191.
8. Fritzmann, T.; Sigl, G.; Sepúlveda, M.J. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Cryptol. ePrint Arch.* **2020**, *2020*, 446. [CrossRef]
9. Perotti, M.; Cavalcante, M.; Wistoff, N.; Andri, R.; Cavigelli, L.; Benini, L. A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design. In Proceedings of the 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP), Gothenburg, Sweden, 12–14 July 2022; pp. 43–51.
10. Zhao, J.; Korpan, B.; Gonzalez, A.; Asanovic, K. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In Proceedings of the Fourth Workshop on Computer Architecture Research with RISC-V, Virtual, 29 May 2020.
11. Chen, C.; Xiang, X.; Liu, C.; Shang, Y.; Guo, R.; Liu, D.; Lu, Y.; Hao, Z.; Luo, J.; Chen, Z.; et al. Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension: Industrial Product. In Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 30 May–3 June 2020; pp. 52–64.
12. Caon, M.; Petrolo, V.; Mirigaldi, M.; Guella, F.; Masera, G.; Maurizio, M. Seeing Beyond the Order: A LEN5 to Sharpen Edge Microprocessors with Dynamic Scheduling. In *CF '24: Proceedings of the 20th ACM International Conference on Computing Frontiers, Ischia, Italy, 7–9 May 2024*; Association for Computing Machinery: New York, NY, USA, 2024.
13. Machetti, S.; Schiavone, P.D.; Müller, T.C.; Peón-Quirós, M.; Atienza, D. X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators. *arXiv* **2024**, arXiv:2401.05548.
14. Tomasulo, R.M. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.* **1967**, *11*, 25–33. [CrossRef]
15. Hennessy, J.L.; Patterson, D.A. *Computer Architecture: A Quantitative Approach*; Morgan Kaufmann: Burlington, MA, USA, 2017.
16. Alves, R.; Ros, A.; Black-Schaffer, D.; Kaxiras, S. Filter caching for free: the untapped potential of the store-buffer. In *ISCA'19: Proceedings of the 46th International Symposium on Computer Architecture, Phoenix, AZ, USA, 22–26 June 2019*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 436–448.
17. OpenHW Group. OpenHW Group Specification: Core-V eXtension Interface (CV-X-IF). 2023. Available online: https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/ (accessed on 28 May 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.