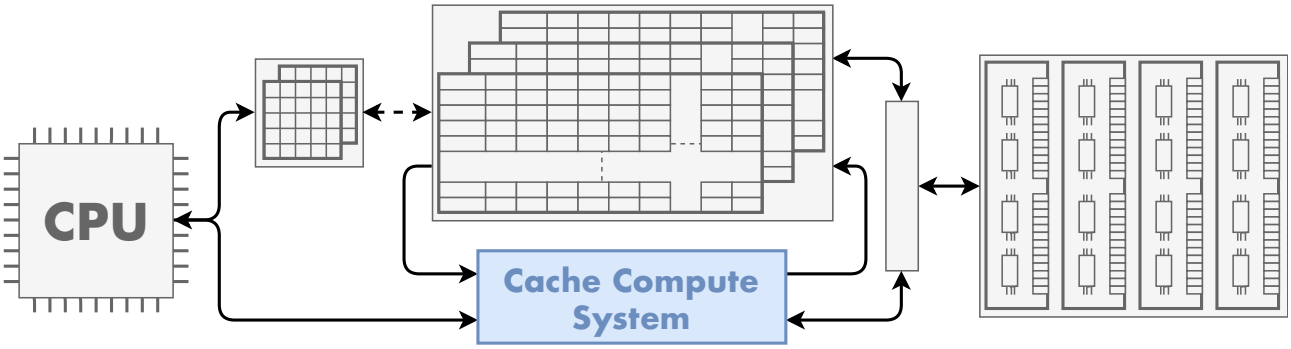
TÉCNICO LISBOA



Exploiting Processing Near Cache for Memory Bound Vector Operations

João Miguel Morgado Pereira Vieira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Doutor Pedro Filipe Zeferino Aidos Tomás
Doutor Gabriel Falcão Paiva Fernandes

Examination Committee

Chairperson: Doutor António Manuel Raminhos Cordeiro Grilo
Supervisor: Doutor Pedro Filipe Zeferino Aidos Tomás
Members of the Committee: Doutor Rui António Policarpo Duarte

November 2018

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Aos meus orientadores, Professor Pedro Tomás, Professor Nuno Roma e Professor Gabriel Falcão, agradeço toda a ajuda e acompanhamento que me deram. Devo-lhes: as reuniões, quase diárias, que me ajudaram a planear o trabalho e a ultrapassar obstáculos, que por vezes pareciam intransponíveis; todo o tempo dispensado, incluindo fins-de-semana, a escrever e a rever os artigos que submetemos. Ao Professor Pedro Tomás e ao Professor Nuno Roma tenho a agradecer a oportunidade que me proporcionaram de desenvolver investigação em associação com o INESC-ID através de uma bolsa de investigação. Em particular, ao Professor Pedro Tomás gostaria de agradecer todas as conversas, que em mim criaram novas ambições e me ajudaram a planear o meu futuro. Ao professor Gabriel Falcão agradeço a oportunidade de estágio na École Polytechnique Fédérale de Lausanne (EPFL), onde passei o meu último semestre letivo a desenvolver investigação em parceria com algumas das mentes mais brilhantes que já conheci.

I would like to thank Paolo Ienne for receiving me as an intern, allowing me to work at Processor Architecture Laboratory (LAP), EPFL. The seven months spent working there allowed me to develop research on structures to perform computation *near-memory*, which was propelled by opinions and suggestions of experts in the field and access to state-of-the-art equipment. Furthermore, this internship constitutes the first contact I have ever had with research, and I could not ask for a better opportunity.

I thank all the fantastic people I had the opportunity to work with at LAP, namely, Mikhail Asiatici, Andrew Becker, Lana Josipovic, Stefan Nikolic and Andrea Guerrieri; to my flatmates Leonardo Kokot and Anastasiia Kucherenko; to André Guignard and René Beuchat, who I had the opportunity to meet, speak with and learn from; to Sukriti Gupta, Amna Masood, Kunal Goyal and Gabor Csordas.

I especially thank Chantal Schneeberger, whose kindness and useful advice made my experience in Switzerland much better than it would be without her.

À minha família e amigos agradeço o apoio incondicional. Agradeço em especial aos meus pais, pois sem eles não teria ido longe. Agradeço ao meu tio, com quem passei a maior parte dos 5 anos do curso, e cuja companhia e apoio me deram força para continuar.

Em geral, agradeço a todos aqueles com quem o meu percurso se cruzou, e contribuíram para o meu crescimento pessoal e profissional.

I would like to acknowledge the financial support from Fundação para a Ciência e a Tecnologia (FCT) under projects UID/CEC/50021/2013, UID/EEA/50008/2013, and PTDC/EEI-HAC/30485/2017 (HAnDLE).

Abstract

To reduce the average memory access time, most current processors make use of a multilevel cache subsystem. However, despite the proven benefits of such cache structures in the resulting throughput, conventional operations such as copy, simple maps and reductions still require moving large amounts of data to the processing cores. This imposes significant energy and performance overheads, with most of the execution time being spent moving data across the memory hierarchy. To mitigate this problem, a Cache Compute System (CCS) that targets memory-bound kernels such as map and reduce operations is proposed. The developed CCS takes advantage of long cache lines and data locality to avoid data transfers to the processor, and exploits the intrinsic parallelism of vector compute units to accelerate a set of 48 operations commonly used in map and reduce patterns. The CCS architecture was validated by simulation using gem5 and compared with a PULPino soft-core running on a Xilinx ZYNQ-7 ZC706 Evaluation Board. Furthermore, the CCS was integrated in a system featuring an MB-LITE soft-core and a memory subsystem, and the system was implemented in a Xilinx Virtex-7 VC709 Development Board. When compared to the MB-Lite core, the proposed CCS presents performance improvements in the execution of the commands ranging from $18\times$ to $94\times$, and energy efficiency gains from $11\times$ to $67\times$. On the other hand, the CCS can perform between $132\times$ and $401\times$ better than PULPino for 1024 32-bit elements operands.

Keywords

Near Data Processing, *Near-cache* computing, Memory bound operations, map and reduce, Single Instruction Multiple Data, Cache Compute System, RTL implementation

Resumo

Para reduzir o tempo médio de acesso à memória, a maior parte dos processadores recentes incluí vários níveis de cache. No entanto, apesar dos benefícios que estas estruturas trazem ao nível do desempenho, operações convencionais como cópias, mapeamentos e reduções simples requerem a transferência de grandes quantidades de dados para os núcleos de processamento. Isto impõe custos significativos em termos de desempenho e energia, sendo a maior parte do tempo de processamento despendido a transferir dados entre o processador e a hierarquia de memória. Para mitigar este problema, um Sistema de Computação em Cache (designado por CCS) que acelera a execução de funções cuja performance é limitada pela largura de banda da memória, tais como operações de mapeamento e reduções, é apresentado no âmbito desta tese. O CCS tira partido de linhas de cache longas e da localidade dos operandos para evitar a transferência de dados para o processador, e explora o paralelismo intrínseco das unidades de computação vetoriais para acelerar 48 operações frequentemente usadas em padrões de mapeamento e redução. A arquitetura do CCS foi validada por simulação usando o simulador gem5 e comparada com o *soft*-processador PULPino a executar numa placa de avaliação Xilinx ZYNQ-7 ZC706. Adicionalmente, o CCS foi integrado num sistema em conjunto com o *soft*-processador MB-LITE e um subsistema de memória, e o sistema foi implementado numa placa de desenvolvimento Xilinx Virtex-7 VC709. Comparado com o *soft*-processador MB-LITE, o CCS apresenta melhorias de desempenho na execução de comandos de $18\times$ a $94\times$, e melhorias de eficiência energética de $11\times$ a $67\times$. Por outro lado, o desempenho do CCS é melhor que o do PULPino entre $132\times$ e $401\times$ para operandos com 1024 elementos de 32 bits.

Palavras Chave

Processamento perto dos dados, Processamento perto da cache, Operações limitadas pela largura de banda da memória, Operações de mapeamento e redução, Instrução única para dados múltiplos Sistema de Computação em Cache, Implementação RTL

Contents

List of Figures

List of Tables

Listings

Acronyms

ALM Adaptive Logic Module

ALU Arithmetic and Logic Unit

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

BP Block Partition

BRAM Block Random Access Memory (RAM)

CCS Cache Compute System

CPI Cycles Per Instruction

CPU Central Processing Unit

CU Compute Unit

DDR Double Data Rate

DIMM Dual Inline Memory Module

DRAM Dynamic Random Access Memory

DSP Digital Signal Processing

DTB Dynamic Binary Translation

FPGA Field-programmable Gate Array

GPGPU General Purpose Graphics Processing Unit

IO Input/Output

IPC Instructions Per Cycle

IRQ Interrupt Request

ISA Instruction Set Architecture

KNN K-Nearest Neighbors

LLC Last Level Cache

LUT Look up Table

MAPE Mean Absolute Percentage Error

MIAOW Many-core Integrated Accelerator of Waterdeep/Wisconsin

MMIO Memory Mapped I/O

MMU Memory Management Unit

MOESI Modified, Owned, Exclusive, Shared, Invalid

MSHR Miss Status Holding Register

NDP Near Data Processing

NFA Nondeterministic Finite Automaton

OoO Out-of-Order

PCPI Pico Co-Processor Interface

PCU Processing-in-Memory (PIM)-Enabled-Instruction (PEI) Computation Unit

PEI PIM-Enabled-Instruction

PIM Processing-in-Memory

PI Programming Interface

PMU PEI Management Unit

RAM Random Access Memory

RTL Register Transfer Level

SDRAM Synchronous Dynamic Random-Access Memory

SIMD Single Instruction Multiple Data

SRAM Static Random Access Memory

SoC System on Chip

TLB Translation Lookaside Buffer

VHDL VHSIC Hardware Description Language

WB Write Buffer

1

Introduction

Contents

In 1965, Gordon Moore predicted that the number of transistors in a dense integrated circuit would double every year [3]. Until today, such observation is still valid, and it is known as Moore's law. Naturally, the increase of hardware resources that could be fit inside a processor's chip allowed to create complex and optimized architectures that delivered increasingly higher processing capabilities. On the other hand, the increase of memory devices' performance could not keep up with the evolution of the processing cores, a phenomenon usually known as the memory wall [4]. Therefore, memory subsystems became the main bottleneck of many architectures dealing with big data problems.

Data-driven applications, also called memory-bound, usually requires the processing of large amounts of data. Therefore, most of the time is spent transferring data between the cores and the memory subsystem, rendering useless the compute resources. To reduce the average memory access, thus mitigating this problem, modern processors feature multi-level cache systems. Notwithstanding the performance improvements that such structures allow (whenever temporal or spatial locality can be exploited under the hierarchy and limited cache sizes), the requirement of moving all the data from memory to the cores still imposes important performance and energy overheads.

Given the computational simplicity of some workloads, the question is raised on whether more efficient approaches can be employed to deal with such applications, or phases, which are bounded portions of the application that are data intensive. One possibility is to minimize the data transfers by performing computation near the operands actual location.

To mitigate the memory-wall problem, some solutions based on Near Data Processing (NDP) have been proposed over the years. The core idea of NDP is to move the computation near the data storage location, therefore reducing data movements between the memory subsystem and the Central Processing Unit (CPU) cores. To enable near-data computation, most solutions propose to modify the memory devices architecture, such that it allows computation, either by repurposing the existing resources or by adding new ones to the original memory architecture. Regardless of the method used to perform NDP, such solutions allow to take advantage of the memory high internal bandwidth, which makes it possible to operate over entire memory lines at once (up to several KBs) and therefore attain significant performance boosts in some kernels.

Although NDP solutions represented a significant contribution against the memory wall problem, previous memory technologies required substantial modifications to accommodate such systems, which were considered not viable. NDP only became popular when 3D-stacking was introduced in the process of fabricating memory devices, which enabled the possibility of connecting additional circuitry to the memory chips without having to modify their architecture. Moreover, most early NDP solutions opted to repurpose the existing hardware of the memory devices to perform computation, using a method called bit-line computation. Bit-line computation relies on the simultaneous activation of several rows of the memory and sensing the accumulated tension on the line, which can be translated into a logical operation

between the logical values of the activated cells. Therefore, bit-line computation only enables a few bitwise operations, limiting the functionality of the memory processing systems, and making them not suitable for most applications. Such solutions may also require particular data placement strategies, since operating over data scattered across different memory regions may be impossible (e.g., data split into two distinct memory arrays). Also, in general, these mechanisms struggle when dealing with coherence mechanisms across cache and memory hierarchy, which leads to performance degradation.

In this thesis, a new approach based on NDP targeting the execution of memory-bound kernels near-cache is proposed. Differently from the earlier NDP solutions, this architecture does not rely on bit-line computation, and it is meant to be tightly coupled with the cache system. The devised system supports a much wider range of operations than conventional NDP solutions, and it is also capable of executing the most common parallel patterns such as map and reduce operations. An operation is a map if it performs a given function over all the elements of a data-set (e.g., increment all the elements of a vector); a reduce-type operation produces typically one result by combining all the elements of a data-set with an associative function (e.g., sum all the elements of a vector). Although the proposed architecture can potentially be attached to any cache level, it is especially fit to be integrated with the Last Level Cache (LLC), to allow the main processor to concurrently operate over data on higher level caches. With such an approach, the Cache Compute System (CCS) can operate over all the data in a cache line (attaining high parallelism levels), while also relying on conventional coherence protocols among the different levels of the memory hierarchy, which makes the CCS model compatible with existing cache coherence protocols.

1.1 The inspiration for this work

The present thesis was originally inspired in a state-of-the-art General Purpose Graphics Processing Unit (GPGPU) architecture for Field-programmable Gate Array (FPGA) called Many-core Integrated Accelerator of Waterdeep/Wisconsin (MIAOW) [5]. Despite the novelty and contributions to the scientific community, the FPGA implementation of MIAOW (NEKO) featured relevant limitations related with the memory subsystem, the operation frequency (which was limited to 50 MHz), and also the hardware requirements, since a complete MIAOW Compute Unit (CU) could not fit in a Xilinx VC709 Virtex-7 Development Board. Efforts have been made by Pedro Duarte et al. [6] to improve the overall implementation (which was renamed SCRATCH), and tackle the memory subsystem limitations by introducing a prefetch memory to store the first memory section that contained the data to be processed by the GPGPU. Also, a tool-chain was created to trim the SCRATCH full implementation by analyzing the code and excluding unnecessary hardware, freeing resources and allowing to implement more CUs. Notwithstanding the improvements, SCRATCH is still limited by its operating frequency, as well as by the

memory subsystem, that turned out to be as inefficient as NEKO's when the required data is not present in the prefetch memory. The source code inherited from NEKO is poorly documented, making it hardly maintainable. Originally, this problem was tackled by placing a cache system with prefetching capabilities near the CUs, but the solution evolved to a system capable of performing computation *near-cache* that could be used with other processing systems. Also, due to the lack of resources in recent FPGAs to further modify its implementation, it was decided to abandon the MIAOW architecture, and design the novel CCS, inspired by one of the main bottlenecks of both NEKO and SCRATCH: the memory subsystem.

1.2 Objectives

The main goal of this thesis is to provide a CCS that enables parallel processing *near-cache*, compatible with the most common computation patterns, namely map and reduce. To accomplish this, the following objectives are proposed:

- Design and implementation of a stream-based architecture for a cache CU supporting common map and reduce patterns existing in applications;6
- Development of a simple framework to allow a developer to program and control the designed mechanism;
- Comparison and integration of the devised architecture with several processing systems;
- In-depth evaluation of the advantages of the proposed system in what concerns performance and energy efficiency.

1.3 Contributions

This manuscript proposes a novel CCS that exploits the high internal bandwidth and the data locality of the LLC to perform NDP. The use of the CCS for memory-bound applications avoids to transfer large amounts of data between the processor and the memory subsystem and also enables massive parallelism by operating over one cache line each cycle. The main contributions of the present work are:

- *Designed architecture:* The architecture of the CCS was conceived that implements 48 common arithmetic, shift, logic and move operations, is optimized for the processing patterns map and reduce, and is capable of processing one entire cache line simultaneously;
- *Model for simulation:* A model of the CCS was developed to be simulated alongside a realistic multi-level cache system and a recent processor using an architectural simulator;

- *Register Transfer Level (RTL) implementation:* A fully functional system was developed in VHSIC Hardware Description Language (VHDL) and implemented in a Xilinx VC709 Virtex-7 FPGA, featuring the advised CCS, an MB-Lite [7] and a memory subsystem, composed by a single-level direct mapped cache and a main memory built of FPGA Block Random Access Memory (RAM)s (BRAMs);
- *Programming and validation frameworks:* A C library of high-level functions to program and control the CCS from the application layer and a battery of unitary tests were developed;

The importance of the previously described contributions led to the publication of a paper in the 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2018):

- João Vieira, Paolo Ienne, Nuno Roma, Gabriel Falcao and Pedro Tomás, *"Exploiting Compute Caches for Memory Bound Vector Operations"*, on *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, September 2018.

1.4 Outline

The remaining of this thesis is organized in 5 chapters. In chapter 2, important concepts related to NDP are presented, and an overview of several state-of-the-art solutions is provided. Chapter 3 introduces the artifacts that supported the development of the work, as well as a summary of the available options. Then, in chapter 4 the projected solution is presented in detail, alongside the methods that were used to validate its functionality and evaluate its performance. The main experimental results obtained by comparing the devised solutions are shown in chapter 5, and the conclusions and future work are presented in chapter 6.

2

Near Data Processing

Contents

In a recent scalar core, on average, less than 1% of the consumed energy is expended by the Arithmetic and Logic Unit (ALU), while nearly one-fourth is spent moving data between the core and the memory subsystem; the remaining is spent in processing instructions in the core [2]. For memory-bound applications, where the CPU spends most of the time performing load and store operations, the relative amount of energy spent on data movements can be even higher. For that reason, approaches where the computations are performed close to the operands actual location are starting to emerge. NDP takes advantage of the memories high bandwidth and parallelism, therefore reducing both execution time and energy consumption. One sub-set of NDP is called Processing-in-Memory (PIM), which performs computation in-memory, frequently using the already available memory structures and requiring minimal changes to the architecture. This is only possible due to the DRAM internal structure, which allows using a method designated as bit-line computation.

To better understand how bit-line computation is performed, some knowledge on how the DRAM works is required. Therefore, the next section briefly explains the main stages of the DRAM operation, and how it supports bit-line computation.

2.1 DRAM background and bit-line computation

One Dual Inline Memory Module (DIMM) of DRAM [8–10] contains several dies, which encapsulate one or more memory arrays, as shown in Figure 2.1. A memory array is a grid composed of storage cells, which are capacitors charged to produce a logic '1' or a '0'. These capacitors are only able to keep their electrical charge for a short period because of the current leakage, therefore requiring periodic recharging. In a memory array, the storage cells of the same column are connected to the same bit-line through a transistor, and all the transistors in the same line are attached to a common word-line. When the voltage on a word-line goes high, all the transistors in that row act as closed switches, connecting the storage cells to the bit-lines. Then, the capacitors share their charge with the bit-lines, which increases or decreases their voltages. Sense amplifiers connected to each bit-line then detects the voltage deviations. Depending on the detected deviation, the sense amplifiers pull the voltage of the bit-line to a full swing of V_{DD} (the maximum voltage) or to zero, to represent the logic '1' and '0', respectively. This also allows recharging the capacitors while they are still connected to the bit-line through the respective transistors, as the sharing of their charges with the bit-lines destroys the data that was stored (read-destructive). In summary, the process of reading one line of a DRAM memory array is composed of four phases:

1. *Pre-charge:* Before activating the word-line which connects all the cells of a given row of the memory array to the bit-lines, the bit-lines are pre-charged to a reference voltage, $V_{ref} = V_{DD}/2$.

2. *Access:* When a voltage is applied to a word-line, the access transistors in that row close the circuit formed by the storage cells (capacitors) and the bit-lines, and the charge of each cell is shared with

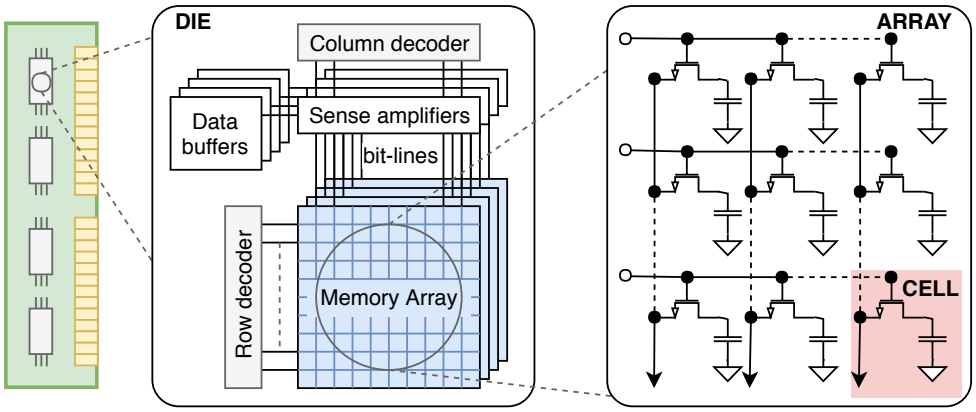


Figure 2.1: DRAM hierarchy. Each DIMM of DRAM usually has multiple dies, which are composed by one or more memory arrays. A memory array is a grid of storage cells that can be read by activating the correspondent word-line, which closes the circuit formed by the cell (capacitor) and its associated bit-line.

the respective bit-line. If the charge in the storage cell represents '1', the sharing process increases the voltage on the bit-line from V_{ref} to V_{ref}^{+}, whereas if it represents '0' the voltage on the bit-line will deviate to V_{ref}^{-}.

3. *Sense:* After the word-line is activated and the memory cells share their charge, the sense amplifier detects the deviations induced to the bit-lines. The time required to perform the sensing is called *sensing-time* or *sensing-delay*, and represents a significant portion of the total memory access time.

4. *Restore:* After the bit-lines are driven to the respective maximum or minimum voltage values, the selected word-line remains active, and the fully driven bit-lines voltages restore the charges in the capacitors through the access transistors. Meanwhile, the voltage value on the bit-line can be driven out of the sense amplifier circuit to provide the requested data. In this manner, the contents of a row can be accessed concurrently with the row restoration process.

In the conventional operation of a DRAM, only one word-line is activated during the *access* stage. If several rows are activated simultaneously, the voltage of a given bit-line is influenced by the several charges of the storage cells associated with the activated rows, and the sensed result will be the combination of the loads. Let us consider the case were two word-lines are activated at once and the charges of two storage cells associated with a given bit-line are combined. In the instant before the word-lines are activated, all the bit-lines are pre-charged with a given voltage V_i. Right after the word-lines are activated, the loads of two cells are shared with their associated bit-line, and three cases are possible:

1. *Both cells represent a logic '1':* The sensed voltage is given by $V_s > V_i + 2\delta$, where δ is the minimum deviation such that the sense amplifier can detect a logic '1';

2. *Both cells represent a logic '0':* The sensed voltage is given by $V_s > V_i - 2\delta'$, where δ' is the minimum deviation such that the sense amplifier can detect a logic '0';

3. *One cell represents a logic '1' and the other a logic '0':* The sensed voltage is given by $V_s \approx V_i$.

Considering a threshold of $V_{th}^{+} = V_i + 2\delta$, the sensed value can be translated into the result of the *and* operation between the two bits stored in the cells connected to the bit-line (if the detected value is higher than V_{th} the result of the operation is '1'). On the other hand, if the reference is $V_{th}' = V_i - 2\delta'$, the sensed value represents the result of the *nor* operation between the two bit operands (if the sensed value is lower than V_{th}' the result of the operation is '1').

It is also possible to activate more than two lines simultaneously, which allows to perform the *and* and the *nor* operations with even more operands. However, increasing the number of the simultaneously activated word-lines also increases the probability of data corruption. Moreover, due to the read-destructive nature of the DRAM, after concluding the bitwise operation, the operands are destroyed, i.e., the cells that contained the operands will end up with the result of the operation. Therefore, if the operands are to be maintained, they have to be previously copied.

Some recent works use bit-line computation to perform logic operations other than the elementary *and* and *nor*, such as *xor* and *inv* [11], or even carry-less multiplication or search [2]. Others, also integrate dedicated logic to perform even more complex operations, not doable using only bit-line processing, such as euclidean distance and dot product [1]. The following sections address some of these approaches and briefly explain how they operate.

2.2 In-memory and near-memory processing

PIM is often associated with bit-line computation, which is performed very fast (typically the necessary amount of time for one read and one write), however, it only supports a somewhat limited set of elementary logic operations. Furthermore, due to the read-destructive nature of DRAM, the operands are destroyed by the end of the computation. On the other hand, other NDP solutions, which do not rely on bit-line computation, can offer a more comprehensive set of operations, by making use of dedicated additional hardware.

As an example, Seshadri et al. [12] proposed a fast bulkwise *and* and *or* using bit-line computation. The principle of this mechanism lies in activating three word-lines simultaneously, making the deviation on each bit-line after the *sharing* phase to be towards the majority value of the three cells. Two out of the three activated rows are the operands, while the third row is for control (to determine whether the performed operation is an *and* or an *or*). Before performing the computation, the control row is homogeneously initialized with '0' or '1'. After the simultaneous activation of the three lines, eight different outcomes are possible, as shown in Table 2.1. If the control row is initialized with '0', the outcome of the *sense* stage is the result of the logic operation *and* of the operands, whereas if it is initialized with '1', the outcome is the result of the logic operation *or*. In general, the result of the logic operation performed in

memory is given by $R(A+B)+\overline{R}(AB)$, where R represents the control row and A and B the operands. The replacement of the operands with the result of the bitwise operation is solved by the authors by previously copying the source data to temporary rows using a customized and efficient mechanism of their creation, the RowClone [13]. As the computation is performed right after copying the operands to the temporary positions, the storage cells are fully refreshed in the *access* phase of the process, which is crucial, since the deviation on the bit-line caused by the simultaneous activation of the three lines is lower than when only one cell is connected. To support the devised system, the CPU is required to implement specific instructions. The mechanism was tested and compared with a system featuring an Intel Core-i7-4790K and 16 GB DDR3-1333, showing speedups as high as $10\times$ and energy consumption reductions up to $50.5\times$.

Under a different perspective, Ahn et al. [1] introduced the concept of PIM-Enabled-Instruction (PEI), which are instructions that are in the format to be executed by an existing PIM mechanism, but they are not necessarily issued to the memory. Their work relies on architectures with previous PIM support, therefore requiring minimal modifications to the compiler and the programming paradigm. The core idea is to create an equivalent PEI for each supported PIM instruction and replace the original instruction by the new one. When a PEI is issued by the host processor, there is a hardware mechanism that determines the best place to process it between memory and host processor. The software is entirely unaware of where the instruction is processed, which is entirely managed by hardware units. The devised mechanism is mainly composed of two types of hardware units: (1) The PEI Management Unit (PMU) and (2) the PEI Computation Units (PCUs). The PMU aggregates the main control logic of the system, and has the responsibilities of serializing the execution of PEIs (PIM Directory) and decide whether PEIs should be executed in-memory or in the host processor, by monitoring the locality of the operands (Locality Monitor). The PMUs are responsible for executing PEIs and are installed in the memory chip (for in-memory processing) and near the processing cores, as shown in Figure 2.2.

Table 2.1: Possible outcomes of the simultaneous activation of three DRAM word-lines. R represents the reference row that is homogeneously initialized with the logic '0' or '1' to control which bitwise operation is performed (*and* or *or*), and A and B designate the operands. To simplify the representation, the entries of the table contain the logical values represented by the actual charges contained in the storage cells and voltage sensed by the amplifier.

R	A	B	Sensed value	
'0'	'0'	'0'	'0'	**and**
'0'	'0'	'1'	'0'	
'0'	'1'	'0'	'0'	
'0'	'1'	'1'	'1'	
'1'	'0'	'0'	'0'	**or**
'1'	'0'	'1'	'1'	
'1'	'1'	'0'	'1'	
'1'	'1'	'1'	'1'	

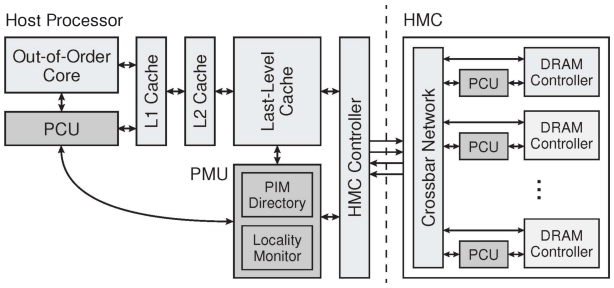
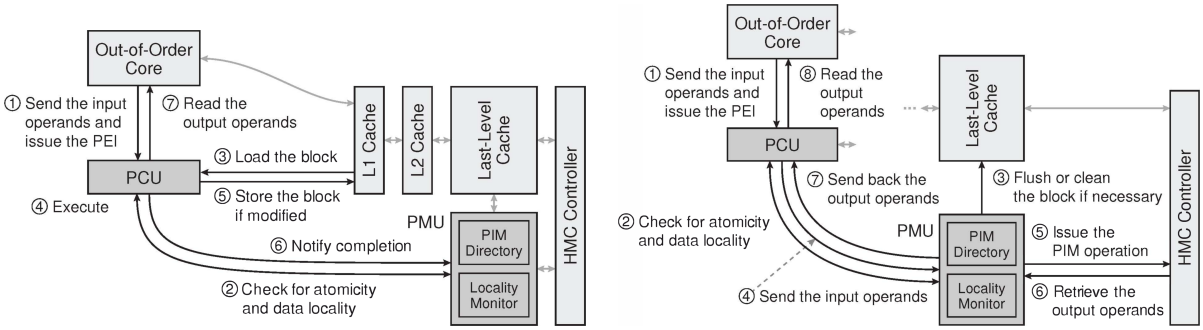


Figure 2.2: Overview of the system proposed by Ahn et al. [1]. The dark gray blocks identify the devised hardware modules for processing PEIs (PCUs) and for ensuring atomicity in the execution of the instructions as well as deciding where the PEIs should be processed (from [1]).



(a) Host-side PEI execution

(b) Memory-side PEI execution

Figure 2.3: Auxiliary diagrams to explain the execution of PEIs both in the processor-side and the memory-side (from [1]).

Depending on the locality of the operands (that is checked by the locality monitor inside the PMU), the PEI can be executed near the processing core (high data locality) or directly in memory (low data locality). Figures 2.3(a) and 2.3(b) show the PEIs execution flow when they are issued to the PCU near the processing core or the PCUs inside the memory device, respectively.

When a PEI has high data locality, the PMU will assign its execution to the PCU near to the host processing core (Figure 2.3(a)). First, the CPU sends the operands to the PCU and issues the instruction ①. Secondly, the host processor accesses the PIM Directory inside the PMU to obtain the lock of the cache block that contains the operands and consults the Locality Monitor to decide whether the PEI should be processed near the host processor or in-memory ②. Because the operands are closer to the processing core (in cache), the PMU advises the execution on the host-side PCU. Therefore, the host-side PCU allocates an operand buffer entry to store the operands, loads the cache block from the L1 cache ③ and executes the PEI ④. Meanwhile, if the PEI modifies the target cache block, a store request is issued ⑤. When the PEI ceases to execute, the PCU notifies both the host processor and the PMU, which maintains the consistency and ensures that the results of the PEIs are committed in the same order the PEIs where issued ⑥. Finally, the processor accesses the output operands through the

memory-mapped registers of the PCU and releases the buffer entry.

If the operands of a PEI are not present in cache, the PMU advises the execution on the memory-side (Figure 2.3(b)). The steps ① and ② are analogous to the case above. Then, simultaneously, the PMU informs the host-side PCU that the PEI is to be processed on the memory-side and sends a back-flush to the LLC to write-back any dirty data that the block may contain and invalidate its content for the next reading request ③. Also, at the same time, the host-side PCU sends the input operands stored in its mapped registers to the PMU ④. The operation type, the target block addresses, and the input operands are used to build a packet, following the memory device protocol for PIM operations, which is sent to the main memory ⑤. Upon completion, the memory device sends back a response packet ⑥, the PMU is notified and sends the output operands to the host-side PCU owned by the core that issued the PEI ⑦, and finally, the processor reads them through the PCU memory-mapped registers ⑧.

To solve data consistency and address translation problems, the authors impose an essential restriction, designated the *single-cache-block-restriction*, which dictates that a PEI can only access one single LLC cache block. While, according to the authors, this condition brings several advantages such as simplified locality profiling and address translation and essential implications at data localization and interoperability levels, it also limits the scope of operation, since the data has to be placed following a particular strategy to be processed by the devised system.

The authors projected this architecture to target applications with large memory footprints and large memory bandwidth consumption, often classified as *big-data* workloads. A set of applications from areas such as large-scale graph processing, in-memory data analysis, machine learning, and data mining was compiled to test to the system, rather than a standard battery of benchmarks. Therefore, the instructions implemented by the PCUs are useful for the considered applications. Table 2.2 presents the instructions supported by the PCUs as also the applications that benefit from each instruction.

For large inputs, it was observed that the conventional PIM-only system achieves 44% speedup over the ideal host processor since massive parallelism is achieved using the internal bandwidth of the memory.

Table 2.2: Instructions implemented by the PCUs in the architecture proposed by Ahn et al. [1].

Operation	Read	Write	Input	Output	Application(s)
8-byte integer increment	Yes	Yes	0 bytes	0 bytes	Average Teenage Follower
8-byte integer min	Yes	Yes	8 bytes	0 bytes	Breadth-First Search, Single-Source Shortest Path, Weakly Connected Components
Floating-point add	Yes	Yes	8 bytes	0 bytes	PageRank
Hash table probing	Yes	No	8 bytes	9 bytes	Hash Join
Histogram bin index	Yes	No	1 byte	16 bytes	Histogram, Radix Partitioning
Euclidean distance	Yes	No	64 bytes	4 bytes	Streamcluster
Dot product	Yes	No	32 bytes	8 bytes	Support Vector Machine Recursive Feature Elimination

However, for small inputs, the PIM-only system degrades the overall performance by 20% because it always performs the computation in the DRAM even though the data is present in cache. The devised system not only maintains the speedup for large inputs, using PIM, as also competes with the host-only system for small operands. On average, the overall system outperforms both the PIM-only and the host-only systems by 32% and 47%, respectively. Concerning area and energy consumption, negligible overheads were determined for both. More specifically, the memory-side PCUs contribute only 1.4% for the energy consumption of the main memory and about 1.85% for the die logic area.

2.3 In-cache and near-cache processing

Recently, some works suggest to move processing logic near the cache or even perform bit-line computation in-cache, rather than in the main memory. These works take advantage of the Static Random Access Memory (SRAM) technology to perform bit-line computation, similarly to the DRAM, achieving massive parallelism due to the memory high internal bandwidth and also taking advantage of the operand's locality.

2.3.1 Compute caches

Aga et al. [2] propose a cache computing architecture where the cache resources are re-purposed to perform active computation. In this work, bit-line computation is used, similarly to early PIM solutions for DRAM. Also, extensions to bit-line computation were added to enable support for additional operations, such as *copy*, *xor*, *equality comparison*, *search*, and *carryless multiplication*. In total, the devised architecture supports 9 operations, as listed in Table 2.3. The architecture is programmable using explicit routines supplied by the authors, therefore no compiler support is required. Transparently to the programmer, the architecture provides two different computing mechanisms: *in-cache* and near-cache. The *in-cache* computation mechanism can only take place when the operands are located in cache blocks within a certain physical region that the authors define as Block Partition (BP), illustrated in Figure 2.4. Otherwise, only *near-cache* computation is possible. The *near-cache* computation mechanism is composed of discrete logic and takes place at the memory controller level.

The computation *in/near-cache* can occur in any of the cache levels, depending on the locality of the operands. However, to simplify the design, the authors decided that the computation can only take place at the L1 or LLC. Figure 2.5 illustrates a working example of the Compute Cache where the computation takes place at the LLC. First, the core issues a Compute Cache operation to the L1 controller ①. As none of the operands is present in the L1 cache, the instruction is forwarded to the L2 ②. The L2 cache has a dirty copy of B and a clean copy of A, however, the computation cannot take place at L2, so B is written back to the LLC and the instruction is forwarded to L3 ③. The resulting address, C, is not cached

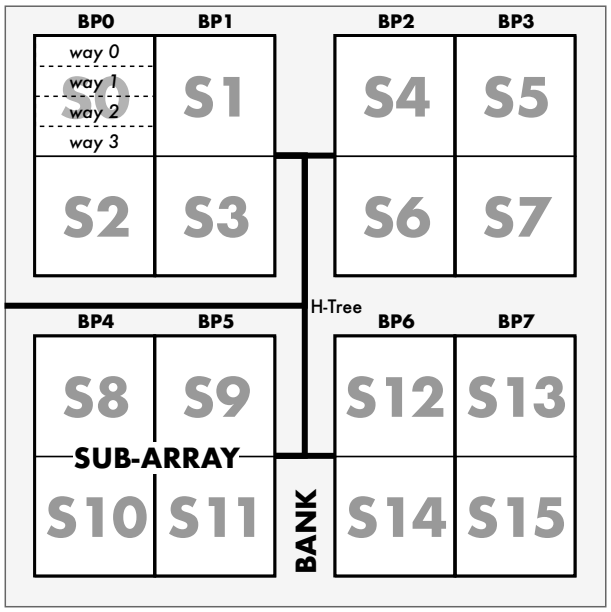


Figure 2.4: Cache hierarchy as described by Aga et al. [2]. $BP0 \dots BP7$ identify different Block Partition, whereas $S0 \dots S15$ represent different sets. *In-cache* computation can only take place when the operands are located in the same BP, otherwise only *near-cache* computation is possible.

in the LLC, so it is fetched from memory ④, the operation is performed ⑤, and the L1 controller is notified of its completion ⑥. Finally, the L1 controller notifies the core that the instruction has been executed ⑦.

The system was modeled using an architectural simulator (SniperSim [14]) and a power, area, and timing modeling framework (McPAT [15]) was used to model power consumption in the cores and caches. The base system for comparison supported 32-byte SIMD loads and stores. Results for a set of micro-benchmarks show that whereas the modifications to the cache result in an area overhead of 8%, the achieved speedup is as high as $54\times$, with an average energy saving of 91%. Also, a set of applications was adapted to use the Compute Cache. Performance improvements ranging from $1.5\times$ to $3.2\times$ where achieved and energy efficiency improvements of $2.7\times$, on average, were observed.

Table 2.3: Supported operations of the Compute Cache devised by Aga et al. [2].

Operation	Description
copy	$b[i] = a[i]$
zeroing	$a[i] = 0$
equality comparison	$r[i] = (a[i] == b[i])$
search	$r[i] = (a[i] == k)$
and	$c[i] = a[i] \& b[i]$
or	$c[i] = (a[i] \| \| b[i])$
xor	$c[i] = a[i] \oplus b[i]$
carryless multiplication	$c_i = \oplus(a[i] \& b[i])$
not	$b[i] = !(a[i])$

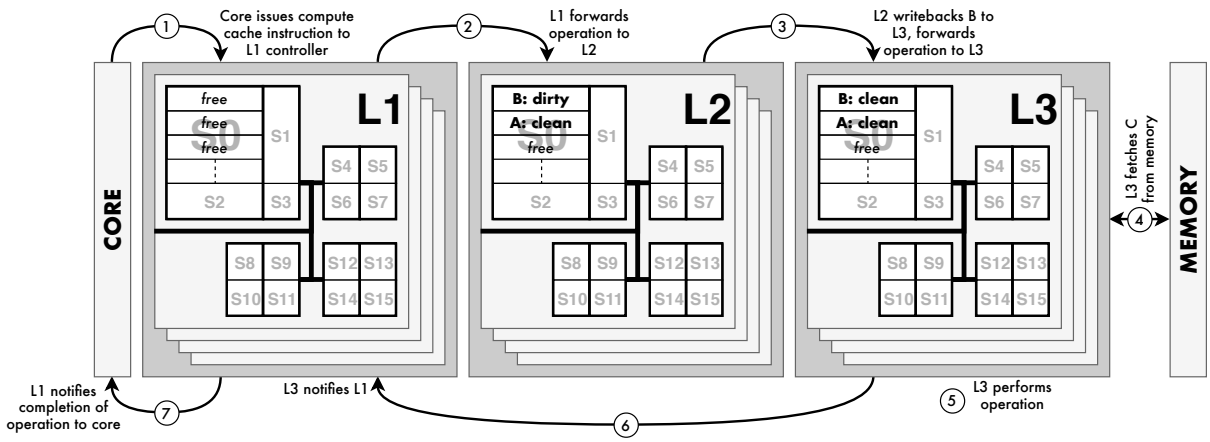


Figure 2.5: Operation example of the Compute Cache devised by Aga et al. [2] where the computation takes place at the LLC.

2.3.2 Cache automaton

Subramaniyan et al. [16] propose an application-specific architecture where the processing takes place *in-cache*. They suggest exploiting the last level caches to perform active computation optimized for the Nondeterministic Finite Automaton (NFA), which is a computational model widely used in data analytics, data mining, and network security, among others. Differently from traditional PIM approaches, including Micron's DRAM-based automata processor [17], the Cache Automaton relies on cutting-edge SRAM cache technology, which provides faster and more energy-efficient memories. The switch architecture, responsible for coordinating the access to the common buses, is redesigned to support the automata processing features, as well as the line multiplexing for sense-amplifying. Compiler support is also provided, which automates the process of mapping real-world NFAs to the Cache Automaton. The authors provide two different architectures, one optimized for performance and the other for space. They observe that the performance optimized system provides a speedup of $15\times$ over Micron's automata processor and $3840\times$ over a conventional x86 CPU core. The space optimized architecture provides a speedup of $9\times$ over Micron's automata processor, and reduces 40% the cache utilization, on average. Plus, the operational frequencies of the space-optimized and performance optimized architectures are $7.5\times$ (1.2 GHz) and $11\times$ (2 GHz) better than Micron's automata processor, respectively.

2.4 Summary

In this chapter, some previous work about NDP is presented. Two types of NDP are distinguished: (1) *in-memory* processing; and (2) *near-memory* processing. *In-memory* processing, is often associated with bit-line computation, whereas *near-memory* processing usually takes place in dedicated hardware units that are placed near the memory devices and still takes advantage of the memory internal high

bandwidth. Although both approaches attend high speedups due to the massive parallelism enabled by the memory internal bandwidth, and, in general, energy efficiency improvements, the implemented set of operations is rather limited and unsuitable for most real applications. Moreover, earlier proposals were difficult to be integrated into the memory chips, and therefore not cost effective. NDP only became popular when 3D-stacking started to be used in the process of fabricating memory devices, which allowed to integrate the logic circuity necessary to perform computation without having to modify the internal structure of the DRAM dies.

Recent proposals suggest to perform NDP at cache level, rather than in the main memory. Such approaches exploit not only the SRAM technology, which is faster than DRAM, but also the locality of the operands. Ahn et al. [1], for instance, propose to combine traditional PIM with cache level computation, by creating a locality-aware system that computes the PIM instructions *near-cache* or *near-memory*, depending on the locality of the operands. On the other hand, Aga et al. [2] and Subramaniyan et al. [16] devise cache-only processing architectures, which offer solutions for generic and application-specific processing, respectively.

3

Implementation Technologies

Contents

To reduce the overhead of transferring large amounts of data between the CPU and the memory subsystem, an architecture that enables computation *near-cache* is presented in the scope of this thesis. Throughout the development process of this system, several technologies were used, namely an architectural simulator was used to simulate a simple system featuring the devised processing structure, validating its behavior; a complete RTL model of the system was described and implemented in an FPGA, alongside a processing core and a memory subsystem. To decide the best technologies to use in the development process, several simulators and soft-CPUs were analyzed and compared. The following sections provide an overview of the main considered options to develop the devised solution.

3.1 Architectural simulators

To evaluate the performance of the mechanism proposed in this thesis, mainly to validate its behavioral when connected to modern super-scalar architectures (e.g., based on the Intel IA-32e–commonly known as x86–and ARM v7 Instruction Set Architectures (ISAs)) the use of an architectural simulator was required. The goal of an architectural simulator is to evaluate the behavior of an architecture and estimate its performance (by measuring clock cycles), particularly since architectural simulators do not feature area and power analysis by default. In the literature, architecture simulators are mainly divided into 2 big groups: cycle-accurate simulators and interval simulators. While cycle-accurate simulators emulate with detail each instruction running on the processor's hardware, interval simulators divide the program into phases and extrapolate the execution time for each phase based on a timing profile, which typically makes them much faster than cycle-accurate simulators.

Multi2Sim [18] is an open-source, cycle-accurate simulator targeting super-scalar, multi-threaded and multi-core x86 CPUs. The simulator has two main components: functional, that emulates the execution of a program in an x86 architecture; and architectural, that tracks the execution of each instruction in the processor hardware on a cycle-by-cycle basis. A variety of benchmarks is supported by the simulator without any porting effort, as also several architectures featuring many Out-of-Order (OoO) core super-scalar pipelines, memory systems with cache coherence protocols, interconnection buses, and additional devices.

MARSSx86 [19], similarly to Multi2Sim is a cycle-accurate architectural simulator. At emulation level, MARSSx86 uses QEMU [20], which is a framework that agglomerates several emulators, such as CPU, Memory Management Unit (MMU), Input/Output (IO) devices, and chipsets. On the top of it, at simulation level, PTLsim [21] is used, which is a cycle-accurate architectural simulator that models several OoO x86 CPUs. As it is a cycle-accurate simulator, it truly simulates each instruction of the ISA, by decomposing it in RISC-like μops (like many x86 architectures) and keeping track of each μop through the pipeline. PTLsim originally uses a modified Xen Virtual Machine hypervisor and had to be extensively modified

to use QEMU instead. Additionally, the authors also added functionalities to MARSSx86 that were not present in PTLsim, such as MMX support and a brand new framework to collect statistics on the simulated architecture.

In contrast with most simulators, gem5 supports a variety of different ISAs and CPU models, that are configured differently depending on the ISA that is used. At the same time, the framework was kept modular, and performance optimization is also a concern of the development community. The simulator resulted of a collaboration involving several universities, namely Wisconsin, Texas, Massachusetts Institute of Technology, and Michigan, and companies such as Hewlett-Packard, Advanced Micro Devices, Google, and ARM, and is under active development by the community. There is a substantial amount of documentation about gem5, and since 2011 the number of publications using this simulator already exceeded one thousand. Moreover, according to comparative studies and accuracy evaluation [22, 23], gem5 is among the best solutions in terms of accuracy, and the ARM model, in particular, is validated by ARM.

Another architectural simulator is called Sniper [14]. In contrast to Multi2Sim, Sniper is not a cycle-accurate simulator, but rather an interval simulator, driven by events. It differs from the previous simulators in the sense that it does not keep track of the instructions' execution through the CPU pipeline. Instead, it divides the streaming of instructions in intervals delimited by what the authors call *miss events*. *Miss events* are originated by all the factors that can cause an instruction to idle in the pipeline, such as branch mispredictions and cache and Translation Lookaside Buffer (TLB) misses. The simulation models for the branch predictor, the memory hierarchy, the cache coherence, and the buses determine the miss events, whereas the timing of the intervals is heuristically calculated based on analytical models. Therefore, different intervals can be analyzed independently, allowing to take advantage of multi-core processors by issuing multiple simulation threads. The analytical models to analyze the intervals in Sniper are supplied by Graphite [24], which offers 3 one-Instructions Per Cycle (IPC) CPU models.

Initially developed to evaluate ZCache [25], ZSim [26] outgrown its purpose and gave origin to a fast x86-64 architectural simulator. Its primary goal is to accurately simulate memory hierarchies and large, heterogeneous systems featuring hundreds to thousands of cores. It is therefore highly optimized with the simulation times scaling linearly with the core count. ZSim not only exploits parallelism but also uses a technique called Dynamic Binary Translation (DTB) to speedup the sequential phases of the simulation. DTB principle relies on modeling the execution of each instruction just once (i.e., assign a fixed latency to an instruction or a group of instructions by emulating it for a limited number of times), which is called instrumentation phase, and on each future execution of the instruction, the simulator just accounts for its previously calculated latency. Therefore, ZSim can be seen simultaneously as a cycle-accurate and an interval architectural simulator.

Comparative studies [22, 27] on the existing architectural simulators have been performed, showing

Mean Absolute Percentage Errors (MAPEs) below 40% for x86 architectures. However, Multi2Sim, MARSSx86, and Xsim are x86-only architectural simulators, not allowing to simulate other architectures. Moreover, the last contributions made to this simulators are not recent, which suggests that they are not actively maintained anymore. Sniper, on the other hand, is under active development, but because it heavily uses heuristics to determine the timing profile of the simulated systems, its accuracy is, sometimes, questionable (although the authors claim otherwise). In contrast, gem5 is a cycle-accurate, architecture-independent architectural simulator under active development, supported by many major processor manufacturers (e.g., ARM) and often used in top micro-architecture venues. Accordingly, gem5 was selected for the purpose of this work. Hence, in the next paragraphs, a more detailed explanation of gem5 is provided.

3.1.1 Gem5

The gem5 simulator [23,28] is a platform for computer architecture teaching and research. It is an open source project, and people are invited to push their own code to the project's repository to help improve and bring more functionalities to the simulator. Gem5 features a number of supported ISAs, including the x86 and ARM, and also several processor models, including OoO architectures. Nevertheless, the simulator is architecture independent, and a framework in C++ is provided to implement new architectures other than the ones that are supplied and maintained off-the-shelf. The simulations are event-driven, which means that the simulator engine reacts to scheduled events rather than emulating hardware parts. Specifically, when describing the timing behavior of a module, instead of keeping track of every event on each cycle, the full processing procedure is scheduled at once to be performed in an atomic manner after the number of cycles corresponding to its latency. Also, gem5 features a powerful framework to obtain statistics on the modeled architectures, allowing to define custom counters that are incremented when some stimulation methods of the components are executed.

When using gem5, there are mainly three types of files: (1) source files; (2) object files; and (3) script files. The source files contain the behavioral and timing descriptions of a component's architecture and are written in C++. They all inherit from the supper class *SimObject* and implement a set of methods required for the architecture to be simulated. Each time the architecture of a simulation object is changed, it has to be recompiled. On the other hand, the object files are Python classes that declare the interface exported by a simulation object. Optionally, the object file can parse parameters to the object's constructor to change the architecture dynamically, without the need for recompilation (e.g., change the capacity of a memory device dynamically). At the most top level, the script files, also written in Python, are responsible for describing the system to be simulated, by connecting the several components, specifying the configuration parameters, and the program to be run (in case the system features a processor).

The most top-level system, described in the script file, can be easily translated into a block diagram.

Listing 3.1: Python code of a gem5 simulation script that implements a system featuring a CPU, two cache levels and a main memory.

```
...
# Create an Out-of-Order CPU
system.cpu = DerivO3CPU()

# Create an L1 instruction and data cache
system.cpu.icache = L1ICache(opts)
system.cpu.dcache = L1DCache(opts)

# Connect the both L1 caches to the CPU
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)

# Create a memory bus (a coherent crossbar)
system.l2bus = L2XBar()

# Hook the L1 caches up to the l2bus
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)

# Create L2 cache and connect it to the l2bus
system.l2cache = L2Cache(opts)
system.l2cache.connectCPUSideBus(system.l2bus)

# Create a memory bus
system.membus = SystemXBar()

# Connect the L2 cache to the membus
system.l2cache.connectMemSideBus(system.membus)
...
# Connect the system up to the membus
system.system_port = system.membus.slave

# Create a DDR3 memory controller
system.mem_ctrl = DDR3_1600_8x8()
system.mem_ctrl.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.master
...
```

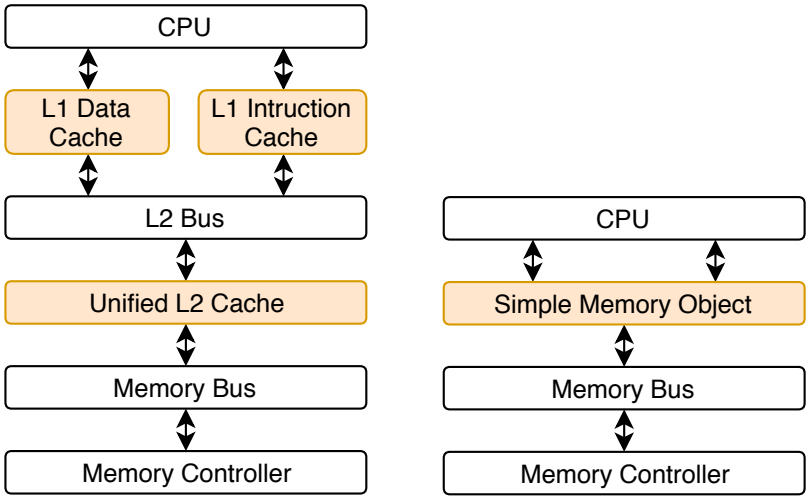
For instance, the code in the Listing 3.1, is equivalent to the block diagram shown in Figure 3.1(a).

At implementation level, each of the blocks presented in Figure 3.1(a) exports ports. These ports allow the components to communicate using packets, which are structures defined in the super-classes from which every simulation object inherits. Packets can be requests or responses and can also contain data (not necessarily a single word). To better understand how architectural components communicate using packets, let us consider a system even simpler than the one depicted in Figure 3.1(a), featuring a processor connected directly to the main memory through a stub that simply forwards the packets from the CPU to the main memory and vice-versa (see Figure 3.1(b)). The Simple Memory Object features

three interfaces, two to connect to the data and instruction ports of the processor and one to connect to the Memory Bus. In general, the object receives requests from the CPU side (read and write requests), and replies from the memory side (when data is retrieved). To implement memory objects, the framework requires a minimum set of functions to be implemented, namely those that deal with requests and that produce answers. Figure 3.2 shows the interface exported by the Simple Memory Object that allows it to be used as a memory component.

Each port can be classified as master or slave. In the example depicted in Figure 3.2, the *CPUSidePorts* are slave ports, whereas the *MemSidePort* is a master port. The slave ports implement a method that allows obtaining the address ranges of the devices that are connected directly or indirectly to those ports. The master ports implement methods to send requests and receive responses, while the slave ports have methods to receive requests and send responses. When an object invokes the method *sendFunctional*/*sendTimingReq* over its master interface, the corresponding method *recvFunctional*/*recvTimingReq* is executed on the slave port of the object connected to it, and a packet is transmitted. However, like in some real systems, there is no warranty that the slave is ready to receive the packet, and if that's the case, the master is notified. If a packet fails to be transmitted, the master should retry to send it until it is successfully received by the slave. The same applies when a slave sends a response to a master's request.

Besides *SimObjects* and *MemObjects*, gem5 also supports an alternative framework to model cache systems and coherence protocols designated Ruby.



(a) CPU connected to cached memory subsystem.

(b) CPU connected to the memory subsystem by a stub.

Figure 3.1: Figure 3.1(a) illustrates the block diagram of the system implemented by the script of the Listing 3.1, whereas Figure 3.1(b) depicts the block diagram of a system featuring a CPU and a main memory connected by a simple memory object whose only goal is forward the requests between the two entities.

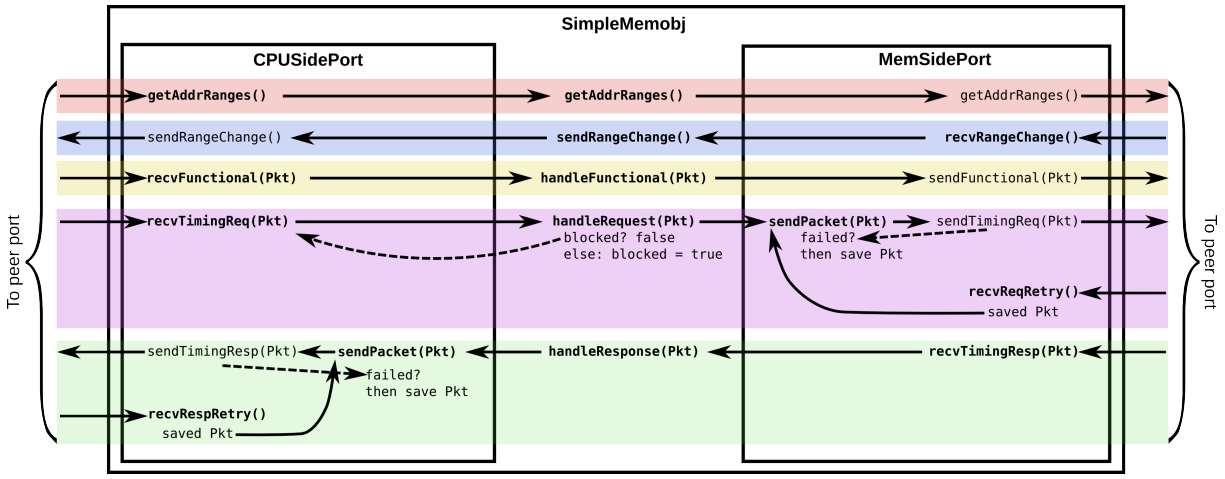


Figure 3.2: Main methods implemented by the Simple Memory Object that allow it to simply forward the packets from one side (memory or CPU) to the other.

Ruby is an engine integrated into the gem5 simulator that allows to model custom cache coherence protocols and works as a replacement to the classic gem5 Memory Objects. Gem5 offers two types of cache memory systems out-of-the-box, one using the classic Memory Objects and another using Ruby. Both of them are highly reconfigurable, and it is possible to configure the *clusivity* of each level of the cache and its replacement policy, among other parameters. However, Ruby caches do not allow as much customization as the classic Memory Objects, which makes it difficult to add more ports or to change some detail in the architecture. Furthermore, the documentation warns the developers that Ruby is a replacement for the classic memory system, and although there are interfaces between the classic gem5 MemObjects and Ruby, in general, the classic caches and Ruby are not compatible. Being difficult to modify the internal architecture of Ruby memory systems, and given the incompatibility with classic caches, it was decided not to use the Ruby engine, since to change some features in the cache system is a requirement to implement the mechanism devised by this thesis.

3.2 Soft-processors Overview

A processor, or a CPU, is responsible for executing the instructions that are part of a program. It contains all the necessary hardware to perform the basic arithmetic, logic, shift and move operations; reads operands from memory or memory mapped devices, performs computation over them and stores back results. A soft-processor is a processor that is implemented in soft-logic (in an FPGA). Soft-processors are not meant to compete with on-chip processors, however, because they are to be implemented in FPGA, their architecture is highly configurable, and can often be used as hosts of custom hardware accelerators, also implemented in the FPGA.

This section reviews several soft-processors, exploring their strengths and weaknesses.

Table 3.1: Available RISC-V ISA extensions and brief description of the included instructions.

Extension	Description
A	Atomic instructions
B	Bit manipulation instructions
C	Compressed instructions
D	Double-precision floating-point instructions
E	Embedded applications, resource-constrained subset
F	Single-precision floating-point instructions
G	General (I+M+A+F+D)
I	Integer base instructions
J	Dynamically translated languages
L	Decimal floating-point instructions
M	Integer multiplication and integer division instructions
N	User-level interruption instructions
P	Packed-SIMD instructions
Q	Quad-precision floating-point instructions
T	Transactional memory instructions
V	Vector operations instructions

3.2.1 RISC-V compatible architectures

RISC-V [29] is an ISA originally born in University of California at Berkeley which became an open standard collaboration for supporting computer architecture research and education. By providing a small base (but complete) set of instructions, it is suitable for direct hardware implementations, avoiding *over-architecting* particular micro-architectures, while it also supports a variety of ISA extensions that enable instructions not included in the base set (see Table 3.1). Also, there are two versions of the ISA, RV32 and RV64, targeting 32-bit and 64-bit processor implementations, respectively.

A Open-Source Micro-Architectures

Also born in Berkeley, Rocket [30] was the very first implementation of RISC-V, and integrates the lowRISC platform. The architecture is highly customizable through the Rocket Chip tool, which provides an automatic way of creating multi-core systems to be implemented on System on Chip (SoC). Rocket supports RV32 and the extensions I, M, and A, offering, by default, no support for floating-point operations. The default architecture also features a one-level cache system. Although the designs produced by the Rocket Chip tool are in Verilog and C languages and can be implemented in FPGA, the implemented designs are, in general, limited to an operation frequency between 25 MHz and 100 MHz.

Differently from Rocket, VexRISC-V [31] is optimized for FPGA implementation. It achieves an operating frequency of 220 MHz and only requires approximately 2000 Look up Tables (LUTs) in a Xilinx Artix-7 Development Board. VexRISC-V is a 32-bit architecture only capable of operating over integers and supports the extensions I and M. It ships with optional instruction and data caches, as also an optional debug extension that allows eclipse debugging via a GDB openOCD JTAG connection. The architecture

was described using SpinalHDL, which is a high-level hardware description language that after compiled generates RTL code (VHDL or Verilog). Notwithstanding the positive aspects of VexRISC-V, after the SpinalHDL code is compiled, the output RTL code is not suitable to be optimized by hand, making it difficult to perform small adjustments that may be difficult to do at SpinalHDL level. Moreover, SpinalHDL seems to be a customized language not widely accepted by the community, since all the references found about such framework are related to projects of the same authors.

RV12 [32] is a single-core-only architecture described in Verilog that can be implemented either as a 32-bit or a 64-bit system, supporting only the I extension. Similarly to the previous architectures, RV12 offers no support for floating-point operations and has no cache by default.

SCR1 [33] is another implementation developed by Syntacore. It is a 32-bit-only architecture that optionally supports the M and C extensions. The authors claim that the implementation is optimized for both area and power consumption. However, there seems to be no public quantitative data supporting that.

Developed by OnChipUIS, mRISC-V [34] implements the RV32 ISA and the I extension, and does not support floating-point operations. Moreover, it is primarily designed for Application Specific Integrated Circuit (ASIC), therefore not optimized to be implemented in FPGAs.

Created by VectorBlox to serve as a host for the accelerators devised by the same company, ORCA [35, 36] is a 5-stage pipeline implementation of RISC-V for FPGA, as shown in Figure 3.3. The RV32I implementation of ORCA requires 541 Adaptive Logic Modules (ALMs) in an Altera Stratix V (and 1623 LUTs in an Altera Cyclone IV) and operates at a maximum frequency of 244 MHz (or 109 MHz in an Altera Cyclone IV). Although ORCA's original implementation is written in VHDL and optimized for Xilinx, the soft-processor was also implemented in a LATTICE iCE40 ULTRA with an operating frequency of 22 MHz. ORCA also has optional support for hardware multiplication, which reduces its maximum operating frequency to 125 MHz in an Altera Cyclone IV and increases the number of LUTs in 45%. Similarly to the previous architectures, no hardware floating-point support is offered. Thus all floating-point operations are unrolled by the compiler and handled by software (soft-floating-point). As the complete architecture fully supports the RV32 with the extensions I and M, the official RISC-V tool-chain can be used to compile applications to run in the processor. Moreover, by exporting an Avalon/Wishbone standard interface, generic memory devices and peripherals can easily be connected to the core (including a generic cache system).

PicoRV32 [37] provides, by far, the smallest RISC-V implementation, requiring only 761 LUTs in a Xilinx 7-Series FPGA and operating at a maximum frequency of 714 MHz in a Xilinx Virtex UltraScale+ equipped with the device xcvu3p-ffvc1517-3-e. However, the base architecture (designated *small*, by the authors) does not provide counter instructions, two-stage shifts, or misaligned memory accesses and illegal instructions catching. Two more configurations are allowed in PicoRV32: *regular* (which

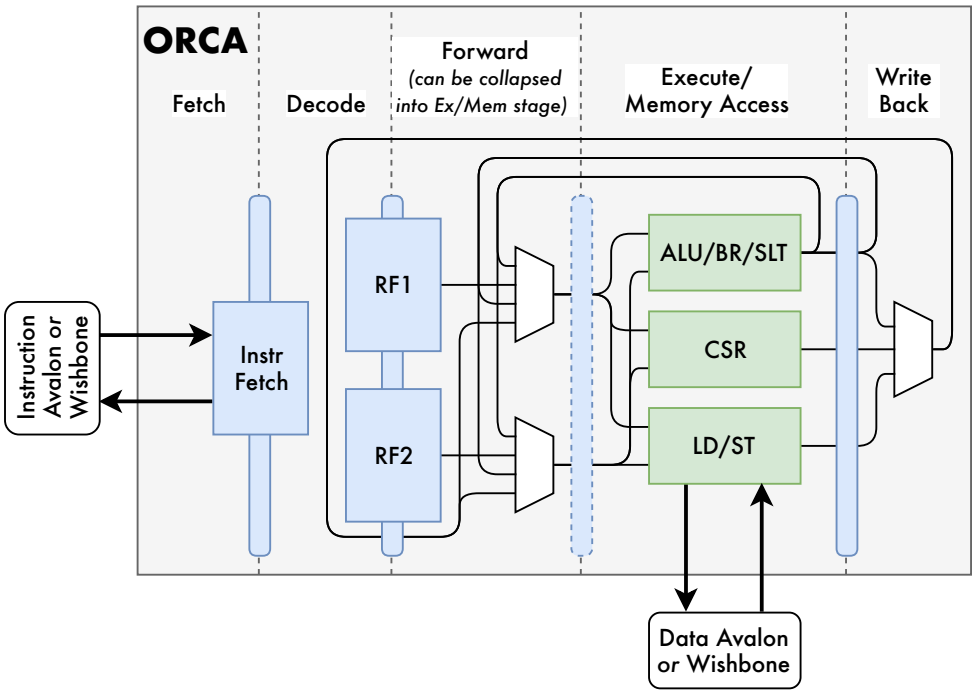


Figure 3.3: ORCA data path. The architecture is composed by 5 pipeline stages: *Fetch*, *Decode*, *Forward*, *Execute/Memory access*, and *Write Back*, where the *Forward* and the *Execute/Memory access* can be collapsed into one single stage.

requires 917 LUTs for Xilinx 7-Series) and *large* (requiring 2019 LUTs). The *regular* configuration enables the options that are disabled in the minimalistic design but does not implement any additional features. On the other hand, the *large* implementation enables all the additional features, such as Pico Co-Processor Interface (PCPI) (which allows implementing non-branching instructions in external cores), Interrupt Request (IRQ) (to react to external events, implement fault handlers, or catch instructions from a larger ISA and emulate them in software), MUL and DIV, the barrel shifter, and the C extension. Depending on the version, PicoRV32 supports the extension E, I, or I, M and C, respectively. Similarly to ORCA, PicoRV32 is also optimized for Xilinx. Nevertheless, it is possible to implement the architecture in FPGAs of different providers. Floating-point operations are, such as in the previous architectures, implemented by software, as there are no hardware floating-point units. The FPGA implementation of the processor exports a custom memory interface, with the possibility of conversion to Advanced eXtensible Interface (AXI), which allows to map peripherals and connect the memory subsystem, including a cache hierarchy.

Another implementation of RISC-V was developed in ETH Zurich and called PULPino [38,39]. PULPino has two main implementations, RISCY and zero-riscy, although a minimalistic implementation with limited support (μ-riscy) is also supported.

RISCY (illustrated in Figure 3.4(a)) is an in-order, single-issue, 4 stage pipeline core that implements RV32 with the extensions I, C and M. Optionally, it also offers hardware floating-point support, being

the only RISC-V core that possesses such a feature. Additionally, it implements a custom extension to the RISC-V ISA designated *Xpulp* that enables hardware loops, post-incrementing loads and stores, bit-manipulation instructions, multiply and accumulate operations, support for fixed-point operations, packed Single Instruction Multiple Data (SIMD) instructions, and dot product. While the IPC coefficient of RISCY is close to 1, the architecture was designed towards energy efficiency in ultra-low-power signal processing applications.

Zero-riscy, illustrated in Figure 3.4(b) only features 2 pipeline stages and implements the extensions I and C. Optional support for integer multiplication on hardware (extension M) is provided. Alternatively, zero-riscy can implement the extension E instead of the extension I, enabling only half of the processor registers and a subset of the instructions provided by the extension I. This minimal architecture has been designed for both ultra-low-power and ultra-low-area constraints.

PULPino offers a wide range of interfaces to allow communication with peripherals and other devices, such as I2S, I2C, SPI, and UART, as shown in Figure 3.5. Three distinct memory devices are connected to the processing core: the *Boot ROM*, used to boot the system in standalone mode, in which case the program is loaded from an external SPI flash into the *Instruction memory*; the *Instruction memory* which can also be pre-initialized, through the JTAG or SPI interfaces; and the *Data memory*, which plays the rule of main memory and contains the data to be processed.

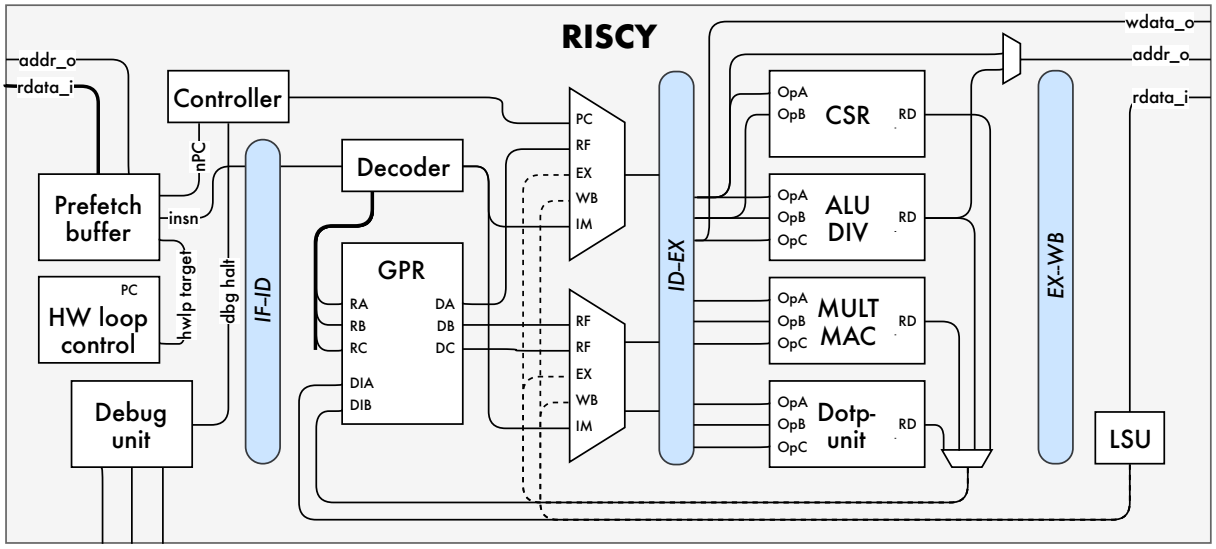
B Final considerations about RISC-V compatible architectures

Rocket, VexRISC-V, RV12, SRC1, and mRISC-V have in common the lack of documentation, which makes these soft-cores unattractive to be adopted as host processor.

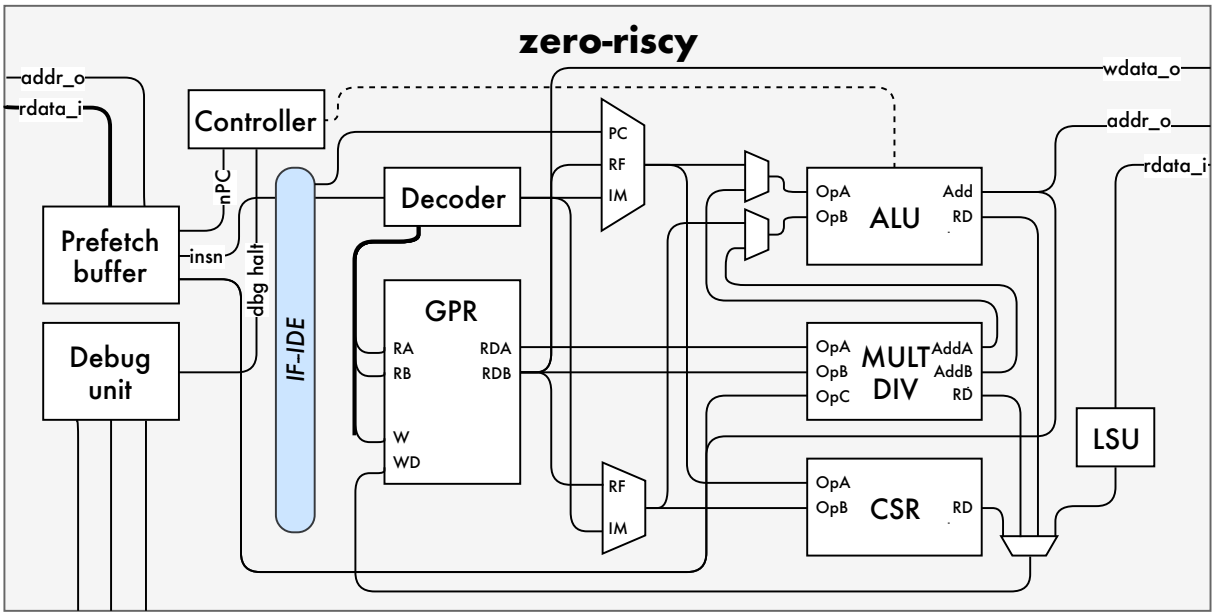
Although practical information on how to use PicoRV32 to integrate customized embedded systems is provided, almost no details about the internal architecture of the processor are documented. Therefore, to understand the interior design of the core, reverse-engineering on the source code to extract its schematics would be required.

ORCA, on the other hand, provides the full schematics of the processor, and also some details on how to implement it alongside a system. However, a base project is not delivered, and the process of implementing ORCA is not straightforward.

Despite not being optimized for FPGA as ORCA and PicoRV (because it targets primarily ASICs), the authors claim that PULPino operates with a frequency of 185 MHz on an FPGA (although, it was proven later that PULPino's operating frequency is limited to 40 MHz). Moreover, it is the only architecture that offers hardware floating-point support, among other advanced instructions. PULPino is an actual and actively maintained architecture, and there is detailed documentation about it. Also, a complete artifact including source files, example projects, and compiler support is provided, which eases the process of getting started with the soft-core and use it as micro-controller/host processor in other projects. Therefore,



(a) RISCY architecture



(b) Zero-riscy architecture

Figure 3.4: Main implementations of PULPino: RISCY and zero-riscy. The RISCY implementation features a 5 pipeline architecture and supports the extensions I, C and M of the RISC-V ISA. It also implements a custom extension called *Xpulp*, which enables hardware loops and additional operations such as dot product. Optionally, it also offers hardware support for floating-point operations (F extension). Zero-riscy only has two pipeline stages and has no hardware support for floating-point operations. It also supports the extensions I, C and M.

PULPino was first adopted to serve as host processor in the work devised by this thesis. A summarized comparison of all the cores is shown in appendix A.

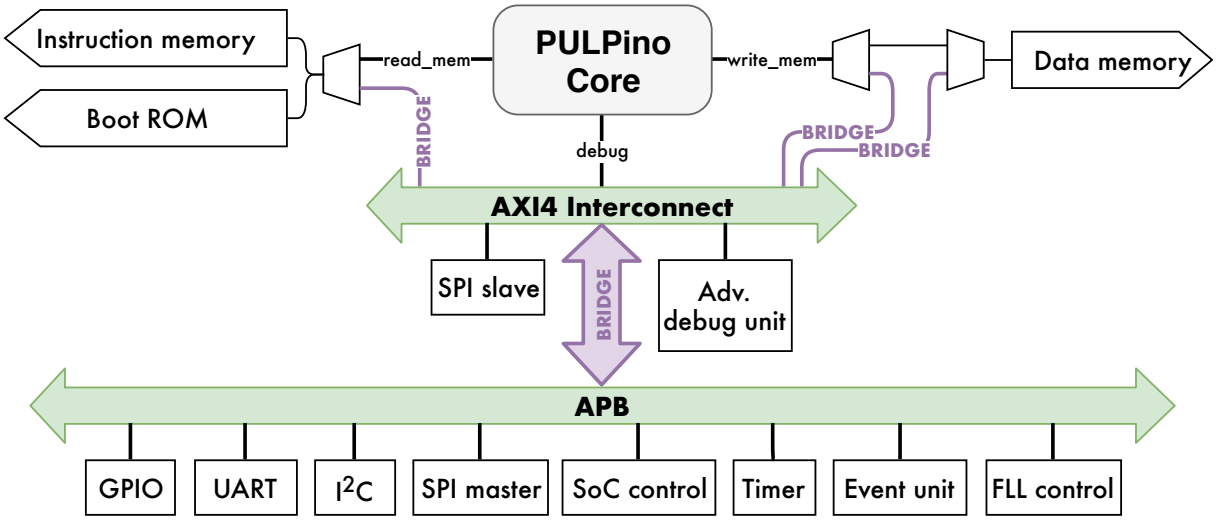


Figure 3.5: System featuring PULPino (RISCY or zero-riscy core), a simple memory subsystem and several peripherals that allow the core to communicate with external devices.

However, when the conclusion came that PULPino was limited to operate at a maximum frequency of 40 MHz on a Xilinx ZYNQ-7000 ZedBoard Development Board, there was the need to find an alternative soft-core.

3.2.2 MB-LITE

MB-LITE [7] is an in-order, single-core-only, open soft-processor originally created in TU Delft. It implements the MicroBlaze ISA, a proprietary soft-core provided by Xilinx, and its architecture is identical to the MIPS architecture of the early 1980s. Similarly to MIPS, MB-LITE has 5 pipeline stages: *Fetch*, *Decode*, *Execute*, *Memory Access*, and *Write Back*; and separated memories for instructions and data (harvard architecture), as shown in Figure 3.6.

The soft-core was designed to have the control as much distributed as possible, which eliminates the need for centralized units to deal with hazards and data forwarding. Therefore, all the dependencies that generate data hazards are solved locally in each of the different stages, either by forwarding the data to the *Execute* stage, when possible, or stalling the pipeline (e.g., when the result of a load instruction is immediately used, forwarding techniques cannot be applied). The control hazards created by branch and jump instructions are solved by flushing the pipeline when the condition is true, after predicting branch not taken.

Although the basic architecture does not feature an integer multiplier or a barrel shifter, the ALU can be extended to support these instructions, and the decoding logic at the *Decode* stage is already prepared to generate the control signals accordingly.

The core exports two interfaces to connect the instruction and data memories. While the interface

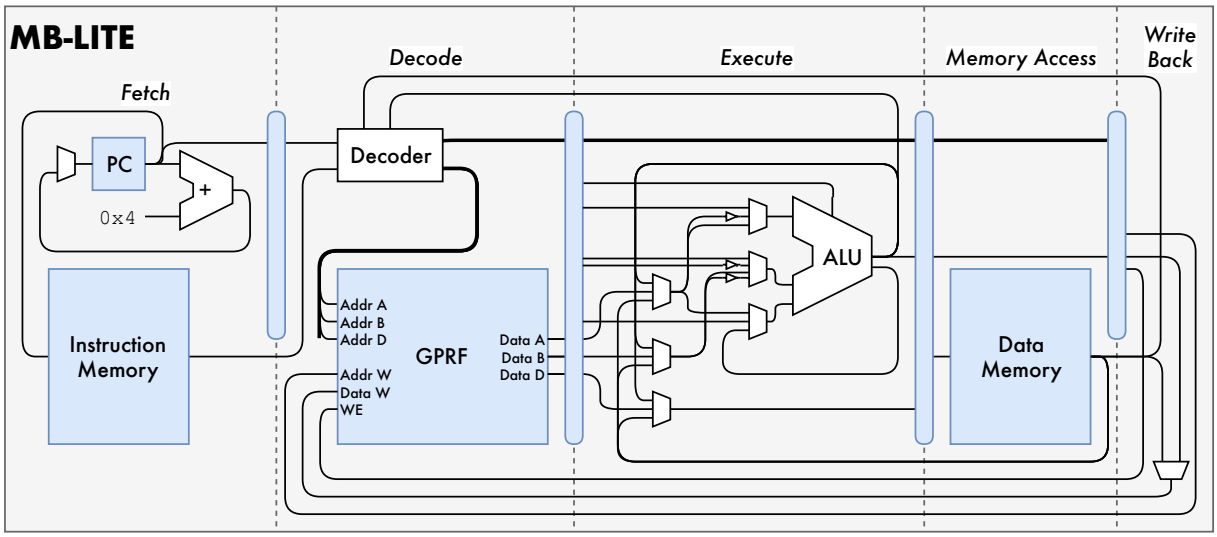


Figure 3.6: Simplified schematics of MB-LITE's architecture (focusing the data path). MB-LITE has 5 pipeline stages: *Fetch*, *Decode*, *Execute*, *Memory Access*, and *Write Back*. Because it is an harvard architecture, it has separated memories for instructions and data.

with the instruction memory is fixed, the data interface is configurable and implements a simple protocol, allowing to connect memory devices and peripherals and map them into the processor address space. The data interface is simply composed by an address bus, an input data bus, a byte-write-enable, an enable, an output data bus and a signal that represents the readiness of the interface. The data memory can be connected to the processor directly through the data interface (if there are no peripherals), or an address decoder can be added in between. Since there is no support for virtual memory, the only function of the address decoder is to forward the memory requests from the processor to the device (and the responses back to the processor) that is mapped in the memory region that includes the selected address, not increasing the device's latency. Because of the readiness bit that is included in the protocol, it is possible to connect to the core devices with response latencies superior to 1 cycle. Until the device responds to the request, the readiness bit of the interface becomes 0, and the processor idles, waiting for the memory interface to be ready.

When implemented in a Xilinx Virtex-5 Development Board, the base processor (without integer multiplier and barrel-shifter) runs at a frequency of 227 MHz. At the time it was released, MB-LITE was, in average, the fastest, the second smallest (regarding hardware requirements), and the second with the highest operation frequency, comparing with the existing alternatives: aeMB, MicroBlaze, OpenFire, OpenRISC, and LEON3.

MB-LITE differs from the other analyzed solutions in the sense that its implementation is clear and well organized, while at the same time it can achieve high operating frequencies without compromising the Cycles Per Instruction (CPI). Because the processor implements the MicroBlaze ISA (except some instructions that can be replaced by equivalent routines), the compiler provided by Xilinx in their software

kit, *mb-gcc*, can be used to compile programs to run in the processor. Therefore, MB-LITE was chosen after PULPino to serve as host processor in a system featuring the architecture devised by this thesis, and a full VHDL implementation featuring an MB-LITE core, a memory subsystem composed by a main memory and a single-level cache system, and the proposed solution was developed.

3.3 Summary

Throughout this chapter, the implementation technologies used to develop the work devised by this thesis were presented.

To verify the correct behavior of the architecture and integrate it with a simple processing core and a memory subsystem, an architectural simulator was used. For this, several simulators were analyzed and compared, and gem5 was chosen as the most suitable solution.

Then, a complete RTL model of the system was developed, to evaluate both area requirements and impact on the critical path. This required the use of a soft-processor to serve as host for the devised mechanism. Several implementations of RISC-V were analyzed, and PULPino was first chosen to act as main core of the system. However, after some efforts to implement PULPino in an FPGA, it was observed that its operation frequency was limited to 40 MHz (rather than 185 MHz), which caused it to be abandoned and MB-LITE to be adopted as main processing core. Despite this issue, given that some of the work necessary for integration with PULPino had already been done, an evaluation with this system is also performed.

4

Cache Compute System

Contents

The motivation for creating a novel architecture to perform computation near the cache had several sources. On the one hand, the current MIAOW implementation holds an inefficient memory subsystem, a low operating frequency, and has limited support, making it hardly maintainable. An initial goal was to mitigate the inefficiencies of the memory subsystem, through a high-efficient cache system, coupled with limited compute capabilities to implement atomic operations. However, in general, nowadays compute systems often struggle when dealing with big data-sets, as the memory subsystem is not able to feed the data to the cores at the rate it is consumed, thus rendering useless the CPU resources.

To tackle the last issue, interesting solutions have been proposed by exploiting PIM. However, they struggle when the data is split across different memory banks, which requires transferring the operands to the memory controller, degrading the overall performance of the mechanism. Moreover, such solutions must enforce coherence between cache and memory DIMMs before initiating PIM. On the other hand, NDP at cache level allows exploiting the operands locality, a very common characteristic among memory-bound kernels, as also long cache lines and faster technology (SRAM), when compared with DRAM. By operating near the LLC, the cost of a cache miss is reduced, since the latency to main memory is lower than in any other cache level. Moreover, by operating at the LLC, it is possible to take advantage of existing coherence protocols across multiple cache levels to guarantee correctness.

Furthermore, PIM solutions often rely on bit-line computation. Although repurposing the resources of the memory to perform active processing leads to a negligible hardware overhead and high efficiency improvements, bit-line computation is rather limited, and only allows to perform a reduced set of logic operations. Therefore, PIM solutions have limited interest when considering real general-purpose applications. Moreover, due to the read-destructive nature of the DRAM, to keep the operands valid after the operation, they have to be copied to another location in memory. In contrast, the placement of dedicated hardware to perform computation *near-cache* allows performing a much larger set of operations, making it suitable for many applications.

The proposed architecture aims to perform active processing near the LLC, which enables the processor to offload some of the computation required by memory-bound applications by performing map and reduce instructions directly *near-cache*. As a result, significant performance and energy efficiency improvements can be achieved.

To validate the devised architecture behavior, a simulation model integrating it with an OoO ARM core (that could be easily replaced with an x86 core) and a multi-level coherent cache system was developed. Then, an RTL version of the CCS for implementation in FPGA was described using VHDL. However, due to the intricacies of RTL development and the limited resources of FPGAs, it was necessary to trim down and simplify the original architecture.

Throughout this chapter, details about the devised mechanism are provided, focusing on its internal architecture, which applications motivated the building of the supported ISA and how the CCS operates

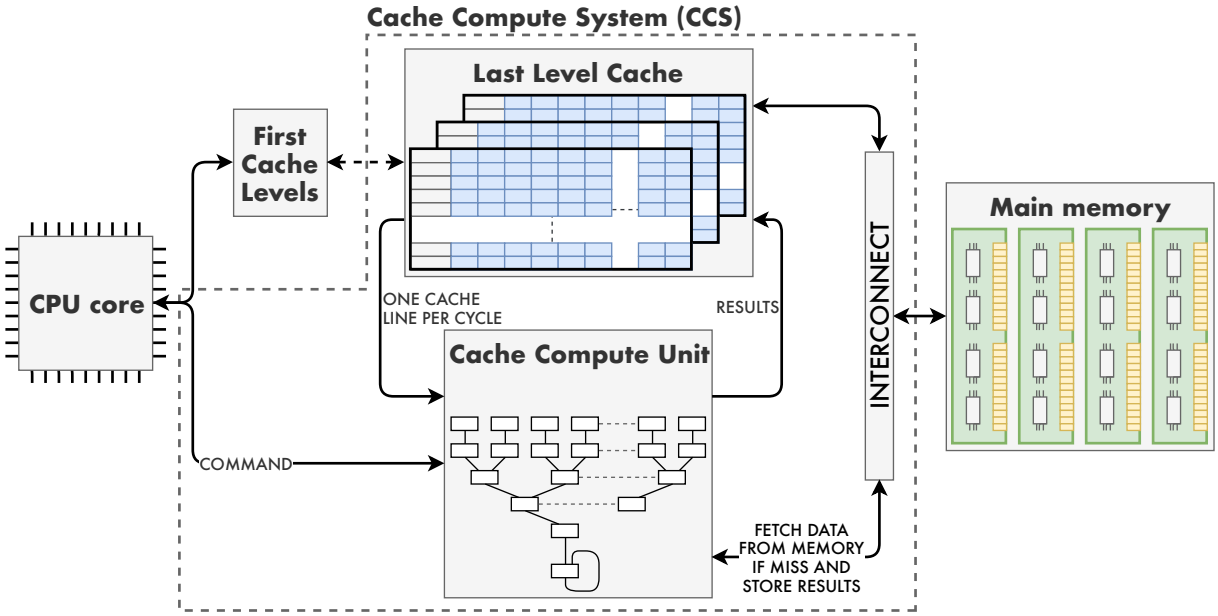


Figure 4.1: System overview integrating the processor, the CCS and the main memory. The CCS is defined as the logic block that includes the cache and the cache CU.

and interacts with the rest of the system. Also, the procedures used to validate and implement the architecture are explained.

4.1 CCS architecture

To tackle NDP at the LLC, the latter is enhanced with a cache CU, which operates at the level of cache controller. Due to physical proximity, the cache CU has a low access latency to a full cache line. Hence, the proposed Cache Compute System (CCS) can be seen as a logical block that includes a conventional LLC and the tightly interconnected cache CU, which agglomerates all the compute logic responsible to perform NDP, as shown in Figure 4.1. From the perspective of the CPU, the CCS behaves like an accelerator capable of performing a set of map and reduce commands while the CPU is executing. From the memory side, the CCS performs simply as a conventional data requester. As such, standard memory interfaces are used, and an interconnect module is responsible for arbitrating the access of the regular cache hierarchy and the cache CU to the main memory. Within the CCS, the LLC also has two memory interfaces: one to the processor and another to the cache CU. Therefore, while the cache CU loads operands or stores results in the LLC, other data can still be exchanged between the LLC and the upper levels of cache or the main memory.

To better understand the full design and operation of the devised architecture, the following subsections detail the commands implemented by the CCS and the structure of the cache CU that enables such

Table 4.1: Applications taken into consideration to build the CCS ISA.

Area	Algorithm	CCS commands (see Table 4.2)
Clustering	kNN	SSDVV
	k-Means	
	Mean-Shift	
	DBSCAN	
Machine Learning	CNN	IPVV
	Linear regression	ADDV, MULVV, SUBVV
	Linear Discriminant Analysis	ADDV, MULVV, SUBVV
Cryptography	Md5 Checksum	ANDVV, ORVV, XORVV, ROLVV
	Serpent	XORV, ROLVC
	Twofish	INITC, XORVC, SLLVV, ANDVV, XORVV
	Blowfish	XORV, INITC
	Cast-128/256	ADDVC, XORVV
Image/Video Processing	Convolutional filters	IPVV
	Pixel Matching	SADVV
Biomedicine	CAST	MAXV, INITC
Linear Algebra	Intel BLAS	COPYV, MULVC, ADDVV, IPVV, SSDVV, SADVV

features. Furthermore, the details on how the CCS fetches the operands and stores the results are also provided, together with how stridden operands are evaluated, and virtual addresses are translated by the CCS controller.

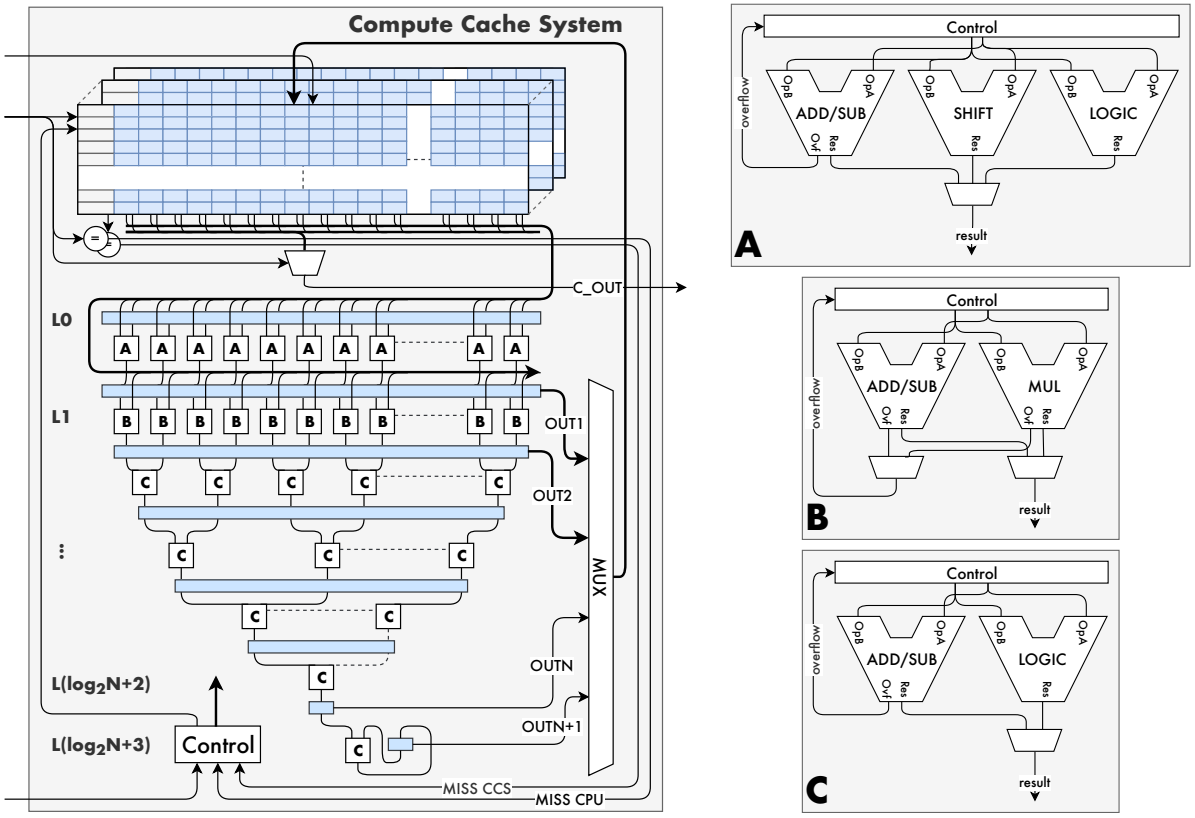
4.1.1 Supported operations

To decide which operations the cache CU should implement, several memory-bound applications or algorithms containing data-intensive phases were analyzed. These applications belong to different domains, namely clustering, machine learning, cryptography, image and video processing, biomedicine, and algebra. Table 4.1 illustrates some of the applications taken into consideration when planning the ISA to be supported by the cache CU, and also some of the CCS commands that were inspired by those applications. To enlarge the ISA of the CCS even further, some other common operations were added. Also, some other commands were added that incurred in negligible hardware overheads, namely all the shift-type commands parameterized by a vector operand. As a result, a set of 48 arithmetic, shift and logic vector operations was compiled to be implemented by the cache CU, as shown on Table 4.2. These operations were implemented in the form of commands that can be classified accordingly to different criteria:

1. *The mathematical operation:* arithmetic, logic, shift, or move;
2. *The number and type of operands:* two vectors (VOP2), a vector and a constant (VCOP), a single vector (VOP1), or only a constant (COP);
3. *The functional operation:* map or reduce.

Table 4.2: Commands supported by the proposed CCS.

		Instr	Description	
Arithmetic	VOP2	ADDVV	$r[i] = a[i] + b[i]$	*map*
		SUBVV	$r[i] = a[i] - b[i]$	
		MULVV	$r[i] = a[i] \times b[i]$	
		SSDVV	$r = \sum_i (a[i] - b[i])^2$	*reduce*
		SADVV	$r = \sum_i \lvert a[i] - b[i] \rvert$	
		IPVV	$r = \sum_i a[i] \times b[i]$	
	VCOP	ADDVC	$r[i] = a[i] + k$	*map*
		SUBVC	$r[i] = a[i] - k$	
		MULVC	$r[i] = a[i] \times k$	
		LESSVC	$r[i] = 1$ when $a[i] < k; 0,$ otherwise	
		GRTRVC	$r[i] = 1$ when $a[i] > k; 0,$ otherwise	
		EQUVC	$r[i] = 1$ when $a[i] = k; 0,$ otherwise	
	VOP1	COMP2	$r[i] = -a[i]$	
		SQV	$r[i] = a[i]^2$	
		ABSV	$r[i] = \lvert a[i] \rvert$	
		ADDV	$r[i] = \sum_i a[i]$	*reduce*
		MAXV	$r[i] = \max(a)$	
		MINV	$r[i] = \min(a)$	
Shift	VOP2	SLLVV	$r[i] =$ sll$(a[i], b[i])$	*map*
		SRLVV	$r[i] =$ srl$(a[i], b[i])$	
		SLAVV	$r[i] =$ sla$(a[i], b[i])$	
		SRAVV	$r[i] =$ sra$(a[i], b[i])$	
		ROLVV	$r[i] =$ rol$(a[i], b[i])$	
		RORVV	$r[i] =$ ror$(a[i], b[i])$	
	VCOP	SLLVC	$r[i] =$ sll$(a[i], k)$	
		SRLVC	$r[i] =$ srl$(a[i], k)$	
		SLAVC	$r[i] =$ sla$(a[i], k)$	
		SRAVC	$r[i] =$ sra$(a[i], k)$	
		ROLVC	$r[i] =$ rol$(a[i], k)$	
		RORVC	$r[i] =$ ror$(a[i], k)$	
Logic	VOP2	ANDVV	$r[i] = a[i]$ and $b[i]$	
		NANDVV	$r[i] = a[i]$ nand $b[i]$	
		ORVV	$r[i] = a[i]$ or $b[i]$	
		NORVV	$r[i] = a[i]$ nor $b[i]$	
		XORVV	$r[i] = a[i]$ xor $b[i]$	
		XNORVV	$r[i] = a[i]$ xnor $b[i]$	
	VCOP	ANDVC	$r[i] = a[i]$ and k	
		NANDVC	$r[i] = a[i]$ nand k	
		ORVC	$r[i] = a[i]$ or k	
		NORVC	$r[i] = a[i]$ nor k	
		XORVC	$r[i] = a[i]$ xor k	
		XNORVC	$r[i] = a[i]$ xnor k	
	VOP1	NOTV	$r[i] =$ not $a[i]$	
		ANDV	$r[i] = a[0]$ and $\cdots$ and $a[B-1]$	*reduce*
		ORV	$r[i] = a[0]$ or $\cdots$ or $a[B-1]$	
		XORV	$r[i] = a[0]$ xor $\cdots$ xor $a[B-1]$	
Move	COP	INITC	$r[i] = k$	*map*
	VOP1	COPYV	$r[i] = a[i]$	



(a) Datapath of the CCS.

(b) Different types of units that compose the cache CU.

Figure 4.2: CCS data-path. The first two levels of the binary structure are meant for maps and the remaining structure acts as a reduction tree. Consequently, different levels are equipped with different units: type-A unit–implements an adder/subtracter, a shift unit and a logic unit; type-B unit–implements an adder/subtracter and an integer multiplier; type-C unit–implements an adder/subtracter and a trimmed logic unit, capable of performing a subset of the total logic instructions.

4.1.2 Data processing structure

The integration of the cache CU within the CCS is depicted in Figure 4.2. To allow the implementation of both map and reduce operations, the cache CU has two processing parts: the first part corresponds to the two first levels of the CU (L0 and L1), providing a parallel-input parallel-output capability required to implement the map operations. Two levels are needed for some map-reduction-type operations, such as the sum of square differences, which needs two consecutive map operations, difference and square, before the reduction. The second part of the processing structure adopts a binary-tree shaped structure, required to implement the reduce-type commands, where each pair of elements from the previous level is combined.

To satisfy all the commands supported by the proposed architecture, the first two levels of the cache CU have to support more operations. The units within the first level feature full logic units, and also an

adder/subtracter and a barrel shifter. The second level of the cache CU is the only one equipped with multipliers but does not feature a logic unit, or a barrel shifter. The remaining levels are only equipped with an adder and a reduced logic unit, that implements the *and*, *or*, *xor* and *not* operations.

Only four levels of the CCS pipeline produce output, depending on the command under execution: the first level produces the output for almost all map operations, except those which involve multiplication. In that case, the output is collected from the second level. The reductions output can be collected either in the last level of the CCS (if the accumulator is used) or in the second-last (if the size of the vector operands is smaller or equals the number of units in the first level of computation).

Hence, the execution time of the proposed compute structure mostly depends on the type of command being issued and on the locality of the operands. Reductions take at least a number of cycles equal to the number of pipeline stages, although the time to finish a cumulative reduction on multiple cache lines is longer. On the other hand, map-type commands take at most two clock cycles to complete.

Independently of the considered operation, the designed pipeline binary tree ensures the maximum computing throughput and allows more than one command to be simultaneously under execution (in different pipeline stages).

4.1.3 Operand fetch and result store

The CCS operates as a stream processor with the operands and result being described as a 1-D descriptor composed of base address, stride, and length. To execute a given operation, the CCS divides the operands into groups of N elements, where N corresponds to the vector unit length, which should be adjusted according to the size of the cache line. Naturally, since the first and last elements may not be aligned in the cache, specific masks are generated to guarantee a correct operation over the described arrays (see subsection 4.1.4).

When executing commands, the CCS loads the operands in a sequential fashion, as shown in Figure 4.3: for VOP2 commands, it first loads one operand, storing it in an input buffer, before proceeding to the second operand; for VOP1 and VCOP, where a single operand exists, it skips the second operand and iteratively fetches data only from a single data input stream. At each data fetch, the CCS makes use of the existing cache structures. Hence, it first checks for a hit on the cache, before issuing a cache line request to the main memory. However, when loading data from memory, the operands are directly forwarded to the cache CU and are not stored in the cache. This avoids cache pollution, by reducing conflicts between CCS operands and processor data.

To store the command output, the CCS again relies on the cache existing structures and policies for data allocation and write. Hence, for write-back write-allocate caches, data is always written on the cache, whereas for write-through write-not-allocate, the result is always stored on the memory, with the cache being updated if the corresponding line(s) already exist on the cache.

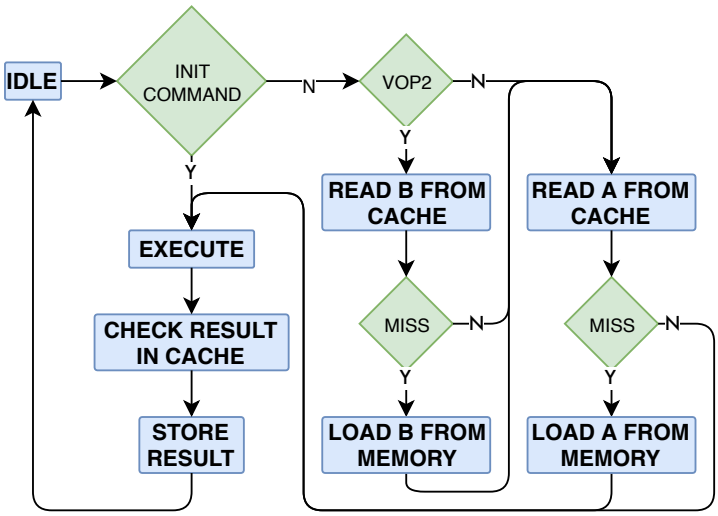


Figure 4.3: Execution flow graph of a command by the CCS, showing the different states of operation by the cache CU.

4.1.4 Mask generation and stride processing

When a command operates on vector operands, the stride functionality is enabled, i.e., the operands are allowed to be stridden, and the only elements considered for execution by the CCS are the multiples of the stride. As long as the stride is a power of two smaller than the size of the cache line, a mask can be specified in such a way that the only set indexes are the multiples of the stride. However, for all commands that involve multiple vectors (e.g., two vector operands or one vector operand and a vector result), all the elements have to be aligned in the cache for the command to succeed (i.e., all vectors have to have the same offset relatively to the beginning of the cache line). For striding and alignment correction purposes, two distinct masks are produced before the CCS command is executed. The boundary mask, generated directly by the CCS, is used to align the operands with the beginning and the end of a cache line and all the elements in the cache line out of the operand range are discarded. The stride mask is generated by software, and each one of its bits is set to one when its index is multiple of the stride. In the CCS, both masks are combined, generating the execution mask. The execution mask determines which elements of the operands will be considered when executing the command, and which results are to be stored in memory. For map commands, only the execution mask is needed; however, for reduction commands, a new sub-mask has to be generated at each level of the reduction three, determining which elements are to be reduced to the next level. Each sub-mask is generated by performing the logic *or* operation on the mask bits associated with the two operands of a given unit. Additionally, the instruction performed by each unit of the reduction tree is influenced by the bits of the mask associated with its operands. When both inputs are to be considered, the instruction performed by the unit corresponds to the one that is meant to be executed in that level (the one that is micro-programmed on the CCS controller), but in case

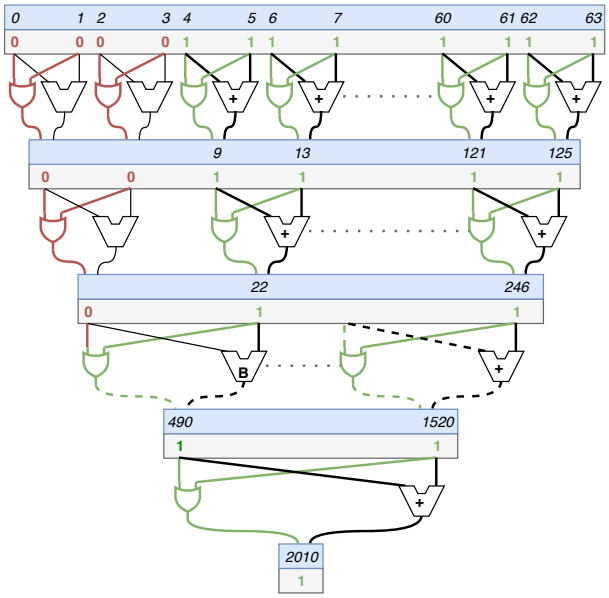


Figure 4.4: Sum of all elements of a misaligned vector with 60 elements and generation of the intermediary execution masks for the reduction tree from the main execution mask. The first 4 elements of the vectors are not valid, and therefore they are discarded.

only one of the operands is valid, the output of the unit will be that same operand. Figure 4.4 illustrates an example of all the elements of a vector being summed, and also the sub-masks of each level being generated from the sub-mask of the previous reduction level.

4.1.5 Virtual address translation

Usually, most recent processing systems are equipped with virtual memory subsystems, which enhances security and allows a virtual addressing space larger than the available physical memory. When the CPU performs a load or a store, the virtual address is first translated into a physical address by the TLB. Analogously, when programming the CCS, the processor writes in the programming interface the virtual addresses of the operands and the result, thus, before loading the data from memory and storing the result of the operation, the CCS requires these addresses to be translated into physical addresses.

Previous solutions suggest either to use the existing translation mechanisms of the processor [1] or to modify the way how virtual memory is managed, by reducing the associativity of the TLB and storing information in the main memory regarding address translation and page walking [40]. However, both approaches present eliminatory issues. When using the CPU translation mechanisms, the CCS has direct access to the physical addresses of the operands, which represents a security issue (e.g., other processes can obtain the physical addresses of data that does not belong to them by reading the CCS programming registers). Moreover, for operands scattered across different pages, this scheme fails, since the CCS is not able to find the following page without resourcing to the CPU TLB, or perform page

walking, if necessary. On the other hand, by changing the way of how the virtual memory is managed, the operating system running on the processor and all the hardware responsible for performing address translation and page walking would have to adapt to the new virtual memory management scheme. This represents a critical drawback.

In contrast, the CCS replicates the structures used by the processor to translate the virtual addresses, i.e., it has its own TLB, which is synchronized with the TLB of the processor issuing the command, and a page walker. Therefore, the mechanism does not store physical addresses, which enables security: it is not possible for an application to obtain the physical addresses of another application data. Moreover, it offers advantages over the early suggestions to tackle this problem: as it does not use the processors TLB, the number of translations to load an operand is not limited to 1 (the operand can be split between pages); the second TLB simply replicates the content of the processor's TLB, which means that no modifications to the virtual memory management mechanism are required.

4.2 CCS operation

The process of issuing a command to the CCS starts with configuring it with the command parameters. As the CCS behaves like a regular Memory Mapped I/O (MMIO), an address range specifies its programming registers. Therefore, to program it, the processor writes the parameters on the memory positions specified by those addresses, the cache hierarchy is bypassed, and the configurations are directly exchanged between the processor and the CCS.

Then, after the processor signals the CCS to start executing, the control unit checks the type of issued command. If the command is type VOP2, VOP1 or VCOP, the CCS switches to a reading state, where it checks if the operands are in cache. If one or more operand(s) is(are) not in cache, the CCS enters an idle state while they are fetched from the main memory to an input buffer. However, if the command type is COP, the CCS changes its state directly from idle to execute, since the only operand is the constant passed by the CPU. When it reaches the execution state, the CCS will keep executing for the predetermined number of cycles that a given command requires to complete.

When finishing the execution of the command, the CCS stores the results of the operation, using the existing memory mechanisms for that (e.g., if the cache implements the write-back policy, the results are written only in cache; however, if the write-through policy is implemented, the data is also written in the main memory, possibly making use of a write buffer), before returning to the idle state and notifying the processor about its readiness, by writing to the corresponding interface register.

Independently of the considered commands, the length of the operands is not limited by the number of functional units of the first level. Instead, larger operands can be specified, and hardware loops will take care of processing them in parts until they are all consumed. Depending on the type of the command,

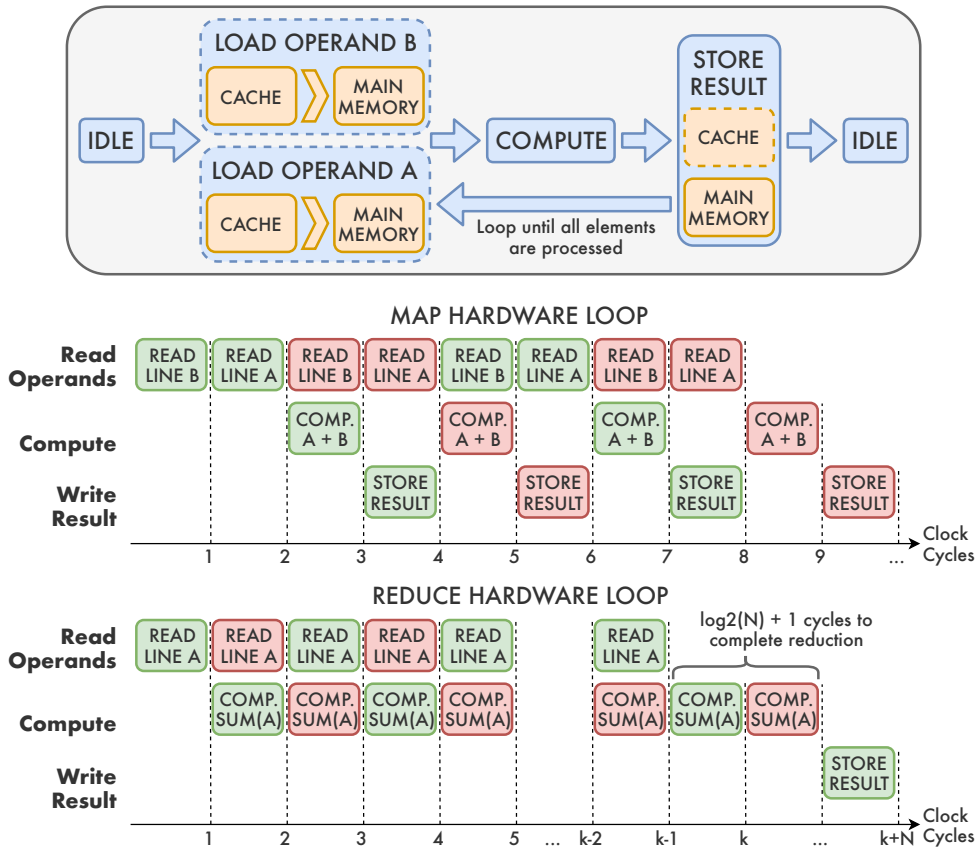


Figure 4.5: Finite state machine that describes the control flow of the cache CU. The examples below describe the flow of the hardware loops. For map commands, the partitions of the operands are fully processed, and the results are stored for each iteration. Differently, for reduce commands, the operands are fully loaded, and the execution happens in pipeline.

these loops are executed differently: for map-type commands, each partition of the operands is processed independently, and the results are stored while the next partition is fetched; on the other hand, for reduce-type commands, the operands are entirely fetched, and the execution happens in pipeline. Figure 4.5 shows two examples of hardware loops, namely for the commands ADDVV and ADDV to illustrate how long operands are processed for map and reduce operations, respectively. For executing the ADDVV command, the CU starts by fetching the first segment of each operand, one after the other. Then, with the first partition of the operands buffered, the CU starts operating over them, while fetching the next segments. When the results corresponding to the first segment of the operands are made available, the CU stores them in memory while operating over the following segments. This process repeats until the operands are entirely consumed. In contrast, during the execution of the ADDV command, only the reading of the operands and the execution phases overlap. After the last partition of the operands is retrieved, the cache CU keeps executing until there are no operations in the pipeline, and only then, the scalar result of the reduce operation is stored.

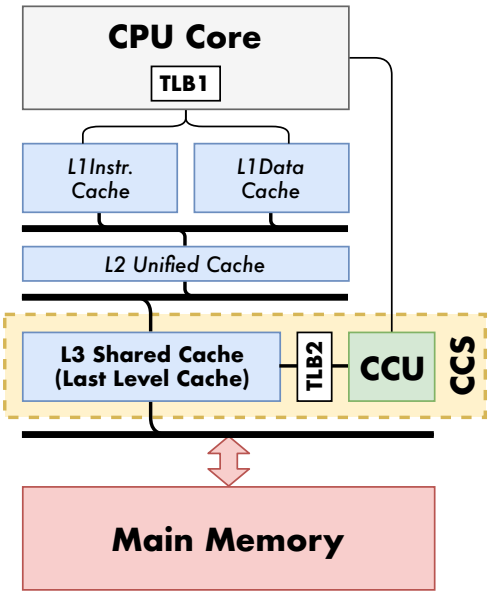


Figure 4.6: Schematics of the system modulated using gem5, featuring an OoO processing core; a non-blocking inclusive multilevel cache system with Miss Status Holding Register (MSHR) and Write Buffer (WB) for read and write misses using a MOESI coherence protocol; the devised CCS; and an address decoder responsible for forwarding requests to the memory hierarchy or the CCS depending on the address.

4.3 Architectural simulation with gem5

To validate the system functionality, a first approach, using the simulator gem5, was adopted. The primary goal of this exercise was to evaluate the impact of the CCS on a real system, featuring an OoO CPU with virtual memory support and a multi-level cache system with realistic coherence protocols.

Recalling the schematics of the system illustrated in Figure 4.1, the modulated system reproduces the originally designed architecture in all the aspects, as shown in Figure 4.6. The OoO processing core can be configured to behave like an x86 CPU from Intel or an ARM core (among other architectures), providing a realistic approximation to a real processor. The cache hierarchy is non-blocking, inclusive, has MSHR and WB for read and write misses, and uses the MOESI coherence protocol. It has three levels: a first level of private separated instructions and data caches; a private unified second cache level; and a shared unified third level of cache. Additionally, each cache level can be configured regarding its size, associativity, tag latency, data latency, response latency, number of MSHR, and maximum number of accesses per MSHR. Between the core's data port and the L1 data cache, there is an address decoder responsible for forwarding data requests to the memory subsystem or the CCS Programming Interface (PI), depending on the address. The system also features a main memory module, which emulates a Double Data Rate (DDR) Synchronous Dynamic Random-Access Memory (SDRAM) memory, and a second TLB, managed by the CCS, that is necessary for translating the addresses of the operands and the result for operations performed *near-cache*.

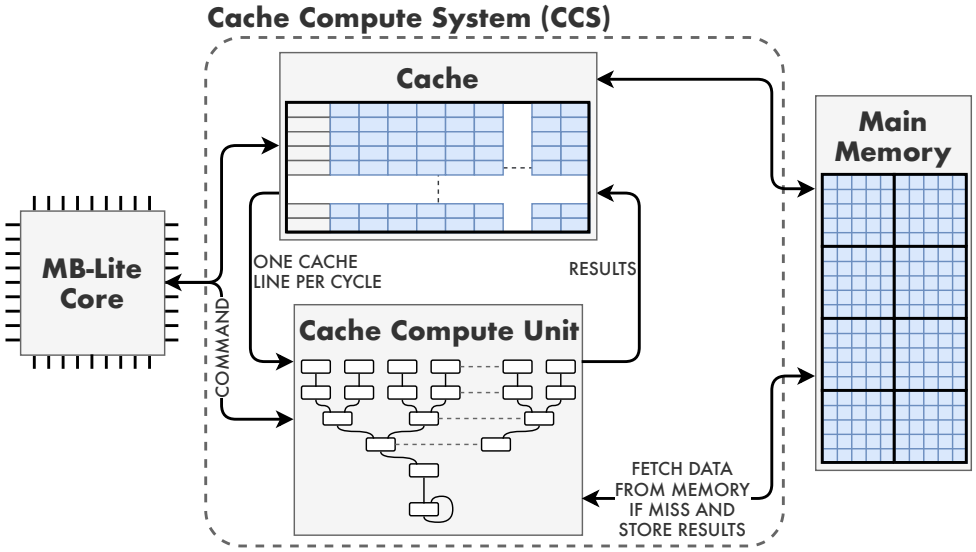


Figure 4.7: Schematics of the simplified system integrating the CCS implemented in FPGA. The cache hierarchy was reduced to a direct mapped single level cache; The OoO processing core was replaced by an MB-LITE soft-core; the main memory was built with FPGA native BRAMs.

4.4 FPGA implementation

Because architectural simulation does not allow to evaluate the hardware and the energy requirements for the system to operate, an RTL prototype for FPGA deployment was developed. Due to the hardware limitations and the intricacies associated with RTL development, several simplifications regarding the original design were made. Although the CCS was designed to be implemented with a conventional multi-level cache system, to prototype the architecture in FPGA the cache hierarchy was reduced to a direct mapped single level of cache. The processor used to serve as host in the system was not an OoO processor, but an in-order, 5 stage pipeline MB-LITE soft-core. And instead of using a conventional DRAM to act as main memory, a storage support made of FPGA native BRAMs was used, avoiding dealing with the DRAM controller protocols. Figure 4.7 depicts the schematics of the simplified system.

4.4.1 Cache architecture

The cache system used in the RTL prototype consists of a single-level, direct mapped cache with the policies write-through and write-not-allocate. It was implemented with FPGA BRAMs, and it has a 1 cycle latency for both reading and writing operations. The cache has two ports: one for reading requests from the processor and load missing lines from the main memory; and another to receive reading requests from the cache CU. The cache CU does not issue write requests to the cache, since any misses of operands are directly fetched from the main memory and not cached, avoiding conflicts between the processor's data and the cache CU operands. Although the cache was described to be fully configurable,

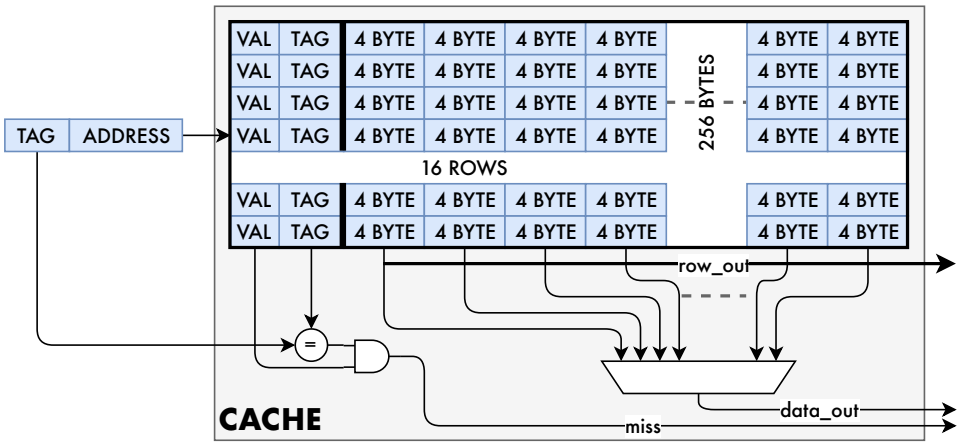


Figure 4.8: Simplified schematics of the implemented cache internal structure. Each cache line is 2048 bit long, and there are in total 16 lines. Per each line, there are 5 additional bits for control: 4 for the tag and 1 validity bit.

the implemented cache had 16 lines of 2048 bits of data each. For each line, there is, additionally, 4 bits for the tag and 1 validity bit, as shown in Figure 4.8.

Figure 4.9 depicts the finite state machine implemented by the cache controller. The latency to the main memory is 1 cycle for reading or writing data. Also, the memory bandwidth only allows accessing 4 bytes per cycle, which means that to load an entire cache line, the cache controller reads one 32-bit word per cycle from the main memory until the entire line was read. When a write request is performed by the processor, as the cache policies are write-through and write-not-allocate, the data is only written in cache if the corresponding address is present, otherwise it is only written in the main memory.

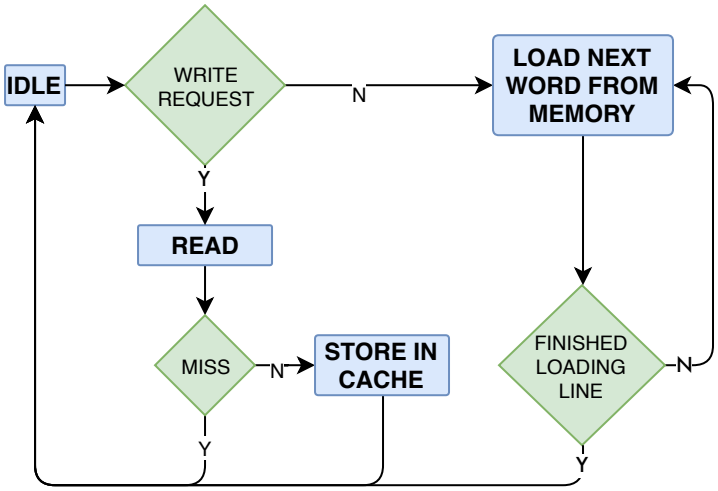


Figure 4.9: Finite state machine implemented by the cache controller. In case of a miss in cache, the missing data is loaded sequentially from the main memory (1 32-bit word per cycle). When a write request is performed by the processor, the data is only written in cache if the corresponding address is cached (cache policies write-through and write-not-allocate).

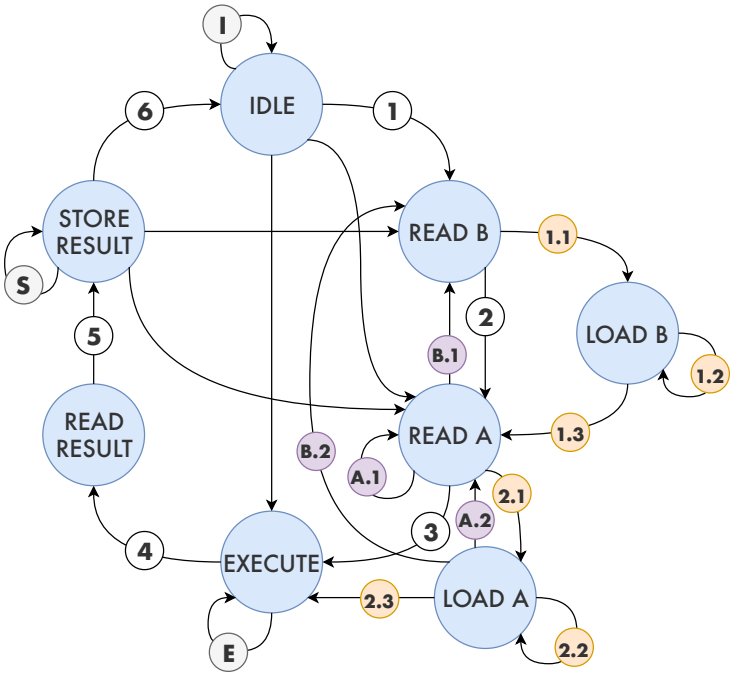


Figure 4.10: Detailed finite state machine of the CU inside the CCS. Each possible transition between states is identified by an arrow.

4.4.2 Cache CU

The CU implementation attempts to reproduce in the most reliable way the designed architecture of the CCS. However, since the main memory was built with BRAMs and all the available ports were used (one was used by the cache hierarchy and another by the CCS), the interface between the main memory and the cache CU was only half-duplex, therefore, in the same cycle, the memory interface can only be used for reading operands or writing results, but not for both. This effect is mirrored by the finite state machine of the CU, shown in Figure 4.10.

Upon receiving the signal to start executing, the CU next state is determined by the type of the operands. Should the operation include 2 vector operands and the CU will first read the operand B (1). However, in the case the command only requires one operand, the CU only fetches the operand A; and when only the constant is required (command INITC), no operands are fetched and the CU jumps directly to execution. While fetching one of the operands, the CU first checks if it is in cache. If the operand is cached, it loads it to an input buffer and proceeds to fetch the next operand (2), otherwise it retrieves it directly from memory in a sequential manner (1.1 and 1.2), before starting to fetch the second operand (1.3). When retrieving the second operand, the same logic applies, except that if the operands are bigger than the cache line and the operation is reduce-type, when a partition finishes loading, another starts being fetched (A.1 or A.2 and B.1 or B.2), while the previously loaded segments are processed. When the operands are both fully loaded, or both have segments in the buffer and the operation is a map, the

CU switches to the execute state (3 or 2.3), where it waits for the execution to cease (4). Then, the CU verifies if the addresses corresponding to the result are in cache, to determine whether the results should also be stored in cache or only in the main memory (5). Finishing storing the results, the CCS idles again (6), except if the command is map-type and the operands were not fully processed. In that case, the CU reads the next segment of the one or the two operands (depending on the command) and continues operating.

4.4.3 Integrating with MB-Lite

To integrate the CCS with the MB-LITE soft-core (although this procedure would be necessary for any core), the programming registers were mapped in the processor's address space. When the processor issues a memory request, the address decoder determines if the request is to be forwarded to the main memory or to the programming registers of the CCS. Table 4.3 details the control registers of the CCS programming interface, as well as the relative addresses where each of them is mapped. To program the CCS, the processor has to provide the following parameters:

- *Command identifier:* specifies the command to be executed in the CCS;
- *Operands length:* specifies the size of the vector operands;
- *Constant:* specifies an optional constant that is used only by the VCOP and COP commands;
- *Operand A start address:* specifies the address of the first element of the operand A;
- *Operand B start Address:* specifies the address of the first element of the operand B;
- *Result start address:* specifies the start address of the vector result (for map-type commands), or the address of the scalar result (for reduce-type commands);

Table 4.3: Control registers of the CCS' programming interface.

Relative address	Function	Register mode	Required
0x00	Command ID	Read/write	Always
0x04	Operands length	Read/write	VOP2, VOP1, VCOP
0x08	Constant	Read/write	VCOP, COP
0x0c	Op. A address	Read/write	VOP2, VOP1, VCOP
0x10	Op. B address	Read/write	VOP2
0x14	Result address	Read/write	Always
0x18	Op(s). stride	Read/write	Always
0x1c	Exec. mask	Read/write	Always
0x20			
0x24	*Reserved*		
0x28	Exec. start	Read/write	Always
0x2c	Readiness	Read-only	*Does not apply*

- *Operands' stride:* specifies the step between two valid elements of the operands (e.g., for a contiguous vector the stride is 1, whereas for a vector which only the even indexes are valid the stride is 2);
- *Execution mask:* specifies which elements are valid for a given operation depending on the stride (e.g., for a vector with 8 elements with stride 2 the execution mask is "10101010").

A Programming framework

To program the different CCS programming registers, a convenient framework was developed requiring the developer to know the minimum about the low-level programming flow of the CCS. It allows the developer to specify some of the required parameters to configure the execution environment, calculates the remaining required fields, and stores the configurations in the addresses where the programming registers of the CCS are mapped. The programming framework offers the following functions to program and control the CCS, together with macros to define the CCS commands:

- `__ccs_setup()`: takes as arguments the command identifier, the length of the operands, an optional constant, the addresses of the operands and of the result, and the stride; it also calculates the software mask and programs the CCS with the required parameters;
- `__ccs_start()`: signals the CCS to start the execution of the command previously programmed, by writing a register of the PI (*Exec. start*);
- `__ccs_check()`: checks (nonblocking) if the CCS is ready to execute by reading a register of the PI (*Readiness*), i.e., if it is not busy executing a command, and returns a logic value;
- `__ccs_wait()`: blocks the execution until the CCS is ready to accept a new command.

Listing 4.1 illustrates an example of C code used to program the CCS to add two vectors, simultaneously execute a routine in the processor, and wait for the CCS to complete.

4.5 Summary

Throughout this section, the architectural details of the CCS were presented, as well as the necessary mechanisms to validate and integrate it with an existing system featuring a processing core and a memory subsystem with a cache system.

The CCS was designed to support the most common map and reduce parallel processing patterns, found on applications from a diverse set of domains, such as clustering, machine learning, cryptography, image processing, biomedicine, and linear algebra. The cache CU within the CCS comprehends 2 levels

Listing 4.1: Example of command being issued to the CCS. After programming the CCS, the processor executes some other routine that does not require to access its results, and before accessing the memory positions where the results are stored, a blocking call is executed that checks if the CCS is ready (if the previous command has finished).

```
1  int opa[VEC_SIZE], opb[VEC_SIZE], res[VEC_SIZE];
2
3  // program the CCS
4  __ccs_setup(ADDVV, VEC_SIZE, 0, opa, opb, res, 1);
5
6  // start the execution
7  __ccs_start();
8
9  routine1();
10
11 // wait until previous command completes
12 __ccs_wait();
```

to deal with map-type operations, whereas the remaining levels adopt a tree-like shape structure and are meant to process reduce-type operations. In total, the CCS supports 48 different arithmetic, shift, logic, and move commands.

When fetching the operands and storing the results from and into memory, the CCS makes use of the existing cache structures and coherence mechanisms. To load an operand from memory, the CCS first issues a data request for an entire line to the cache. If the requested data is not cached, a request is sent to the main memory. However, when the data arrives to the CCS, it is not stored in cache, avoiding cache pollution and decreasing the number of collisions caused by conflicts with the data requested by the processing core.

The cache CU also supports processing of nonaligned operands or stride patterns. For that, it uses an execution mask that is responsible for controlling which elements of the result of a map-type operation are stored, and which elements are reduced in a reduce-type operation (influencing the operation performed by each of the ALUs in the reduction tree of the cache CU).

To perform address translation of virtual addresses into physical addresses (which represents one of the biggest challenges of NDP), the CCS replicates the hardware responsible for managing virtual memory in the processor, which presents advantages over the proposals made by early solutions, such as it does not limit the maximum size of the operands.

In case the operands cannot be entirely processed by the CCS at once because they exceed the maximum supported size by the cache CU, they are divided into segments which are fetched independently. Because the architecture is highly pipelined, the partitions of the operands can be processed in pipeline, i.e., while the second part of the operands is being fetched, the first part is being processed, and while the second parts are being processed, the first part of the result is being stored

back into memory (in case of a map-type operation).

The two methodologies used to implement the devised architecture were presented. Firstly, a simulation model that allowed to verify the correct behavior of the system was developed using gem5. In this system, the CCS was integrated with an OoO processing core and a realistic memory subsystem, composed by a multi-level cache system and a DRAM memory. Secondly, an RTL implementation of the CCS, an MB-LITE soft-core, a direct mapped single level write-through/write-not-allocate cache, and a main memory built with FPGA native BRAMs were developed.

5

Experimental Results

Contents

For implementation on FPGA, 2 soft-processors were considered: PULPino and MB-LITE. Therefore, two systems, each one featuring a different soft-core, were implemented, and their operating frequencies and hardware resources compared. Due to time limitations, the CCS was not integrated with the PULPino core, which was only used as a reference for performance comparison. On the other hand, the MB-LITE soft-core and the CCS were fully integrated into a system alongside a memory subsystem composed of a single cache level and a main memory made of FPGA native BRAMs. Several micro-benchmarks were executed on both systems, showing the impact of the CCS versus the sequential version of the commands running on the processors. Furthermore, a set of benchmarks was adapted to use the CCS, allowing to estimate its impact on real applications.

5.1 Area and power requirements comparison

Table 5.1(a) shows the hardware resources and operating frequencies for the PULPino core and the CCS when implemented on a Xilinx ZYNQ-7 ZC706 Evaluation Board. Table 5.1(b) summarizes the hardware and time constraints for the system featuring the MB-LITE, when implemented on a Xilinx VC709 Development Board featuring a XC7VX690T Virtex-7 FPGA.

Because the CCS implements a massive quantity of functional units, the area required to implement it is superior to the one needed by both the PULPino and the MB-LITE. As a consequence of the hardware requirements, it is also expected the dynamic power of the CCS to be higher.

When considering PULPino, the CCS requires approximately 3 times more slice LUTs, and about 20 times more Digital Signal Processings (DSPs). On the other hand, for the same frequency, it reveals a lower total power consumption. Furthermore, the CCS operates at a maximum frequency of 132 MHz, while the PULPino operation is limited to 40 MHz.

Minor updates to the CCS implementation after comparing it with PULPino caused a slight increase of hardware resources, as shown in Table 5.1(b). The MB-LITE core is approximately 3.5 times smaller than PULPino, and therefore one order of magnitude smaller than the CCS. In contrast, the static power consumption of both systems is very similar, differing on the dynamic power consumption.

5.2 Speedup and energy efficiency improvements

To evaluate the performance of the CCS and compare it with the considered soft-processors, several benchmarks and micro-benchmarks were run on both systems. Firstly, a battery of micro-benchmarks was developed and compiled for the reference processors. Secondly, the same set of micro-benchmarks was tuned manually at assembly level for the MB-LITE, achieving the most optimized version of each micro-benchmark. Finally, a set of benchmarks was adapted to use the CCS, allowing to evaluate its

Table 5.1: Timing and hardware resources summary of the reference processors and proposed CCS. When integrating the CCS with the MB-LITE, the clock frequency decreases by approximately 30 % (regarding the individual systems) due to routing issues. Additional CCS-to-MB-LITE pipeline stages were not introduced because it would impact the performance of the MB-LITE.

(a) Hardware resources and timing constrains of PULPino and CCS. These results were obtained using Xilinx Vivado 2016.4 and a ZYNQ-7 ZC706 Evaluation Board.

System	Timing	Resources				Total Power Consumption
		Slice LUTs	Slice Registers	Memory	DSP	
PULPino	40MHz	22193 (10.15%)	13472 (3.08%)	16 (2.94%)	10 (1.11%)	1.91W
CCS	132MHz	59749 (27.33%)	10464 (2.39%)	0 (0.00%)	192 (21.33%)	5.05W
	40MHz					1.69W

(b) Integration with MB-LITE. These results were obtained using Xilinx Vivado 2018.1 and a Xilinx VC709 Development Board featuring a XC7VX690T Virtex-7 FPGA.

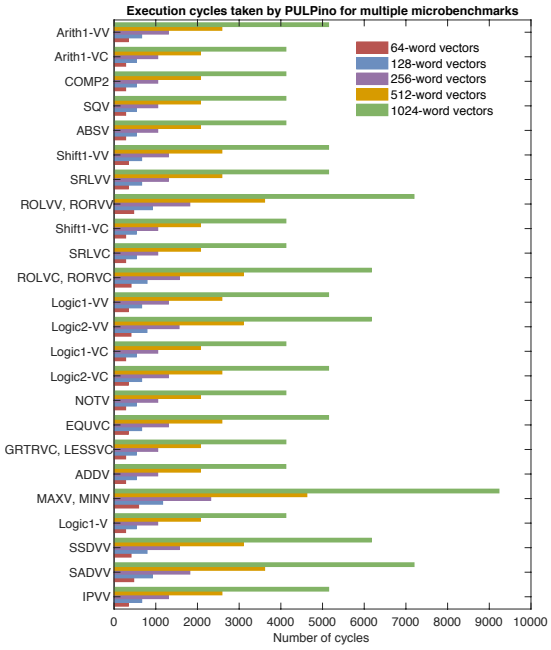
		MB-LITE + Cache		CCS		Complete System	
		Utilization	%	Utilization	%	Utilization	%
Resources	LUT	6351	2	87675	21	91596	22
	LUTRAM	2053	2	7	1	7	1
	FF	2483	1	13014	2	13368	2
	BRAM	34	3	128	9	162	11
	DSP	3	1	192	6	195	6
Frequency [MHz]		100		100		67	
P [W]	Static	0.329		0.345		0.345	
	Dynamic	0.150		0.793		0.612	

impact on real applications. To avoid the influence of the operating clock frequency, all results are presented in clock cycles instead of execution time. Because PULPino was only used as a reference for comparison, all the results were extrapolated. In contrast, the results involving the MB-LITE and the CCS were measured directly, since the CCS was completely integrated with it.

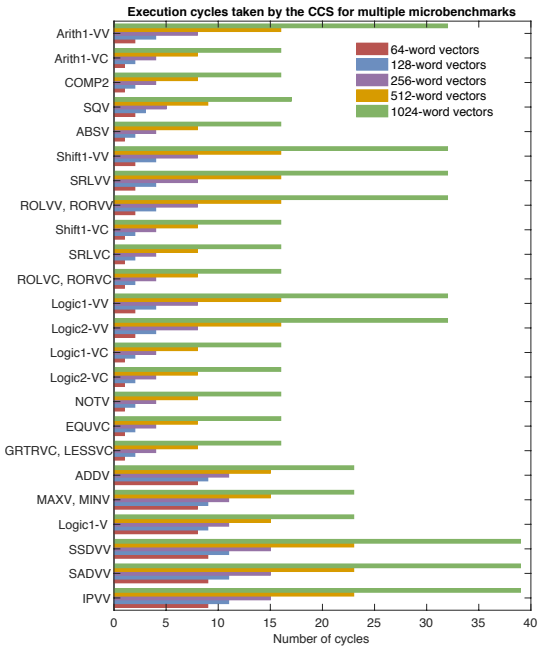
5.2.1 Micro-benchmarks

The considered micro-benchmarks match the commands supported by the CCS, i.e., for each CCS command there is an equivalent sequential version that executes on the processor. The microbenchmarks were compiled by the compilers provided for each of the cores (the RISC-V toolchain for PULPino and the mb-gcc for MB-LITE).

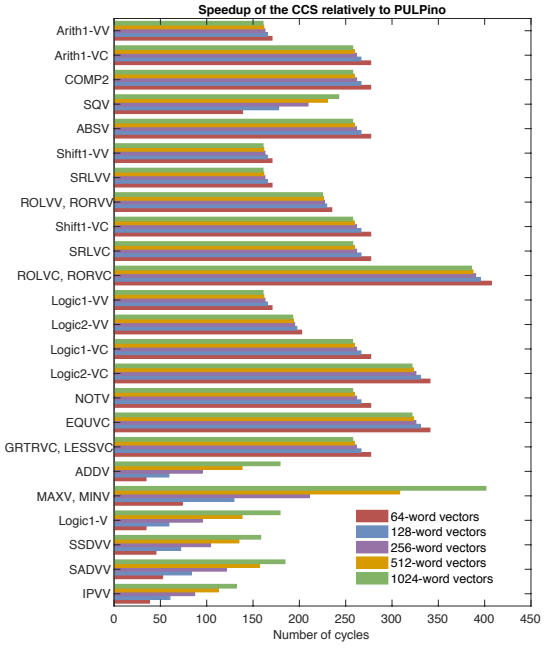
Figures 5.1(a) and 5.1(b) show the execution time of the micro-benchmarks when executed on PULPino and on the CCS for 5 operand sizes. Such results are compared in Figure 5.1(c), which shows the speedup achieved by performing the operations in the CCS rather than in the processor. Moreover, Figure 5.1(d) illustrates hypothetical results considering the existence of a 256-bit SIMD unit in PULPino, which are obtained by dividing the PULPino execution time by 8 (8 words of 32-bit each are operated at



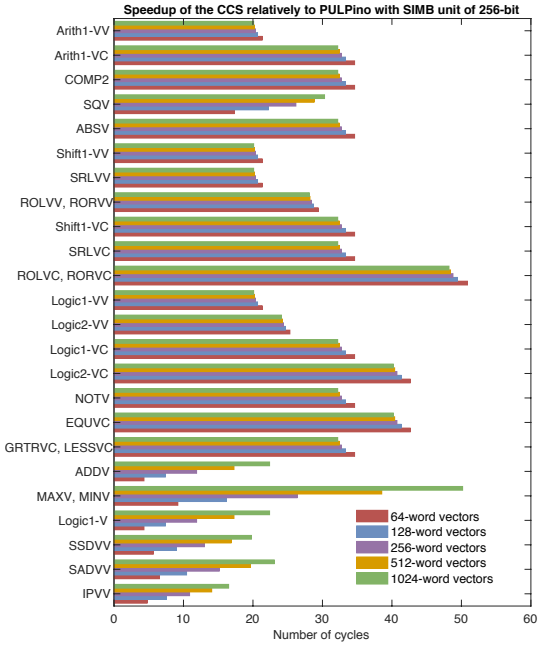
(a) Execution cycles taken by PULPino.



(b) Execution cycles taken by the CCS.



(c) Speedup over PULPino.



(d) Speedup over PULPino with a 256-bit SIMD unit.

Figure 5.1: Execution time comparison between the micro-benchmarks executing on PULPino and the CCS. *Arith1* operations are ADD, SUB and MUL; *Shift1* includes SLL and SRA; *Logic1* encapsulates AND, OR and XOR; *Logic2* is composed by NAND, NOR and XNOR.

the same time).

On average, for 1024-element operands, the CCS achieves a speedup of $235\times$ over PULPino, and, in general, the speedup is comprised between $132\times$ (for IPVV) and $401\times$ (for MAXV, MINV). Also, it is noticeable that the speedup for reduce commands increases significantly with the size of the operands, while maps are less affected. This is due to the overhead of storing the results. Since the cache has the writing policy write-through, the results are written sequentially in memory, and while maps produce results with the same size than the operands, reduce-type commands produce a scalar result, therefore they do not present such overhead.

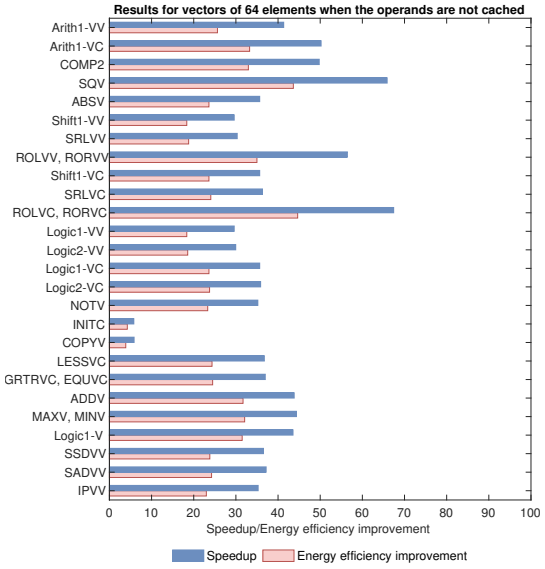
Some map-type commands decrease the speedup with the size of the operands. That is due to the overhead of storing the vector results sequentially in the main memory (the cache has the writing policy write-through and due to the WB limited capacity, the CCS does not use it). When the size of the operands increases, the overhead associated with storing the results in the main memory increases faster than the load and processing times. As a consequence, according to Amdahl's law, the storing phase will limit the achievable speedup.

Figure 5.2 shows the results comparing the execution of the micro-benchmarks when using the CCS or the MB-LITE. Figures 5.2(a) and 5.2(b) consider vectors of length 64, each element with 32-bits, and 5.2(c) and 5.2(d) consider vectors of 128 elements. For each operand size, it was considered the case where the operands are in cache and the case where they are only in the main memory.

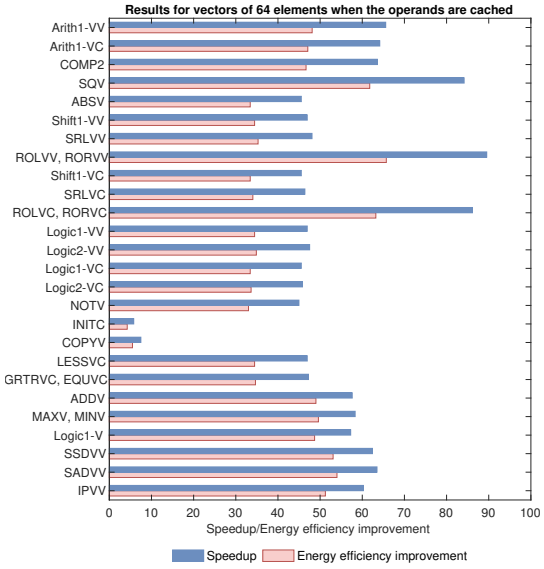
Considering 128-element vector operands, the CCS executes, on average, $38\times$ faster than the MB-LITE, when the data is not cached. When data is cached, the average speedup is $63\times$ and, in general, the speedup ranges from $18\times$ (obtained for COPYV micro-benchmark) to $94\times$ (obtained for ROLVV, RORVV micro-benchmarks). Also for 128-element operands, on average, the CCS spends $23\times$ less energy when the data is only in memory and $45\times$ when data is in cache. The energy efficiency improvements, in general, are comprised between $11\times$ and $67\times$. Similarly to the results obtained with PULPino, the reduce-type commands benefit the most from an increased vector length, although, in general, all commands benefit from wider vectors. To verify this effect, 4 micro-benchmarks were chosen to evaluate the impact of increasing the vector length on the speedup. As shown in Figure 5.3, the speedup of map-type commands is bounded by the overhead of storing all the elements of the result sequentially. On the other hand, the reduce-type commands are not restrained by the store of the results (since only a scalar result is produced).

5.2.2 Assembly optimized micro-benchmarks for MB-LITE

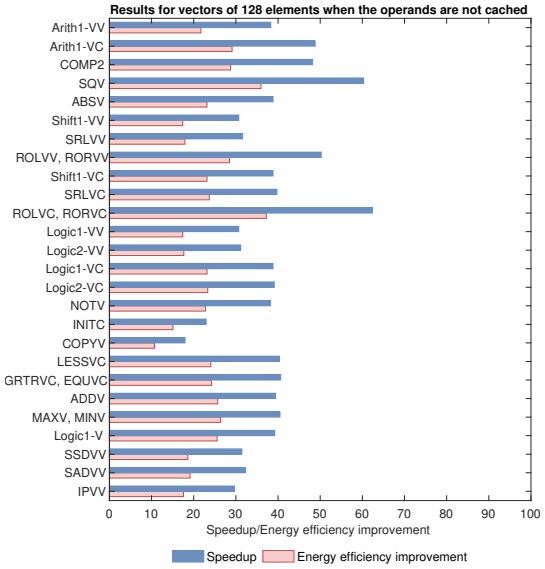
Although the achieved speedups are mainly explained by the parallel access and processing of the vector operands elements, the loop control overhead associated with loops being executed on the processor is also removed when using the CCS. Moreover, such results are limited by the compiler



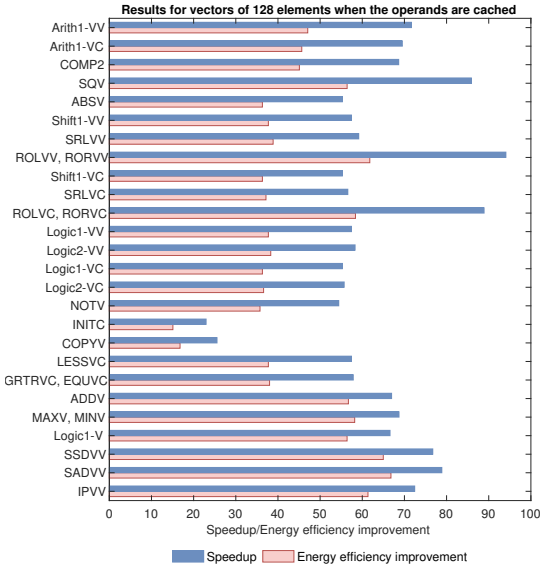
(a) Operands with 64 elements not in cache.



(b) Operands with 64 elements in cache.

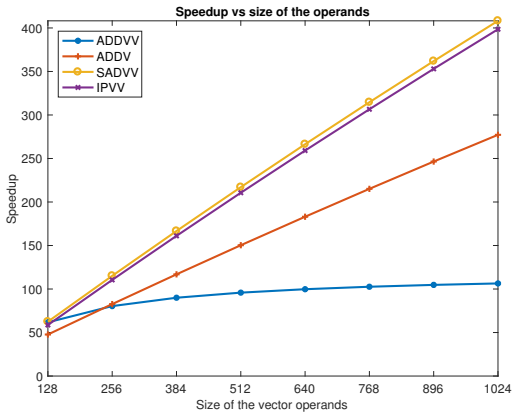


(c) Operands with 128 elements not in cache.

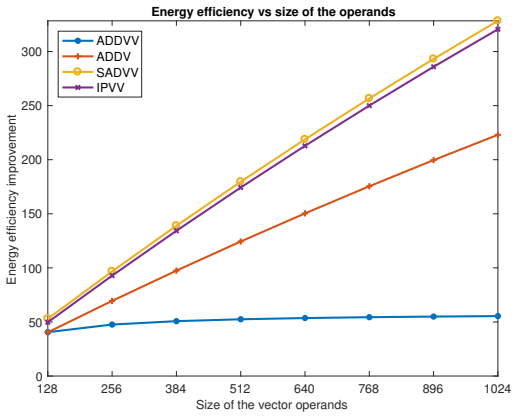


(d) Operands with 128 elements in cache.

Figure 5.2: Speedup and energy efficiency improvements obtained using the MB-LITE + CCS versus using only the MB-LITE for 4 operand sizes. *Arith1* operations are ADD, SUB and MUL; *Shift1* includes SLL and SRA; *Logic1* encapsulates AND, OR and XOR; *Logic2* is composed by NAND, NOR and XNOR.



(a) Speedup versus operands size.



(b) Energy efficiency improvements versus operands size.

Figure 5.3: Speedup and energy efficiency improvements obtained using the CCS versus using only the MB-LITE when the operands size vary between 128 and 1024 elements and all the data is present in cache.

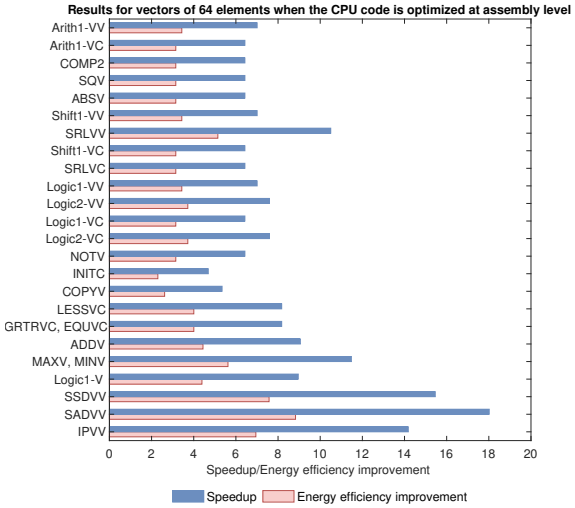
efficiency to optimize the code. Therefore, to better evaluate the impact of the CCS on the system, it was necessary to reduce the loop control to the minimum on the MB-LITE side, which could only be achieved by tuning the micro-benchmarks at assembly level. The optimized assembly code for the micro-benchmark ADDV is shown in appendix B.

Figure 5.4 illustrates the speedup and energy efficiency improvements achieved by executing a set of micro-benchmarks on the CCS versus the assembly optimized MB-LITE code. Comparing these results with the ones obtained for the micro-benchmarks compiled using the MicroBlaze compiler, the loop control overhead consumes more than 70% of the execution time for the SADVV micro-benchmark. However, it is still possible to observe that the map-type commands are limited by the result storing overhead, and the reduce-type commands benefit the most from increasing the size of the operands.

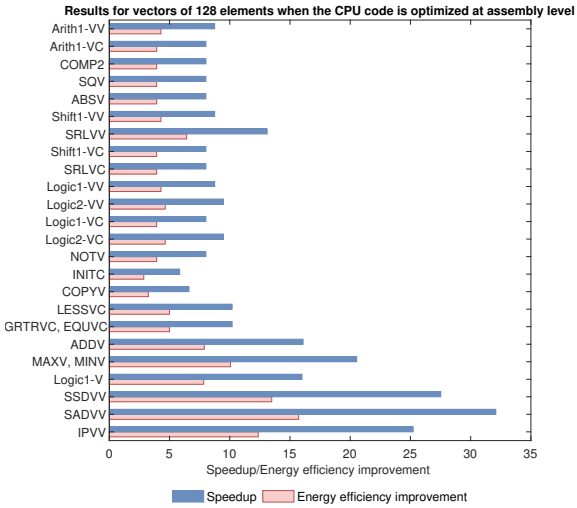
5.2.3 Application benchmarks

Finally, to better understand the impact of the CCS on real applications, a set of benchmarks was adapted to use it. When comparing the CCS with PULPino, 2 clustering applications were considered: the K-Nearest Neighbors (KNN) and the K-Means. Figure 5.5(a) shows the speedups achieved by using the CCS versus using only the PULPino core, while Figure 5.5(b) considers a PULPino core featuring a 256-bit SIMD unit. The data-sets used to run these applications were provided by the Machine Learning Repository of the University of California, Irvine (UCI) [41–44]. Tables 5.2(a) and 5.2(b) list the data-sets and applications parameters.

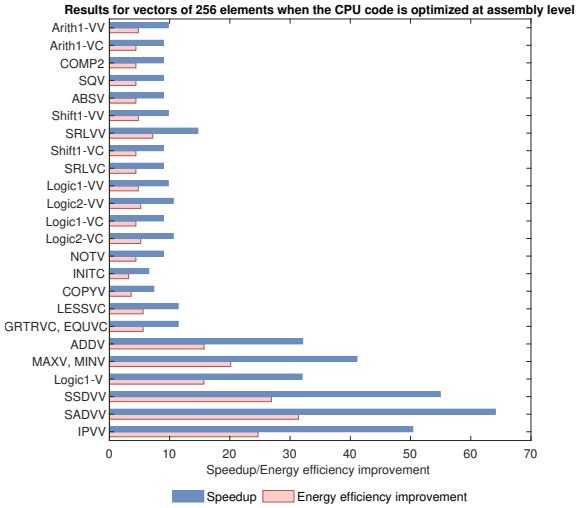
The same exercise was done for the MB-LITE soft-core. For evaluating the impact of the CCS on



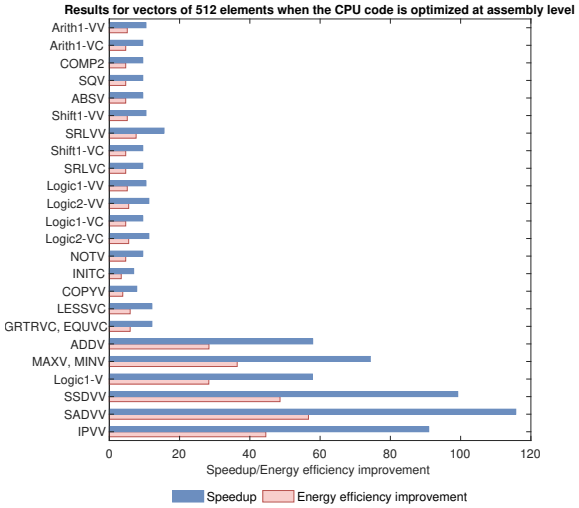
(a) Operands with 64 elements.



(b) Operands with 128 elements.



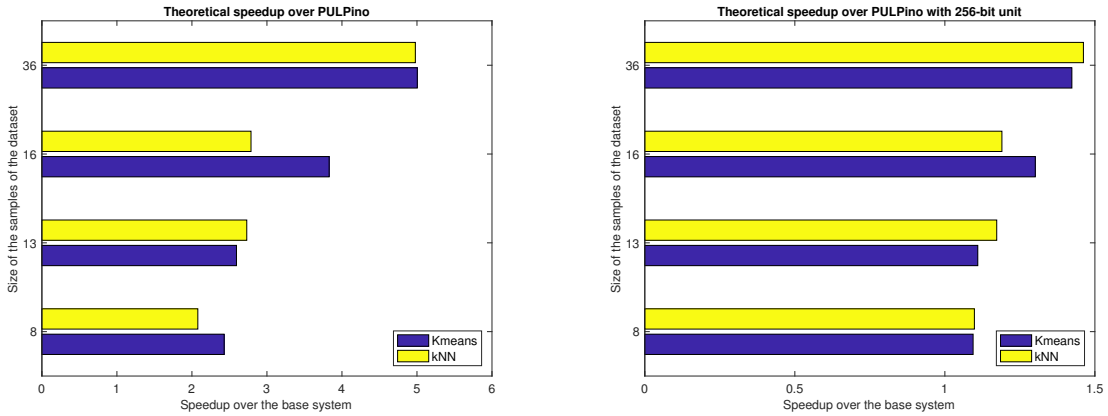
(c) Operands with 256 elements.



(d) Operands with 512 elements.

Figure 5.4: Speedup and energy efficiency improvements obtained using the CCS versus using only the MB-LITE for 4 operand sizes when the operands are present in cache. The reference code executing in the MB-LITE was optimized by hand at assembly level. *Arith1* operations are ADD, SUB and MUL; *Shift1* includes SLL and SRA; *Logic1* encapsulates AND, OR and XOR; *Logic2* is composed by NAND, NOR and XNOR.

the system featuring the MB-LITE, 3 applications were adapted to use the processing structure: KNN, matrix multiplication and linear regression. For the KNN algorithm, a set of 64 control samples was used, the number of nearest neighbors was 4, each sample featured 64 coordinates and only 1 sample was



(a) Comparison with scalar PULPino.

(b) Comparison with PULPino featuring a 256-bit SIMD unit.

Figure 5.5: Speedup achieved by using the CCS over PULPino for 2 application benchmarks: KNN and K-Means.

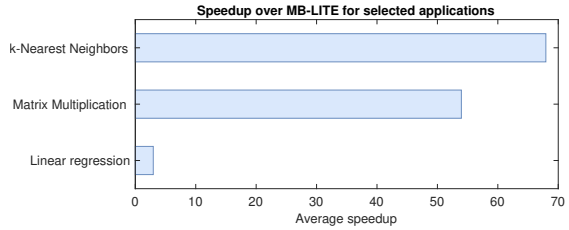


Figure 5.6: Speedup obtained for 3 application benchmarks when using the CCS rather than using only the MB-LITE core. For the KNN algorithm, a set of 64 control samples was used, the number of nearest neighbors was 4, each sample featured 64 coordinates and only 1 sample was classified; for the matrix multiplication benchmark, 2 64×64 matrices were considered; and the linear regression was performed based on 2 64-coordinate euclidean points.

categorized; for the matrix multiplication benchmark, 2 64×64 matrices were considered; and the linear regression was performed based on 2 64-coordinate euclidean points. Figure 5.6 shows the speedup achieved for each of the considered applications.

Table 5.2: Parameters used on the runs of the application benchmarks for PULPino.

(a) Parameters of the several runs of the KNN benchmark.

Number features	**8**	**13**	**16**	**36**
Control samples	410	148	70	70
Classify samples	30	30	30	30
Number of classes	5	3	26	26
K	4	4	4	4

(b) Parameters of the several runs of the K-Means benchmark.

Number of features	**8**	**13**	**16**	**36**
Number of samples	440	178	100	100
Change penalty	75	3	100	100
Threshold	1	1	1	1
Number of clusters	5	5	26	26
Iterations	24	35	8	5

Listing 5.1: C code responsible for programming the CCS to calculate the euclidean distance between 2 points in the context of the KNN benchmark.

```
// calculate distances
for (i = 0; i < N; i++) {
    __ccs_setup(SSDVV, N, 0, b, (a + N * i), c + i, 1);
    __ccs_start();
    __ccs_wait();
}
```

Listing 5.2: C code responsible for programming the CCS to calculate the dot product between two vectors in the context of the matrix multiplication benchmark.

```
for (i = 0; i < 1; i++) {
    for (j = 0; j < MATRIX_SIZE; j++) {
        __ccs_setup(IPVV, MATRIX_SIZE, 0, (mta + i * MATRIX_SIZE),
            (mtb + j * MATRIX_SIZE), mtc, 1);
        __ccs_start();
        __ccs_wait();
    }
}
```

The results shown on Figure 5.6 indicate that the speedup for the linear regression benchmark is only $3\times$, while for the KNN and matrix multiplication benchmarks the speedups achieve $68\times$ and $54\times$, respectively. The cause of such difference is related to the workload of the linear regression benchmark, which reflects the number of times that the CCS requires to be re-programmed. As shown in Listing 5.3, the linear regression benchmark requires to re-program the CCS $6\times$, while the other considered benchmarks require the CCS to be programmed only once per iteration (Listings 5.1 and 5.2).

When comparing the speedups of the common benchmark KNN taking as reference PULPino and MB-LITE, the performance improvements over MB-LITE are significantly higher than PULPino. This result has 3 main revelations: (1) both the ISA and the architecture implemented by the MB-LITE are different from the ones implemented by PULPino, which produce code more optimized than mb-gcc

Listing 5.3: C code responsible for programming the CCS to execute the operations required by the linear regression benchmark.

```
__ccs_setup(ADDV,  VEC_SIZE, 0, x, NULL, s_xx,   1); __ccs_start(); __ccs_wait();
__ccs_setup(ADDV,  VEC_SIZE, 0, y, NULL, s_yy,   1); __ccs_start(); __ccs_wait();
__ccs_setup(SQV,   VEC_SIZE, 0, x, NULL, z,      1); __ccs_start(); __ccs_wait();
__ccs_setup(ADDV,  VEC_SIZE, 0, z, NULL, s_xx_s, 1); __ccs_start(); __ccs_wait();
__ccs_setup(MULVV, VEC_SIZE, 0, x, y,    z,      1); __ccs_start(); __ccs_wait();
__ccs_setup(ADDV,  VEC_SIZE, 0, z, NULL, s_xy,   1); __ccs_start(); __ccs_wait();
```

and the MicroBlaze ISA; (2) the compilers for both processors are different, and therefore there are no indicators on how the optimizations to the code running on the processors affect the execution; (3) while the timing results for the MicroBlaze versus the MicroBlaze using the CCS were obtained running the benchmarks on the system, the timing results for PULPino using the CCS were extrapolated based on Amdahl's law, which makes them less accurate.

5.3 Summary

To evaluate the impact of the CCS on a system, the devised architecture was compared with 2 setups featuring 2 different soft-processors: PULPino and MB-LITE. PULPino was only used as a comparison reference, while MB-LITE was fully integrated with a memory subsystem (containing a cache system) and the CCS. In general, the CCS implementation occupies significantly more area than either of the soft-cores alone. Nevertheless, it presents a static power consumption similar to the soft-cores.

A set of micro-benchmarks was run on both systems using only the CPU or only the CCS, and it was observed that for the system featuring the PULPino, for 1024-element operands, the speedup achieved by using the CCS is comprised between $132\times$ and $401\times$. When compared with the MB-LITE soft-core, the CCS presents an average speedup of $38\times$, for 128-element operands when data is not present in cache. On the other hand, when data is cached, the average speedup is $63\times$. The energy efficiency improvements are superior to $11\times$, and can be as high as $67\times$. Furthermore, all micro-benchmarks benefit from increasing the size of the operands, in particular the reduce-type commands.

Reducing to the minimum the loop control overhead on MB-LITE side allowed to better understand how the CCS influences the overall execution time and energy consumption. It was concluded that the loop control represents approximately 70% of the execution time for the SADVV benchmark. Nevertheless, all the previous conclusions regarding the relation between the size of the operands and the performance and energy efficiency improvements are still valid.

Finally, to access the performance of the CCS for real applications, a set of benchmarks was adapted to use the processing structure. For the system featuring the MB-LITE, the KNN benchmark revealed the highest speedup ($68\times$), while when compared with PULPino, the CCS could only improve $5\times$ the performance. This contrast can be justified by the different ISAs and architectural optimizations of the processors, which reflect different execution times; the compilers are also different, which causes the codes to be optimized differently; and the fact that the results regarding PULPino were extrapolated makes them less accurate than the ones regarding MB-LITE, which were obtained by executing the code directly on the system and measuring the elapsed cycles.

6

Conclusions

To address the limitations of the memory subsystems in the nowadays compute systems, a mechanism capable of performing computation *near-cache* was proposed. Although the devised system can be coupled with any cache line, it is specially fit to operate near the LLC. The LLC cache is shared among processing cores, which makes it easier for a core to access data that was ordered to be processed by another. The LLC is the largest of the cache levels, which reduces the probability of miss. Furthermore, in case of cache miss, the LLC latency to the main memory is the shortest among cache levels.

The devised CCS was designed to support the common map and reduce parallel patterns, commonly found on data-driven applications, and operate closer to where the data is actually stored. Therefore, it eliminates the overhead of transferring large data-sets across the memory subsystem, while exploring the intrinsic parallelism of the cache structures and removing the overhead of software-side loop control. The CCS also provides support for hardware loops, in case the vector operands are longer than cache lines. In total, the proposed structure provides an ISA of 48 commands inspired by common operations, and found in algorithms belonging to different domains, such as clustering, machine learning, cryptography, image and video processing, biomedicine, and algebra.

The correct behavior of such system was validated by architectural simulation using gem5, and the system was implemented in FPGA alongside an MB-LITE processing core with a simplified memory subsystem, featuring a cache level. When compared with MB-LITE standalone, for vectors of length 128 and 32-bit operands, the CCS allows performance improvements ranging from $18\times$ to $94\times$ and energy efficiency improvements from $11\times$ and $67\times$. The contributions of the CCS and this particular implementation were published in the 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'18) [45]. Furthermore, the devised architecture was compared with a PULPino soft-core. The main reason for this comparison is the recent interest of the community in RISC-V implementations for embedded systems. When compared with PULPino, the CCS has shown performance improvements between $132\times$ and $401\times$.

Several applications were also adapted to use the CCS, allowing to assess its impact for real-world applications. When comparing with PULPino, the KNN and KMeans applications were considered, whereas KNN, Matrix Multiplication and Linear regression were the benchmarks used for the system integrating an MB-LITE and the CCS. The benchmark KNN, in particular, presented a speedup of $68\times$ when running in the system featuring the MB-LITE and the CCS versus the MB-LITE standalone.

The architecture presented in this thesis represents an important contribution, with the current results already suggesting that relevant performance and energy efficiency improvements are achievable. However, further efforts should be made both to optimize the solution and to validate its integration with real systems.

When validating the architecture by simulation, the system-call emulation mode of gem5 was used (rather than the full system mode), which does not simulate an operating system on the top of the

hardware system. Therefore, the only process running on the system is the one that uses the CCS, not allowing to assess eventual security conflicts between concurrent processes that may try to interact with the CCS simultaneously. The number processes being executed and the usage of the memory subsystem may influence the performance of the CCS, factors that cannot be weighted without having an operating system running on the simulated hardware. Furthermore, the simulated system featured only one processing core, which is not accurate for the majority of real systems. Therefore, a more accurate system should be modeled and simulated in full system mode using gem5, allowing to tackle these issues.

Another critical issue is the large quantity of resources required to implement the CCS, which represents a significant area overhead. Consequently, adaptations to the current structure to reduce the number of hardware resources without sacrificing the performance are required, such as multiplexing the resources in order to change their purpose in time, rather than using different components for each task.

Extending the functionality of the CCS to receive groups of commands in the form of a kernel rather than only one command per execution cycle is one of the possibilities to improve even further the impact of the CCS, since the programming overhead would be reduced and the CCS could perform more complex tasks. Also, by allowing a more refined control over the configuration signals of the CCS, it would be possible to combine different functionalities of each unit, enabling to perform commands that are not provided by the ISA.

Bibliography

[1] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *ISCA*. ACM, 2015, pp. 336–348.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *HPCA*. IEEE Computer Society, 2017, pp. 481–492.

[3] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[4] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[5] R. Balasubramanian, V. Gangadhar, Z. Guo, C. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, "Enabling GPGPU low-level hardware explorations with MIAOW: an open-source RTL implementation of a GPGPU," *TACO*, vol. 12, no. 2, pp. 21:21:1–21:21:25, 2015.

[6] P. Duarte, P. Tomás, and G. Falcão, "SCRATCH: an end-to-end application-aware soft-gpgpu architecture and trimming tool," in *MICRO*. ACM, 2017, pp. 165–177.

[7] T. Kranenburg and R. van Leuken, "MB-LITE: A robust, light-weight soft-core implementation of the microblaze architecture," in *DATE*. IEEE Computer Society, 2010, pp. 997–1000.

[8] B. L. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[9] IBM, "Understanding dram operation," 1996. [Online]. Available: http://www.archive.ece.cmu.edu/~ece548/localcpy/dramop.pdf

[10] HP, "Memory technology evolution: An overview of system memory technologies," 2008. [Online]. Available: https://support.hpe.com/hpsc/doc/public/display?sp4ts.oid=78193&docLocale=en_US&docId=emr_na-c01552458

[11] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, “Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories,” in *DAC*. ACM, 2016, pp. 173:1–173:6.

[12] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Fast bulk bitwise AND and OR in DRAM,” *Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, 2015.

[13] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Rowclone: fast and energy-efficient in-dram bulk data copy and initialization,” in *MICRO*. ACM, 2013, pp. 185–197.

[14] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *SC*. ACM, 2011, pp. 52:1–52:12.

[15] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *MICRO*. ACM, 2009, pp. 469–480.

[16] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, “Cache automaton,” in *MICRO*. ACM, 2017, pp. 259–272.

[17] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. T. II, J. Wadden, M. R. Stan, and K. Skadron, “An overview of micron’s automata processor,” in *CODES+ISSS*. ACM, 2016, pp. 14:1–14:3.

[18] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. R. Kaeli, “Multi2sim: a simulation framework for CPU-GPU computing,” in *PACT*. ACM, 2012, pp. 335–344.

[19] A. Patel, F. Afram, and K. Ghose, “Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors,” in *1st International Qemu Users’ Forum*, 2011, pp. 29–30.

[20] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 2005, pp. 41–46.

[21] M. T. Yourst, “Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator,” in *ISPASS*. IEEE Computer Society, 2007, pp. 23–34.

[22] A. Akram and L. Sawalha, “A comparison of x86 computer architecture simulators,” 2016.

[23] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, “Accuracy evaluation of GEM5 simulator system,” in *ReCoSoC*. IEEE, 2012, pp. 1–7.

[24] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA*. IEEE Computer Society, 2010, pp. 1–12.

[25] D. Sánchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *MICRO*. IEEE Computer Society, 2010, pp. 187–198.

[26] ——, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA*. ACM, 2013, pp. 475–486.

[27] M. Aleem, M. A. Islam, and M. A. Iqbal, "A comparative study of heterogeneous processor simulators," *International Journal of Computer Applications*, vol. 148, no. 12, 2016.

[28] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[29] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

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

[31] "Vexrisc-v core." [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[32] R. Logic, "Rv12 risc-v processor." [Online]. Available: https://roalogic.com/portfolio/riscv-processor/

[33] Syntacore, "Scr1 core." [Online]. Available: https://github.com/syntacore/scr1

[34] OnChipUIS, "mrisc-v core." [Online]. Available: https://github.com/onchipuis/mriscv

[35] V. C. Inc., "Orca core." [Online]. Available: https://github.com/VectorBlox/orca

[36] ——, "Orca: Fpga-optimized risc-v." [Online]. Available: https://riscv.org/wp-content/uploads/2016/01/Wed1200-2016-01-05-VectorBlox-ORCA-RISC-V-DEMO.pdf

[37] C. Wolf, "Picorv32 - a size-optimized risc-v cpu." [Online]. Available: https://github.com/cliffordwolf/picorv32

[38] A. Traber *et al.*, "Pulpino: A risc-v based single-core system," in *OpenRISC Conference, ORCONF2015*. ORCONF2015, 2015.

[39] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. Gürkayank, and L. Benini, "Pulpino: A small single-core risc-v soc," in *3rd RISCV Workshop*, 2016.

[40] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, “Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions,” *CoRR*, vol. abs/1802.00320, 2018.

[41] UCI Center for Machine Learning and Intelligent Systems, “Wine Data Set,” 1991. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/wine

[42] ——, “Wholesale customers Data Set,” 2014. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Wholesale+customers

[43] ——, “Letter Recognition Data Set,” 1991. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Letter+Recognition

[44] ——, “Statlog (Landsat Satellite) Data Set,” 1993. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Statlog+(Landsat+Satellite)

[45] J. Vieira, P. Ienne, N. Roma, G. Falcao, and P. Tomás, “Exploiting compute caches for memory bound vector operations,” in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD’2018)*, 2018.

A

RISC-V Open-Source Micro-Architectures Summary

Table A.1: Summary table presenting the main features of some state-of-the-art implementations of RISC-V.

	Rocket	ORCA	PicoRV			PULPino				mRISC-V	VexRISC-V	RV12	SCR1
			Small	Regular	Large	RISCY	RISCY w/ FPU	Zero-riscy	μ-riscy				
Plat.	FPGA/ SoC	FPGA	FPGA			FPGA				SoC	FPGA/ SoC	FPGA	FPGA
Lang.	Verilog/ C	VHDL	Verilog			System Verilog				Verilog	SpinalHDL	Verilog	System Verilog
Arch	32-bit	32-bit	32-bit			32-bit				32-bit	32-bit	32-bit 64-bit	32-bit
Ext.	I, M, A	I, M	E	I	I, M, C	I, M, C *Xpulp*	I, M, C, F *Xpulp*	I *or* E, [M], C	E, C	I	I, M	I	I *or* E, [M], [C]
FP	No	No	No	No	No	No	Yes	No	No	No	No	No	No
Cache Sup.	Yes	Yes	Yes			Yes				No info	Yes	No	No info
Area (LUTs)	No info	2353	761	917	2019	6783	11550	3150	1933	No info	2085	No info	No info
Freq. (MHz)	No info	125	714			185	No info	185	185	No info	223	No info	No info
FPGA	No info	Altera Cyclone IV	Xilinx Virtex UltraScale+			No info				No info	Artix-7	No info	No info
DMIPS (MIPS)	No info	212	No info			No info				No info	No info	No info	No info
DMIPS /MHz	No info	0.87	0.52			No info				No info	1.16	No info	No info
CMark /MHz	No info	No info	No info			3.19	3.19	2.44	0.91	No info	No info	No info	No info
Doc. (1–5)	2	4	2			5				1	3	2	2

B

Assembly Code for MB-LITE

Listing B.1: Optimized assembly code of the micro-benchmark ADDV for MB-LITE

```
1         /*operand addresses initialization*/
2         addi r6,  r0,  2048  /*size of the operands times four (4 bytes per word)*/
3         add  r7,  r0,  r0    /*vector 1 start address*/
4         add  r8,  r7,  r6    /*vector 2 start address*/
5         add  r9,  r8,  r6    /*result start address*/
6
7         /*stdio address and codes*/
8         addi r10, r0,  65280 /*stdio address*/
9         addi r11, r0,  -1    /*code for resetting counter*/
10        addi r12, r0,  -2    /*code for dumping time*/
11        addi r13, r0,  -3    /*code for terminating program*/
12
13        /*initialize operands*/
14        addi r1,  r0,  1     /*constant to initialize vector*/
15        add  r2,  r7,  r0    /*copy vector 1 start address*/
16  init: sw   r1,  r2,  r0    /*initialize element of vector 1*/
17        addi r2,  r2,  4     /*increment index of vector 1*/
18        rsub r3,  r2,  r8
19        bnei r3, init        /*while not fully initialized*/
20
21        /*force operands in cache*/
22        lwi  r1,  r7,  0
23        lwi  r1,  r8,  0
24        lwi  r1,  r7,  256
25        lwi  r1,  r8,  256
26        lwi  r1,  r7,  512
27        lwi  r1,  r8,  512
28        lwi  r1,  r7,  768
29        lwi  r1,  r8,  768
30        lwi  r1,  r7,  1024
31        lwi  r1,  r8,  1024
32        lwi  r1,  r7,  1280
33        lwi  r1,  r8,  1280
34        lwi  r1,  r7,  1536
35        lwi  r1,  r8,  1536
36        lwi  r1,  r7,  1792
37        lwi  r1,  r8,  1792
```

```
38
39          /*clear the pipeline*/
40          nop
41          nop
42          nop
43
44          sw    r11, r10, r0    /*reset the cycle counter*/
45
46          /*microbenchmark goes here*/
47          add   r1,  r0,  r0
48    proc: lw    r3,  r7,  r0
49
50          addi  r7,  r7,  4     /*increment index of vector 1*/
51          rsub  r4,  r7,  r8
52
53          add   r1,  r1,  r3
54
55          bnei  r4, proc
56
57          /*store the operation result*/
58          sw    r1,  r10, r0
59
60          /*empty the pipeline and terminate the program*/
61          /*special code written to the stdio address*/
62          nop
63          nop
64          sw    r12, r10, r0    /*dump execution time*/
65          sw    r13, r10, r0    /*terminate program*/
```